Session-Based Concurrency: Between Operational and Declarative Views

M.A Cano Grijalba

The Netherlands

2019

The work in this book has been carried out at University of Groningen as part of the doctoral studies of the author.

IPA Dissertation Series No. 2019-14
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming Research and Algorithms).

ISBN (printed): 978-94-034-2190-2
ISBN (e-book): 978-94-034-2189-6

Typeset with XeLaTeX
Printed by Ridderprint, the Netherlands
Cover designed by Tom van Akken, Ridderprint

university of
groningen

Session-Based Concurrency: Between Operational and Declarative Views

PhD Thesis

to obtain the degree of PhD at the
University of Groningen
on the authority of the
Rector Magnificus Prof. C. Wijmenga
and in accordance with
the decision by the College of Deans.

This thesis will be defended in public on

Tuesday 7th January 2020 at 11.00 hours

by

Mauricio Alejandro Cano Grijalba

born on 13th July, 1992
in Cali, Colombia

Summary

The analysis of message-passing programs remains an open challenge for Computer Science. In particular, certifying a program's conformance with respect to some intended protocols is an active research problem. These programs can be specified by considering an *operational* view, in which explicit sequences of actions describe the program's protocol interactions. These operational specifications have been much investigated, but may miss important requirements. Consider, e.g., a client-server interaction: the client may want to drop the connection if the server does not respond within t seconds, or the server may want to react to a failure by executing certain actions. This kind of requirements fit better in what we call the *declarative* view of the program, which describes the conditions that govern the program's behavior, rather than the behavior itself.

This dissertation investigates the relation between the operational and declarative views of message-passing programs, following a rigorous approach that lies at the intersection of Concurrency Theory, Programming Languages, and Software Verification. Concretely, our work rests upon *process calculi*, small programming languages that rigorously specify interacting programs, and on *behavioral type systems* that can statically verify the conformance of a program specification against a protocol (abstracted as a behavioral type). We focus on *session-based concurrency*, an approach to message-passing concurrency in which the messages exchanged by protocol participants are structured using *session types*. Our approach exploits techniques for assessing the *relative expressiveness* of a process calculus with respect to another one; we also propose new typed process calculi when necessary.

In our work, we relate operational and declarative languages using *encodings* (correct language translations). As operational languages, we consider a π-calculus with binary session types, called π, and several variants of it. As declarative languages, we consider linear concurrent constraint programming (`lcc`), a specification language in which behavior is governed by constraints shared in a centralized store, and ReactiveML (`RML`), an extension of OCaml to the paradigm of *synchronous reactive programming* (SRP).

Our contributions are divided in three parts. In the first part, the focus is on process calculi: we develop two encodings of π into `lcc`. The first encoding relates the notions of *linearity* present in both calculi and allow us to analyze how the point-to-point communication of π can be modeled in `lcc`. The second encoding shows that `lcc` can give a low-level declarative view of *session establishment mechanisms* by using private information and authentication protocols. In the second part, we connect process calculi and actual programming languages: we develop two encodings of π into `RML` to examine the relation between session-based concurrency and timed behavior with event-handling constructs. The first encoding demonstrates that signals (the synchronization unit of SRP) can represent the communication structures in π. The second encoding considers variants of π into RML with asynchronous communication: it showcases how synchronous reactive structures can be used to add time and event-handling constructs to session-based programs. In the final part, we consider the interplay of π and SRP in a slightly different way: we define MRS, a new process calculus with sessions, broadcast communication, event-handling constructs, and timed behavior. We equip MRS with an expressive typing discipline, based on multiparty session types, which allows us to statically check message-passing programs with session protocols, deadlines, and event-handling requirements.

All in all, this dissertation stresses the importance of adopting a *unified view* to the analysis of message-passing programs in which the interplay of declarative features (time, events, and partial information) influences and informs the operational descriptions of behaviors (session protocols). Our approach is comprehensive and rigorous, and allows us to give correct, declarative specifications of message-passing programs. In our view, our technical contributions shed light on the foundations required to ensure the harmonious integration of operational and declarative views that guarantees program correctness and reliability.

Samenvatting

Het analyseren van message-passing programma's blijft een belangrijk vraagstuk binnen de informatica. Een relevant onderzoeksprobleem betreft met name de relatie tussen programma's en communicatieprotocollen waarin zowel de uit te voeren handelingen zijn omschreven alsook de volgorde waarin deze dienen te worden uitgevoerd. Deze programma's kunnen nader worden gespecificeerd door gebruik te maken van een operationeel perspectief, waarin duidelijk aangegeven reeksen van uit te voeren handelingen de protocol-interacties van het programma beschrijven. Deze operationele specificaties zijn reeds veelvuldig onderzocht, maar belangrijke aspecten bleven hierbij onderbelicht. Te denken is bijvoorbeeld aan *client-server* interactie: de *client* wil de verbinding mogelijk verbreken als de *server* niet binnen t seconden reageert, of de *server* wil wellicht op een fout reageren door bepaalde acties uit te voeren. Dit soort vereisten passen beter in wat we het declaratieve perspectief van het programma noemen, waarin de voorwaarden voor het gedrag van het programma worden beschreven in plaats van het gedrag zelf.

Dit proefschrift onderzoekt het verband tussen het operationele en het declaratieve perspectief op *message-passing* programma's. Hierbij wordt gebruik gemaakt van een rigoreuze aanpak die op het snijvlak ligt van *Concurrency Theory*, programmeertalen en softwareverificatie. In concreto berust dit werk op procescalculi: kleine programmeertalen die met elkaar communicerende programma's op rigoreuze wijze specificeren, en op gedragstypesystemen waarmee statisch geverifieerd kan worden of een programmaspecificatie voldoet aan een protocol (als gedragstype geabstraheerd). We richten ons hierbij op *session-based concurrency*: een benadering voor *message-passing concurrency* waarbij de berichten die worden uitgewisseld door protocoldeelnemers gestructureerd worden met behulp van session types. Deze aanpak maakt gebruik van technieken voor het beoordelen van de relatieve expressiviteit van een procescalculus ten opzichte van een andere. Ook nieuwe gedragstypesystemen voor procescalculi worden voorgesteld.

In dit werk worden operationele en declaratieve talen met elkaar in verband gebracht met behulp van coderingen (correct bewezen vertalingen). Als operationele talen beschouwen we een π-calculus met binaire session types, genaamd π, en verschillende varianten hiervan. Als declaratieve talen beschouwen we linear concurrent constraint programming (`lcc`), een specificatietaal waarin gedrag wordt beschreven door constraints die worden gedeeld in een centraal bestand, en ReactiveML (`RML`), een uitbreiding van OCaml met het paradigma van synchronous reactive programming (SRP).

Onze bijdragen zijn verdeeld in drie delen. In het eerste deel ligt de nadruk op procescalculi: hierin worden twee coderingen ontwikkeld van π naar `lcc`. De eerste codering heeft betrekking op de verschillende vormen van lineariteit dat aanwezig is in beide calculi en stelt ons in staat om te analyseren hoe de communicatie van π kan worden gemodelleerd in `lcc`. De tweede codering toont aan dat `lcc` een eenvoudig declaratief perspectief kan bieden op *session establishment mechanisms* door privé-informatie en authenticatieprotocollen te gebruiken. In het tweede deel verbinden we procescalculi en programmeertalen: twee coderingen worden ontwikkeld van π naar RML om de relatie tussen *session-based concurrency* en gedrag in de tijd te onderzoeken met event-handling constructs. De eerste codering toont aan dat signalen (de synchronisatie-eenheid van SRP) de communicatiestructuren in π kunnen representeren. De tweede codering houdt rekening met varianten van π in RML met asynchrone communicatie: het toont aan hoe synchrone reactieve structuren kunnen worden gebruikt om constructies voor tijds- en gebeurtenisafhandeling toe te voegen aan session-based programma's. In het laatste deel beschouwen we het samenspel van π en SRP op een iets andere manier: hierin definiëren we MRS, een nieuwe procescalculus met sessies, uitzendcommunicatie, constructies voor gebeurtenisafhandeling en gedrag in de tijd. We rusten MRS uit met een expressief typesysteem, gebaseerd op multiparty sessions, waarmee we message-passing programma's statisch kunnen verifiëren met sessieprotocollen, deadlines en vereisten voor gebeurtenisafhandeling.

Al met al wordt in dit proefschrift het belang benadrukt van een overkoepelend perspectief op de analyse van *message-passing* programma's waarin het samenspel van declaratieve kenmerken (tijd, gebeurtenissen en partiële informatie) de operationele gedragsbeschrijvingen (sessieprotocollen) beïnvloedt. Onze inclusieve en rigoreuze aanpak maakt het mogelijk om correcte, verklarende specificaties te geven voor message-passing programma's. Naar ons inzicht werpen onze technische bijdragen nieuw licht op de fundamenten die essentieel zijn om een harmonieuze vereniging van operationele en declaratieve inzichten te verzekeren die de correctheid en betrouwbaarheid van programma's garandeert.

To my family and friends,
to whom I owe their staunch
support and without whom
this dissertation would have never
been completed.

Acknowledgments

Four years and three months have passed since the beginning of this adventure. It all started with a simple question: what is the role of logic in Computer Science? I remember thinking of this exact issue while attending a course aptly named "Logic in Computer Science" during my undergraduate studies. Searching an answer to this question led me to discover some of the most fascinating results in Logics, Mathematics, and Computer Science. They sparked my interest in Theoretical Computer Science and to this day, I find myself amazed at the elegance and beauty in them. I cannot talk about my PhD without mentioning the many people that have inspired me and supported throughout these years. In my view, this work has as much of them as it has of me. It is unfortunate that it is almost impossible to mention everyone who has supported me during this process, but I would like to share a few words of gratitude towards some of them.

First of all, I would like to extend my gratitude to my supervisors Prof. Gerard Renardel de Lavalette and Prof. Jorge A. Pérez. It is their guidance and experience that made these results possible. Gerard, I am extremely grateful to you for receiving me in the University of Groningen as one of your students. I never felt out of place, even though I come from a quite far away place. Your experience and advice were invaluable to my development as a researcher. Jorge, you are one of the most talented researchers I have met. I am immensely grateful for all your teachings and supervision during these years. It has been a really enriching experience to be able to conduct research under your guidance. I still remember when you offered me a PhD position in the Netherlands when I was finishing my undergraduate studies back in Colombia. Looking back to that time, I believe that taking you up on that offer has completely changed my perspective on the world and opened me to many experiences that I would never have thought possible. For this I am extremely grateful.

I would also like to thank Prof. Frank de Boer, Prof. Mariangiola Dezani and Prof. Tijs van der Storm for agreeing to assess my dissertation. Prof. Dezani, I am particularly grateful for your detailed comments, which served as a basis to improve the manuscript. I also want to thank Prof. Jos Roerdink for agreeing to chair the defense of this dissertation and to Prof. de Boer, Prof. Dezani, Prof. Lazovik, Prof. Verbrugge, Prof. van der Storm, Dr. Turkmen and Dr. Jongmans for agreeing to be part of the examining committee.

I am extremely grateful to Dr. Cinzia di Giusto and Dr. Ilaria Castellani for receiving me for three months at Inria in Sophia-Antipolis. All our discussions on session types allowed me to gain a much more deeper understanding of the nuances and subtleties one can find in theoretical work. This research internship provided me with invaluable experience in the field of Theoretical Computer Science. Spending time in France also allowed me to experience several enriching discussions with

talented and dedicated researchers, all to whom I am grateful. I would also like to extend my gratitude to Dr. Jaime Arias in Paris. Jaime, I am really grateful for all the support you have given me in the most applied aspects of my research and all the recommendations when dealing with pesky compilers that sometimes seem to not work.

I also want to thank all the members of the Fundamental Computing research group for all their support during this experience. Although most of the current PhD students in the research group are rather new, meeting them is one of the highlights of my experience in the Netherlands. Alen, Joe, and Bas, thank you for our lengthy discussions on topics that go from research to board games and finances. I am extremely grateful to have met you all and I wish you all the best during your academic careers.

During my undergraduate studies I had several mentors who served as an inspiration to start this PhD. I would like to thank, in particular, Prof. Camilo Rueda, Prof. Carlos Olarte, and Prof. Frank Valencia. They opened the doors that let me into this wonderful world and for that I am immensely grateful.

Living in the Netherlands has been a really enriching experience. From learning what are *bitterballen* to understanding the unspoken biking rules, I have enjoyed my experience here thanks to the people who have become part of this amazing experience. To Jasmijn, I am extremely grateful to have met you, binge watching series with you has always made me feel like I never left my home. Our conversations about history and life have always been a highlight of my stay in this country. To Elisa, thank you for our discussions about teaching. I remember all the sushi dinners we had; they helped me to understand that even all-you-can-eat buffets can have a limit. To Ana, thank you for being my friend and showing me that attending the odd party every few months can always be a fun and rewarding experience.

I cannot close this letter without mentioning all my friends back in Colombia. To Carlos, Daniel, Jessica, Juan Pablo, Luis and Nelson, thank you for all the discussions and games we played. Your friendship has been a source of strength during these years. To Alejandro Lopéz, Isabella Lopéz, Camilo Arévalo, Daniela Orozco, Santiago Quintero, Santiago Juri, Felipe Renjifo, Edgar Amézquita, Ghina Garcés, Mónica Hurtado, and Lina Rozo I am grateful to all the experiences and meetings we have had in the past.

Finally, I want to thank my family, without whom nothing of this would have been possible. To my mother, thank you for your unconditional love and support throughout these four years. To my father, thank you for being a source of inspiration and admiration since I started this process. To my brothers and sister, I am extremely grateful to all your support and kindness.

Contents

List of Figures

PART I

INTRODUCTION AND PRELIMINARIES

1 Introduction

1.1 Context and Motivation

Concurrency is the phenomenon exhibited by software systems in which multiple entities, typically called *processes*, interact with each other. These interacting processes can be seen as independent components that execute in parallel to better exploit computational resources and form larger software systems. The study of concurrency and concurrent systems is a long-standing research question in Computer Science, especially as concurrency increases its influence in modern computer systems. Indeed, from banking software to government databases, it is almost impossible to conceive a software system nowadays that does not rely on concurrency in some way.

Concurrency is a wide and multi-faceted phenomenon. It is useful to classify concurrent programs according to the way in which concurrency manifests itself in them. A broad distinction accounts for *shared-memory* and *message-passing* concurrency. We illustrate this distinction intuitively, by means of examples:

Shared-Memory Concurrency: Here we find processes that interact by reading and writing data from a shared medium, such as a shared memory. Multi-threaded programming is a good specific example: we have a set of *threads* (i.e., processes) that execute in parallel within a single device, sharing access to a common pool of *resources* (e.g., RAM, processor time, bandwidth).

Message-Passing Concurrency: Here we find processes that interact by exchanging messages with each other. Thus, concurrency can be assimilated to *communication*. Distributed applications, such as Web services, provide a good specific example: processes may run and reside in different devices, and their overall behavior depends crucially on appropriate coordination patterns between them.

Concurrent programs suffer from specific issues (e.g., *deadlocks*) that are not present in sequential programs. Hence, a common concern for concurrency, be it shared-memory or message-passing, is that of *program correctness*. Broadly speaking, we may say that a (concurrent) program is correct if it does what it is supposed to do. In practice, correctness is associated to some (in)formal properties that specify a set of intended behaviors. Examples of these properties include "every process is allowed to access memory during execution" or "all the message exchanges should respect the prescribed order". Notice that these correctness properties will be different for shared-memory and message-passing concurrency. In the shared-memory case, program correctness largely depends on enabling as much parallelism as possible while disallowing *malicious interferences* between concurrent threads (e.g., ensuring that two threads never modify the same variable at the same time). In the message-passing case, program correctness largely depends on enforcing that parallel components follow some appropriate *communication structure* for exchanged messages (e.g., ensuring that every sent message will be eventually received by the intended receiver).

The focus of this thesis is on message-passing concurrency, and on correctness techniques for message-passing programs. *Formal methods* represent an appealing approach to ensure the correctness of message-passing programs: these are rigorous techniques for specifying, developing, and verifying programs. In this approach, a system is typically modeled using some formal language so as to obtain a precise specification. Such specifications are meant to capture the *essential features* of the system, abstracting away from aspects not directly related to the intended correctness properties. Formal specifications can then be used to establish the correctness of the program using techniques such as *model checking* and *equivalence checking*. Examples of specification languages for concurrent systems include Petri nets [Pet62], the actor model [HBS73], and *process calculi* [CRS18], the main object of study in this work.

Generally speaking, a process calculus is a small programming language with a precise mathematical formulation, which can be used to precisely specify concurrent systems [Fok09]. In this sense, process calculi are for concurrent programs what the λ-calculus is for sequential programs. There are three essential ingredients for defining process calculi:

(1) A *minimal* set of constructs describe the behavior that can be expressed in the calculus, and define the ways in which processes can be composed to describe larger programs. Minimality is beneficial when developing theory, and helps to ensure that specifications are as precise and compact as possible.

(2) An *operational semantics* formalizes a computational interpretation of the interactions between processes.

(3) A *process equivalence* serves to rigorously compare the behavior of processes. A fundamental question is whether two processes exhibit the same behavior—whether they are observationally equivalent (see, e.g., [San09]).

Views of Concurrency In contrast to the canonicity of the λ-calculus in the sequential realm, a myriad of process calculi have been developed, each focused specific aspects of concurrent programs. Given this diversity, we will find it useful to classify process calculi in terms of the *view* they have on concurrent programs. For example, while some calculi focus on describing explicitly the set of steps that the program must execute, others may focus on describing the conditions that trigger such behavior, leaving execution mechanisms implicit. In this dissertation we distinguish two views for process calculi: *operational* and *declarative*. Although there is not a consensus on the precise features that fall within the declarative view [RH04, FMR$^+$09, Har], for our work it is enough to use the following distinction:

Operational View: Here we find process calculi that explicitly describe the execution behavior of a concurrent program. Under an operational view, specifications typically consist of a set of states that can be modified by using control structures, i.e., statements that indicate precise execution steps, such as conditional, looping, and jumping statements. In this view, specifications often exhibit an explicit control flow, i.e., the order in which each statement is executed. Thus, an operational view of programs should allow to reconstruct the program execution just by looking at its code.

We could say that the operational view embodies the idea behind Kowalski's equation: $\mathsf{Algorithm} = \mathsf{Logic} + \mathsf{Control}$ [Kow79]. Process calculi that can be considered as inducing an operational view for message-passing concurrency include the *Calculus of Communicating Systems* (CCS) [Mil80], the π-calculus [MPW92a, MPW92b], and *Communicating Sequential Processes* (CSP) [Hoa85]. Indeed, these calculi explicitly model the interactions between processes using prefixing, recursion, nondeterministic choice, among others.

Declarative View: Process calculi in the declarative view are more concerned with the conditions that govern the program's behavior, rather than with its execution flow. In general, these calculi try to abstract away from notions such as state and control flow; hence, they are often called *stateless* [RH04]. In the declarative view it is not uncommon to specify concurrent programs using statements in some form of logic (e.g., *linear temporal logic* [MP95]). Because languages in the declarative view are more concerned with the conditions that govern the behavior of processes, the execution mechanisms of a program appear implicit.

In a way, we can say that the declarative view is only concerned with half of Kowalski's equation: logic, while control becomes a secondary concern. Examples of declarative languages include Prolog [Kow88], *concurrent constraint programming* (`ccp`) [Sar93], *linear temporal logic* (LTL) [MP95], Reo [Arb16], *Dynamic Response Graphs* [HM10], and *Constraint Handling Rules* [FH93]. They all focus on representing the governing conditions of the system, rather than on explicitly representing interactions between processes.

Note that the distinction between operational and declarative views is not limited to process calculi; rather, it can be extended to other formal languages used for modeling concurrent systems. In this sense, we may also consider a third view:

"Hybrid" View: Intuitively, these languages are hybrid in the sense that they combine both operational and declarative aspects to allow more flexibility in the descriptions of concurrent programs. We use this category to classify some formal languages that are not process calculi, but have formal semantics that allow us to rigorously analyze their behavior. Examples of hybrid languages include OCaml and ReactiveML [MP05].

Using the distinction between operational, declarative, and hybrid views we can recognize which features are more easily represented in each class of languages. Indeed, it can be harder to use operational languages to represent certain features that are more easily expressed in declarative languages and vice versa. For example, LTL (a declarative language), can easily represent the evolution of a program across time, whereas in CCS (an operational language) the notion of time is not explicit, and therefore extensions are needed [MT90]. We will use the terms operational, declarative, and hybrid language to denote languages with operational, declarative, and hybrid views, respectively.

It is worth noticing that realistic message-passing programs are often the product of an amalgamation of features that requires both operational and declarative views to be fully specified and verified. Hence, in this heterogeneous world, a natural question that arises is:

Can we reason about the correctness of message-passing programs from a *unified view* that integrates the best from operational and declarative views?

1.2 Towards a Unifying Perspective of Concurrent Systems

Most realistic concurrent systems cannot be comprehensively described using exclusively an operational or declarative view. Let us consider, for example, the informal requirements of the *Travel Agency* scenario presented in [KCD$^+$09]:

(R1) The customer should pay the selected offer by providing his credit card data within 30 minutes after the reservation step. Otherwise, the reserved offer will be canceled.

(R2) In a 5 minutes interval the customer can only do 3 failed payment trials.

(R3) The customer can cancel a travel reservation the latest 7 days before his travel.

(R4) The customer can change his travel reservation only 2 times. Changes are only allowed between 1 day and 5 days after the reservation date.

(R5) If the booking is done in a special period, a discount is given.

The scenario above is composed of at least two interacting entities: a customer and the travel agency. As these entities most likely run in different devices, we can see this scenario as an instance of a distributed system based on message-passing. We highlight two interesting aspects from this example:

- The requirements are not operational. Rather, since they describe the conditions that affect the execution of the whole system, they fall within the declarative view. For example, Requirement (R2) imposes a condition on the number of failed payment trials within a *time interval*. Similarly, Requirement (R5) indicates that the travel agency must change its behavior depending on information provided by a context external to the system (i.e., special periods for discounts). Since the description only includes the requirements for the travel agency (not for its potential customers), we are left with an incomplete view of the system. This is a common situation: components are often not aware of the requirements and specifications of the other components that will interact with them.

- This partial view of the scenario suggests that involved participants have an *heterogeneous* nature. Indeed, it is likely that both the agency and the customer are implemented using different programming languages, each having different features. For example, while the travel agency can be specified in a declarative language that allows to describe timed behavior, the customer may be well specified in an operational language in which timed behavior is not needed. This heterogeneity is not uncommon in message-passing systems, which are often formed out of the interactions of dozens of heterogeneous components. In this setting, *communication* is the essential glue between these distinct components.

Other examples in which we observe the interplay between the operational and declarative views have appeared in the literature: in [BFM98], the authors specify a multimedia stream with timing requirements. In [CPS09], the authors specify an intrusion detection system that must redirect messages whenever a certain threshold in the number of received messages has been reached. Finally, the Simple Mail Transfer Protocol (SMTP) [Kle08] is another example, in which specific retry strategies are described for whenever the timing requirements of the protocol are not satisfied.

Scenarios such as the one above stress the need for adopting a *unified view* of message-passing programs and systems. Viewing message-passing programs from a unified perspective allows us to obtain comprehensive specifications, which are more robust than the ones obtained by solely viewing them operationally or declaratively. In this way, all the features of the components that form the system can be analyzed independently of the different formal languages used for their specifications.

In our opinion, the first step towards a unified view that homogenizes the components of message-passing programs is to identify a class of formal languages endowed with an adequate abstraction level to capture all the individual features of the program's components. Then, one needs to develop mechanisms to systematically *translate* component specifications into the language identified before. These mechanisms must be carefully defined: we want to guarantee that translated specifications respect the behavior of the source specifications. Once this unified representation has been obtained, we can then proceed to uniformly study the behavior of message-passing programs without many of the issues introduced by their heterogeneous nature. In this unified setting we can also study the interplay of operational and declarative views, because the identified target language must be able to represent both the specific execution steps and the conditions that govern the behavior of the system.

Fig. 1.1 illustrates the concept of a unified view for specification and analysis, using three interacting components (C_1, C_2, and C_3). We use different shapes to represent the different views adopted by the specification language in each case: rectangles denote a declarative language, diamonds denote an operational language, and rect-

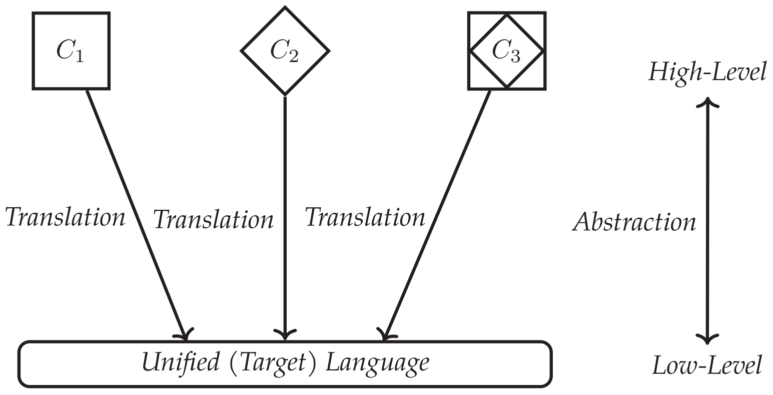


Figure 1.1: A unified view for message-passing programs.

angles with diamonds denote a hybrid language. This way, component C_1, specified in a declarative language, needs to react to certain events, similarly to Requirement (R1). Component C_2 has been specified in an operational language and does not have any additional requirements. Component C_3, specified in a hybrid language, involves timed constraints similar to Requirements (R2) and (R4) above. In the bottom, there is a target language that *unifies* these different views; its incoming arrows represent mechanisms to *translate* the views used to specify C_1, C_2 and C_3 into the chosen target language. These translations compile specifications in a high-level of abstraction into specifications expressed in the lower-level of abstraction of the target language.

A key hypothesis of our work is that declarative languages are powerful enough to provide a foundation for the unified view we advocate. Indeed, in several works declarative languages have been used as target languages for operational languages. Examples include the translations of the asynchronous π-calculus into first-order logic [PSVV06], linear concurrent constraint programing [Hae11], and Flat Guarded Horn Clauses [MM12]; also, the translation of a *session* π-calculus into universal concurrent constraint programming [LOP09]. In contrast, expressing declarative features in operational languages seems much harder; to cope with such requirements, several extensions of process calculi have been developed, see, e.g., [Ama07, KYHH16, CDV15, BYY14]. These extensions often specialize in specific application areas and do not provide the flexibility of a proper declarative language.

Motivated by the previous context and the need for a unified view for analyzing message-passing programs, our work focuses on (1) identifying declarative languages which can be used as foundations for such a view and on (2) investigating the translation mechanisms between languages implementing different views.

1.3 Research Challenges

In this dissertation we shall focus on operational process calculi for message-passing programs and their relations with declarative and hybrid process calculi. We shall use these relations to develop the translation mechanisms required for a unified view of message-passing programs. We are particularly interested in analyzing the interplay of specifications that fall within the operational view and two features that are common in declarative languages: (1) *behavior driven by partial information* and (2) *timed and reactive behavior*:

Partial Information: In message-passing systems, it is often the case that components are influenced by *partial and contextual information*. For example, in the Travel Agency presented in § 1.2 requirements (R2) and (R5) depend on information external to the program like the number of failed payment trials and the current date. While it may be easy to specify the behavior of the program for failed payments or special dates in an operational language, it can sometimes be difficult to express the conditions that trigger these behaviors. To address this shortcoming, previous works have extended operational process calculi with declarative features—see, e.g., [DRV98, BM07a, CD09, BJPV11, BM11]. On the other hand, declarative models of concurrency naturally express partial and contextual information because their logical foundations allow them to deduce new information about the current state of the program—see, e.g., [dBGM00, FRS01, NPV02, OV08b].

Time and Reactivity: As we have seen, the execution of message-passing programs sometimes depends on *timing constraints*. For example, in the Travel Agency presented in § 1.2 requirements (R1) and (R3) specify timing constraints for executing certain actions. If this timing constraint fails, the Travel Agency should be able to *react* accordingly and, perhaps, cancel the order (in the case of (R1)). Once again, while it may be easy to express the communication behavior of the Travel Agency using an operational language, representing the timing constraints can sometimes be unnatural. Considering this shortcoming, previous work have added timed and reactive capabilities to operational languages—see e.g., [Ama07, KYHH16, CDV15, BYY14]. In contrast, some declarative languages are conceived to represent this kind of behavior—see e.g., [CPHP87, BG92, SJG94, NPV02, MP05].

Given this, we refine the challenge stated at the end of § 1.1 into three research questions, given below. We aim at understanding to what extent declarative languages are powerful enough to support a unified view for the analysis of message-passing programs:

(*Q1*) *Can we use declarative languages tailored to describe behavior driven by partial information to analyze message-passing programs specified in operational process calculi?*

(*Q2*) *Can we use declarative languages tailored to describe timed and reactive behavior to analyze message-passing concurrent programs specified in operational process calculi?*

(*Q3*) *Can we use a hybrid process calculus, extended with features to describe timed and reactive behavior, to analyze message-passing programs specified in operational process calculi?*

1.4 Approach: Relative Expressiveness

To address questions (Q1) and (Q2), we shall use *relative expressiveness* techniques to relate two formal languages that fall within different views. Then, to address question (Q3), we shall develop a hybrid process calculus with timed and reactive behavior that allows the analysis of these features within both the operational and declarative views of message-passing programs.

We start by introducing the concept of relative expressiveness. When studying formal languages, such as process calculi, it is natural to ask about their expressive power. In general terms, the expressive power of a rigorous language refers to *what can be expressed* in it [Par08]. There are two main approaches to study the expressive power of formal languages: *absolute expressiveness* and *relative expressiveness* [Par08] (or translational expressiveness [Pet12]).

Absolute expressiveness focuses on proving whether a formal language can be used to solve some kind of computational problem [Par08, Gor10]. Whenever there is a proof that the studied language can solve some specific problem or define some operator, we call it a *positive result*. Proofs of the contrary are considered *negative results*. Hence, absolute results are obtained irrespectively of how the language at hand *relates* to other languages. Two examples of absolute results in the area of process calculi include: De Simone's work on proving that synchronous process calculi SCCS and MEIJE can express all recursively enumerable transition graphs up-to a bisimulation equivalence [dS84] and the work by Baeten et al. which shows that an extension of the *algebra of communicating processes* (ACP) [BK84] can represent every computable transition graph using finite expressions [BBK87].

Relative expressiveness studies translations (or mappings) between formal languages to determine if a *target language* is "as expressive" as some *source language* [Par08, Gor10, Pet12]. At the heart of relative expressiveness lies the notion of *encoding*: a mapping that transforms terms of the source language into terms of the target language and satisfies certain correctness properties. The set of correctness properties that these mappings should satisfy is commonly called *encodability criteria*. There is not a consensus on the specific criteria that should be used to assess the correctness of an encoding. Nonetheless, several proposals have been laid out regarding the desirable properties that should be satisfied by the mapping to ensure it is a correct encoding [Par08, Gor10, Pet12, PvG15]. All these proposals agree on the fact that correct encodings should ensure that translated terms preserve the structure and behavior of their source terms. In this way, together with the mapping itself, encodability criteria relate the expressive power of the source and target languages.

In this dissertation we shall focus on three criteria to ensure encoding correctness: (1) *name invariance*, which ensures that substitutions are preserved by the translation (2) *compositionality*, which guarantees that the translation of a compound process is given in terms of the translations of its sub-processes, and (3) *operational correspondence* which ensures that the translation correctly represents the behavior of source processes. More in detail, operational correspondence is given by two properties: *operational completeness*, which guarantees that the semantics of translated terms matches that of the source language and *operational soundness*, which ensures translated terms can *only* execute what the source language can—for details, see § 2.1. In the literature, name invariance and compositionality are referred to as *static criteria* (i.e., criteria that concern the syntactic structure of the translation), whereas operational correspondence is a *dynamic criterion*, as it is concerned with the semantics of translated processes.

The existence of an encoding that satisfies these encodability criteria is a good indicator that the target language is as expressive as the source language. Hence, we consider the existence of these encodings positive results. On the other hand, negative results (also called *separation results*) are obtained when we can prove that the target language cannot encode (all the constructs of) the source language up to the selected criteria.

Importantly, correct encodings enable the *transference of reasoning techniques*. This means that we would be able to analyze source specifications in the target language [Pér10]. The idea is that the reasoning techniques native to the target language can be used to analyze properties in encoded specifications. For example, an encoding from an untimed process calculus into a timed process calculus can be used to analyze untimed programs in a timed setting. It is because of this characteristic of encodings that we consider them to be the most adequate foundations to our proposed unified view (cf. § 1.2). In particular, encodings can help us cope with the heterogeneity of message-passing programs and analyze the behavior of components in a unified setting, where both operational and declarative views are considered.

1.4.1 Source Languages: Session-Based Concurrency

Most of our work focuses on encoding message-passing programs described using operational process calculi into declarative languages. We concentrate on message-passing programs whose interactions can be structured as sequences of communication actions. This kind of programs are part of what is known as *session-based concurrency*.

Session-based concurrency was first proposed by Honda in the 1990s [Hon93, HVK98] as an approach to specify message-passing programs with the goal of structuring the exchange of messages between interacting components. Session-based concurrency uses *types* as an abstraction of the communication protocols that govern the interactions in a message-passing program; it views communication as *structured sequences* of message exchanges called *sessions*. In each session, two or more *parties* interact by executing sequences of communication actions as indicated by a *session type*. Then, using a *type system*, session types can be verified against an implementation to guarantee that the program ascribes to the desired communication protocol. The main premise of session-based concurrency is that *type soundness* entails *communication correctness* (i.e., the program correctly follows the intended protocol).

A key idea of session-based concurrency is the distinction between *linear* and *unrestricted* behavior. Intuitively, linear behavior corresponds to well-behaved interactions that are *deterministic*, in the sense that they must always be executed as prescribed by the session type. On the other hand, unrestricted behavior corresponds to component interactions that are *nondeterministic*, in the sense that they may occur zero or more times. Session-based programs are considered to execute two phases: (1) a *session establishment* phase where sessions are created and (2) a phase that deals with the communication within a session. The differences between linear and unrestricted behavior become clearer in this context. In the session establishment phase

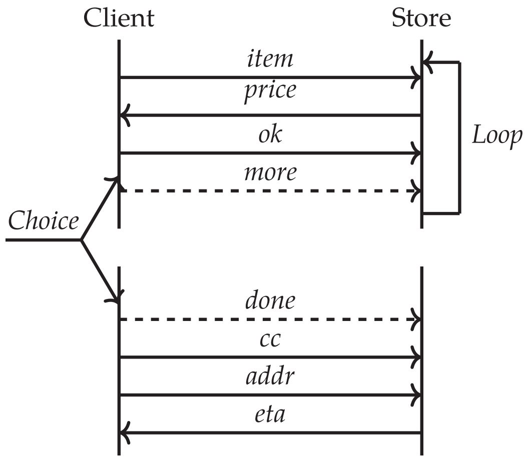


Figure 1.2: Client-Store-Shipper.

there are several services which may be requested by several clients. The system's behavior in this phase is unrestricted since there is not a specified order in which clients may interact with services and clients do not always request a service. This introduces nondeterministic behavior in the interactions between components. Once a session has been established, the system's behavior becomes linear as we would like all the communication within a session to be well-behaved and deterministic.

Session types are often investigated on top of the π-calculus because it provides the essential constructs for specifying message-passing programs. In the π-calculus, *messages* are sent across *channels* between *parallel* processes. A particularly appealing feature of the π-calculus is known as *mobility*: the ability to send channel names along channels themselves. This feature allows the π-calculus to describe concurrent systems in which the network configuration may change during execution. In the context of session-based concurrency session types are assigned to channels names in the π-calculus. Then, the type system verifies that the channels in the process execute the actions prescribed by the assigned session type. Notice that, given their nature, types "evolve" with process reductions.

Another appealing feature of the π-calculus is that it supports both *synchronous* [MPW92a, MPW92b] and *asynchronous* [Bou92] communication. In synchronous communication, output actions are considered to be *blocking*: they can only be executed when there is a corresponding input action. In asynchronous communication, outputs are *non-blocking*: they can be executed at any point during the program run, relying on *buffers* that store messages in transit. Remarkably, in both the synchronous and asynchronous case the syntax of session types remain unchanged; the main differences are introduced in the details of the type system.

We can use the message sequence chart in Fig. 1.2 to intuitively illustrate the kind of communication protocols that can be described with session types. In the figure two parties interact: a Client and a Store. The description of the communication protocol follows:

(1) Client sends the *item* it wants to buy.

(2) Store responds by sending a *price* for the item.

(3) Client responds by stating it agrees to the price of the first item.

(4) At this point, Client can select whether to close the transaction or to keep buying more items:

   (i) If Client wants to keep buying more items then the protocol iterates back to the beginning.

   (ii) If Client wants to stop its purchase then the following checkout protocol begins:

      (a) Client sends his credit card details (*cc*) and its address (*addr*).

      (b) Once Store has confirmed these details, it sends an estimated time for delivery (*eta*), finishing the protocol.

A session type provides a compact way to describe this protocol, and can be used to verify that implementations of the system correctly follow the desired behavior. For example, we could represent Client's behavior as the type:

$$\mu\mathbf{t}.!\mathit{item}.?\mathit{price}.!\mathit{ok}.\oplus\{\mathit{more}\colon \mathbf{t}, \mathit{done}\colon !\mathit{cc}.!\mathit{addr}.?\mathit{eta}.\mathtt{end}\}$$

where '$.$' represents the *sequencing operator*, which ensures that actions are executed in sequence. Construct '$\mu\mathbf{t}$' denotes the *fixpoint* operator for declaring *recursive types*, used to represent infinite protocols. Construct '$!\mathit{item}$' represents the output of an element of type *item* while the construct $?\mathit{price}$ represents the input of an element of type *price*. Also, type $\oplus\{\mathit{more}\colon \ldots, \mathit{done}\colon \ldots\}$ represents a *selection* in which the protocol chooses between two possibly different behaviors. Type end represents the termination of the session. To ensure that communication is correct, we require the type representing the protocol of Store to be *complementary* to that of Client:

$$\mu\mathbf{t}.?\mathit{item}.!\mathit{price}.?\mathit{ok}.\,\&\,\{\mathit{more}\colon \mathbf{t}, \mathit{done}\colon ?\mathit{cc}.?\mathit{addr}.!\mathit{eta}.\mathtt{end}\}$$

In this complementary type all the sending actions are transformed into receive actions, and vice versa. Notice that in the case of the selection $\oplus\{\mathit{more}\colon \ldots, \mathit{done}\colon \ldots\}$, the complementary type *offers* the complementary actions for the alternative behaviors in the type. When two types are complementary, they are said to be *dual*. It is important to notice that duality ensures the communication correctness by guaranteeing the absence of mismatches between the communications action of dual types.

While the original work by Honda et al. focused on *binary communication* (such as in Fig. 1.2) [HVK98], the study of session-based concurrency has been extended in numerous directions. One of the most salient extensions is *multiparty asynchronous session times* [HYC08]. Multiparty types enable the specification of communication protocols with more than two participants from a *global view*, represented as a *global type*. This type represents all the interactions between the components of a program. Global types can be *projected* into *local types* to obtain the individual contributions of each participant to the communication protocol. In a sense, local types are very similar to the binary session types we presented above; the main difference appears because the notion of duality changes. Specifically, multiparty types require local types to be compatible with all the local types of the protocol participants.

Other research directions in session types include *timed session types* [BYY14], and *eventful asynchronous session types* [KYHH16]. Also, there have been extensions of session types for link failures [APN17], self-adaptation [CDV15], robotics [MPYZ19], and exceptions [FLMD19]. The interest in session types has sparked both theoretical and practical advances. Arguably, one of the most important theoretical advancements in session-based concurrency is the discovery of a *Curry-Howard isomorphism* between session types and *linear logic* [CP10, Wad14]. Several implementations of session types have appeared, both as programming language with native session types and libraries to be used in mainstream languages. Examples of languages with native session type support include SePi [FV13] and SILL [PG15]. Similarly, session types have been implemented in mainstram languages such as C [NYH12], Java [KDPG18], Erlang [Fow16], Haskell [PT08, OY16], and OCaml [Pad17].

1.4.2 Target Languages

As representatives of declarative and hybrid languages, we focus on *concurrent constraint programming* (ccp) and *synchronous reactive programming* (SRP). In ccp we find it natural to describe partial and contextual information by using *constraints*, while on SRP, we are able to naturally specify timed and reactive behavior. Notice that synchrony in SRP refers to the fact that programs have periods in which all the components compute until they cannot evolve anymore (see below for details). This differs with the notion of synchronous communication in the π-calculus, which refers to communication where the output is a blocking construct.

(*Linear*) *Concurrent Constraint Programming*

Concurrent constraint programming is a specification language proposed by Saraswat in the 1990s [Sar93]. It is inspired by logic and uses a shared-memory model. The fundamental idea behind ccp is that concurrent systems can be specified in terms of *constraints*. A constraint is a (first-order) logical formula that represents partial information about the state of the shared variables used in the program. The calculus ccp is parameterized with a *constraint system* that specifies the logical formulas relevant to the program and an *entailment relation* that allows to deduce information from the formulas used (e.g., assuming that x is an integer, $x > 5$ entails $x \geq 6$). In ccp there are two forms of interaction: *telling* and *asking*. The telling action indicates that a process adds constraints to a shared medium called a *store*. The store is a constraint itself (i.e., the conjunction of all added constraints) and therefore represents partial information about the system. The asking action indicates that processes can deduce new information from the constraints contained in the store. The information that can be deduced by asking processes is solely given by the entailment relation defined by the constraint system. Asking processes are *suspended* until there is enough information in the store to answer their query up to logical entailment (assumed to be decidable). Because ccp is rooted in (first order) logic, information cannot be removed from the store, hence, the store is a *monotonic* object. To understand this property, let us consider Fig. 1.3.

In it, two ccp processes are exchanging information with the store. The leftmost process is asking if variable y is greater than 5 and the rightmost process is telling

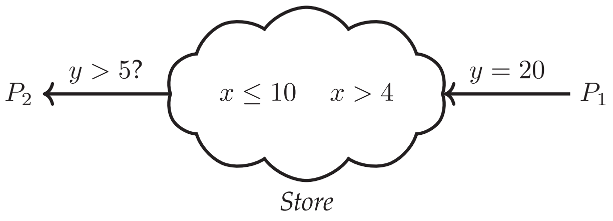


Figure 1.3: An intuitive view of lcc semantics.

that y is equal to 20. We can formally represent this scenario as:

$$\underbrace{\overline{x \leq 10 \wedge x > 4}}_{\text{Store}} \parallel \underbrace{\overline{y = 20}}_{P_1} \parallel \underbrace{y > 5 \rightarrow \overline{x \leq 7}}_{P_2}$$

where '$\parallel$' stands for *parallel composition*, '$\overline{\cdot}$' stands for the *tell* process that adds information to the store, and $y > 5 \rightarrow \overline{x \leq 7}$ is the *ask* process. Intuitively, P_2 represents a process that adds constraint $x \leq 7$ to the store once the constraint $y > 5$ is satisfied. One of the highlights of lcc is that the store is represented as a process (the leftmost one in this case). After the query from P_2 is resolved, the process evolves to:

$$\underbrace{\overline{x \leq 10 \wedge x > 4 \wedge y = 20}}_{\text{Store}} \parallel \overline{x \leq 7}$$

where the monotonicity of the store ensures that constraint $y = 20$ is not removed after synchronization.

As a well-established model for concurrency, ccp has been extended to account for different aspects of concurrent systems such as time [SJG94], nondeterminism and asynchrony [NPV02], mobility [OV08a], and stochastic modeling [BP07]. The most relevant extension of ccp for this work is *linear concurrent constraint programming* (lcc) [SL92, BdBP97, FRS01]. The lcc calculus was designed to have strong ties with *classical linear logic* [Gir87]. As a consequence, the monotonicity of the store is relaxed in lcc, allowing for constraints to be linear resources to be used exactly once.

We can intuitively understand the semantics of lcc by once again considering Fig. 1.3. Using lcc, we can represent the scenario in the figure as:

$$\underbrace{\overline{x \leq 10 \otimes x > 4}}_{\text{Store}} \parallel \underbrace{\overline{y = 20}}_{P_1} \parallel \underbrace{y > 5 \rightarrow \overline{x \leq 7}}_{P_2}$$

The main difference with the ccp process shown before is that the logical conjunction '$\wedge$' has been replaced by the multiplicative conjunction from linear logic (i.e., '$\otimes$'). Due to the non-monotonic nature of the store, once the query from P_2 is resolved, the process evolves to:

$$\underbrace{\overline{x \leq 10 \otimes x > 4}}_{\text{Store}} \parallel \overline{x \leq 7}$$

where constraint $y = 20$ has been consumed.

Synchronous (Reactive) Programming

In synchronous languages [Hal98] programs continuously react to events (*signals*). These events can come from the environment or can be generated by the programs themselves. That is, a program reacts to a set of *input events* by emitting their own *output events*. In synchronous programming, the program reaction to its input events defines a *time instant* (or simply, instant). As their name implies, time instants represent discrete logical time units that start from the moment an input event has been detected up to the moment all the output events have been emitted.

We recognize two flavors of synchronous languages: *dataflow* and *imperative*. In dataflow languages, such as LUSTRE [CPHP87] and SIGNAL [GLGB87], a program reaction consists on the evaluation of a set of equations that define the values of output variables (which represent output events). These equations are evaluated using input events and the values of variables in previous instants. On the other hand, in imperative synchronous languages such as ESTEREL [BG92] a reaction starts from a set of *control points* and finishes when the program reaches a new set of them. Be it dataflow or imperative, all synchronous languages must be *deterministic* (i.e., every sets of inputs must always yield the same results) and must adhere to the *synchronous hypothesis* [Hal98]; i.e., programs are supposed to react rapidly enough to perceive all events in a *suitable order*. This implies: (1) the implicit notion of time in synchronous languages is given by the order in which events appear, and (2) program reactions are considered to be instantaneous. Moreover, the deterministic nature of these languages allows a precise analysis of the event sequences generated by the reactions of the program.

Synchronous reactive programming (SRP) is a programming paradigm for reactive systems proposed by Boussinot and De Simone in 1995 as an answer to some of the shortcomings of ESTEREL [BdS96]. The goal of SRP is to avoid so-called *causality issues* caused by the simultaneous presence and absence of events in ESTEREL programs. This paradoxical situation occurs because of the *presence* and *absence hypotheses*, which allow programs to react instantaneously to both the presence and absence of events. Hence, an ESTEREL program can react to the absence of an event e by emitting e during the same instant, thus making making the event both absent and present at the same time. Boussinot's and De Simone's proposal is to circumvent causality issues by forbidding the presence and absence hypotheses. Namely, synchronous reactive programs events are only considered to be present once they have been emitted and the reaction to event absence is postponed until the next instant.

There are some implementations of synchronous reactive programming: Reactive-C [Bou91], SugarCubes [BS00], FairThreads [Bou06], and ReactiveML [MP05]—see § 2.4. Our work focuses on ReactiveML, as it has received attention in recent years [MP14, MPP15b], and has been found successful in applications such as sensor networks, electrical grids, and interactive modeling, among others [MB05, SMMM06, MM07, BMP13, Ari15, MPP15a]. The main features of ReactiveML are: (1) dynamic process creation, (2) a hybrid approach between functional and synchronous reactive programming, and (3) an intrinsic notion of time, which can be exploited to study time-related properties.

ReactiveML is a programming language built on top of OCaml so that every OCaml program (without objects, labels, and functors) is a valid program in ReactiveML. Furthermore, ReactiveML extends OCaml by adding *processes*, which are *state*

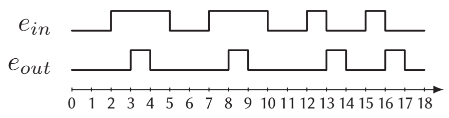


Figure 1.4: Behavior of program *edge*.

machines that can be executed through several time instants. Notice that standard OCaml functions are considered to be *instantaneous* [MP05].

To intuitively understand the semantics of ReactiveML consider an adapted version of the *edge detector* presented in [MP05], called *edge*. The program receives two parameters: events e_{in} and e_{out}. The expected behavior of the program requires that the presence of event e_{in} triggers event e_{out} in the *next* time unit, provided e_{in} was not present in the previous one. This behavior repeats indefinitely, depending on the presence of input event e_{in}. A possible implementation in ReactiveML of *edge* is given below:

```
let process edge $e_{in}$ $e_{out}$ =
  loop
    present $e_{in}$? pause :
    await $e_{in}$ in emit $e_{out}$
```

The input and output events of a ReactiveML program can be described using *timing diagrams* such as the one in Fig. 1.4. In the diagram, events are modeled as continuous lines which can can tick upwards or downwards. Whenever the line corresponding to an event ticks upwards, it means that the event is present; otherwise, it is absent. In particular, Fig. 1.4 describes the behavior of program *edge* and we can observe that whenever e_{in} is present, event e_{out} is emitted by the program exactly once in the next time instant. This shows how the reaction to events can be delayed in ReactiveML. Due to the nature of ReactiveML, we consider it to be a hybrid (programming) language.

1.5 Contributions

Considering our research questions (i.e., (Q1), (Q2) and (Q3)) and the proposed approach, we now describe the three main contributions of our work. They are summarized in Fig. 1.5.

1.5.1 *Sessions and Partial Information*

Our first contribution addresses (Q1) and develops two encodings of session-based concurrency in `lcc`. This work takes inspiration on [LOP09], where the authors present an encoding of the session π-calculus in [HVK98] into declarative processes in *universal* `ccp` (`utcc`) [OV08b], a declarative language based on `ccp`. Although the encoding in [LOP09] already enables us to reason about communication-centric systems from a declarative standpoint, it has two limitations: (a) the role of *linearity* in session-based concurrency is not explicit in the encoding, and (b) declarative encodings of mobility and scope extrusion in `utcc`, based on *abstraction* constructs, are

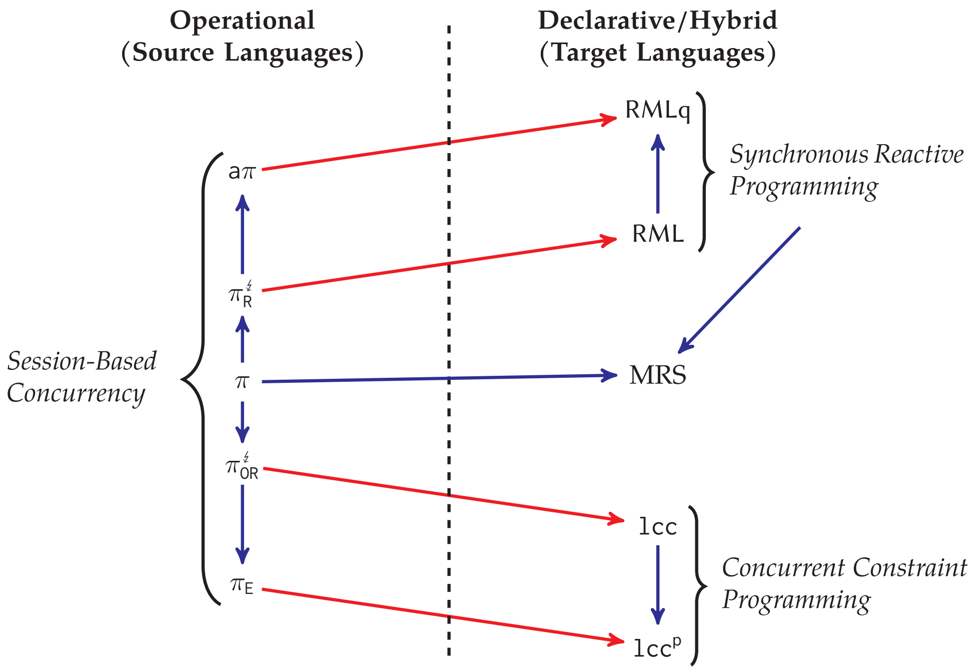

Operational (Source Languages)
Declarative/Hybrid (Target Languages)
RMLq
RML
Synchronous Reactive Programming
aπ
π_R^↯
π
π_OR^↯
π_E
Session-Based Concurrency
MRS
lcc
lcc^p
Concurrent Constraint Programming

Figure 1.5: Summary of results: blue arrows symbolize extensions and red arrows symbolize encodings.

not robust enough to properly match their operational counterparts in the π-calculus. Our work addresses these shortcomings:

- To address (a), we develop an encoding of $\pi_{\mathsf{OR}}^{\lightning}$, a variant of the session π-calculus in [Vas12], into lcc. Using lcc as a target language provides a direct treatment of linearity, as essential in operational approaches to session-based concurrency. Moreover, we prove that this encoding satisfies a subset of Gorla's criteria for *valid encodings* [Gor10]. We also show that this encoding can be used to express temporal session patterns as the ones presented in [NBY17] via examples.

- To address (b) above, we introduce $\mathtt{lcc}^{\mathsf{p}}$, an extension of lcc with *abstractions with local information*. Then, building upon the approach in [HL09], we endow $\mathtt{lcc}^{\mathsf{p}}$ processes with a *type system* that precisely stipulates which variables can be abstracted. Thus, we limit the generality of abstractions, allowing us to faithfully represent hiding and scope extrusion in $\pi_{\mathsf{OR}}^{\lightning}$.

- Finally, we present π_{E}, an extension of $\pi_{\mathsf{OR}}^{\lightning}$ with constructs for *session establishment*. We encode π_{E} into $\mathtt{lcc}^{\mathsf{p}}$: by exploiting declarative specifications, we implement the session establishment phase by embedding the well-known Needham-Schroeder-Lowe authentication protocol [Low96]. This second encoding is shown to satisfy the same correctness properties as the first one, and highlights the benefits of the type system for $\mathtt{lcc}^{\mathsf{p}}$.

1.5.2 *Sessions and Timed, Reactive Behavior*

Our second contribution addresses (Q2) by developing two encodings of session-based concurrency into ReactiveML [MP05]. These encodings enable us to reason about both timed and reactive behavior. Although previous works on session types have developed session π-calculi for events [KYHH16] and time [BYY14, BMVY19], to the best of our knowledge, there are not session-based calculi that allows the specification of both features at the same time. Moreover, our encodings yield *executable programs* which can be used as inputs for the ReactiveML compiler.

- We first present an encoding of $\pi^{\lightning}_{\mathsf{R}}$, a variant of the session-based calculus in [Vas12], into ReactiveML (RML). Using RML as a target language provides direct mechanisms to deal with events and timed behavior, as they are intrinsic to the model [MP05]. Moreover, using the continuation-passing style in [DGS12] allows us to account for the preservation of linearity, similarly to what is done in [Pad17]. We prove our encoding correct, and show that we can use it to model temporal session patterns as the ones introduced in [NBY17].
- Next, we develop an encoding of an asynchronous session-based π-calculus based in [KYHH16], called aπ, into a conservative extension of RML with *queues* and *explicit states*, called RMLq. States and queues aim to explicitly represent the memory that a RML program would have to its disposal. In RML and RMLq, signals can be emitted *asynchronously*; therefore, the second encoding aims at investigating the relation between the asynchronous behavior of signals and messages. This encoding also enjoys strong correctness properties [Gor10].

1.5.3 *Synchronous Reactive Multiparty Sessions*

Our third contribution addresses (Q3) and is a synchronous reactive extension of a multiparty session π-calculus (MPST), called MRS. This calculus is equipped with a multiparty session type system [HYC08, CDPY15]. Being a synchronous reactive extension, MRS naturally accounts for timed and reactive behavior. Its type system ensures *session fidelity* and *communication safety*, as well as two interesting temporal properties:

P1. *Output persistence*: Every *communicating participant* sends a message (exactly) once during every instant.

P2. *Input timeliness*: Every input is matched by an output during the current instant or the next one.

The synchronous reactive type system relies on the usual features of multiparty session types: *global types* that describe a multiparty protocol; *local types* which describe protocol associated to each participant, and a *projection function* relates global and local types.

1.6 Timed Patterns in Communication Protocols

We summarize a number of interesting *timed patterns* for communication protocols that were collected in [NBY17]. The authors identified these patterns in realistic scenarios and verified them using Scribble [YHNN13], a language to describe application-level communication protocols based on multiparty session types. Throughout this thesis we will use these examples as a reference for our developments. Specifically, we shall show that our results can be used to analyze timed protocols in a uniform setting that accounts for the interplay of operational and declarative views.

In recent years there has been an interest in studying the timed patterns that appear on communication protocols for message-passing programs [NBY14, NBY17, BMVY19]. This is particularly important when analyzing application-level communication protocols. Timed patterns are necessary to correctly structure the communication between the components of distributed systems. In realistic scenarios, time requirements can appear in many different ways. For example, as *deadlines* requesting the interaction to occur in a given time-frame or *timed loops* in which actions must be repeated during a certain amount of time. The timed patterns that were identified in [NBY14] follow:

1. *Request-response timeout pattern:* This pattern is used to enforce requirements on the timing of a response, ensuring *quality of service*. The pattern can be required both at server or client side, as illustrated in Fig. 1.6. In [NBY17], the authors

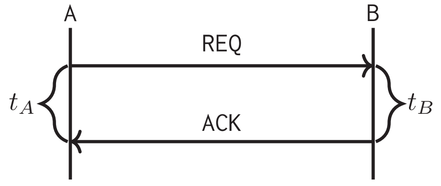


Figure 1.6: Request-response timeout.

identified three use cases for this protocol:

(1) In [CPS09], a service is requested to respond timely: "*an acknowledgment message* `ACK` *should be sent (by the server) no later than one second after receiving the request message* `REQ`".

(2) Similarly, in the same article, a Travel Agency web service specifies the pattern at the client side: "*A user should be given a response* `RES` *within one minute of a given request* `REQ`".

(3) Finally, extracted from the Simple Mail Transport Protocol specification [Kle08], we have a requirement that exhibits a composition of request-response timeout patterns: "*a user should have a five minutes timeout for the* `MAIL` *command and a three minutes timeout for the* `DATABLOCK` *command*".

Requirement (1) above (i.e., the pattern at the server side) specifies that a reply should be sent within a fixed amount of time after the request have been received. In requirement (2), which represents the client side, the server must be ready to receive the client's response within a fixed amount of time. In general, these patterns can be written as:

(a) *Server side:* After receiving a message `REQ` from `A`, `B` must send the acknowledgment `ACK` within t_{A} time units.

(b) *Client side:* After sending a message REQ to B, A must be able to receive the acknowledgment ACK from B within t_B time units.

2. *Messages in a time-frame pattern:* This pattern is used to enforce a limit on the number of messages within a given time-frame (see Fig. 1.7). Regarding this

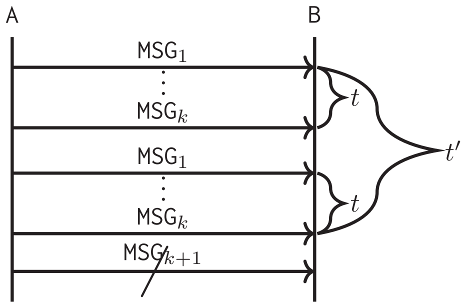


Figure 1.7: Messages in a time-frame.

pattern, the authors of [NBY17] identified two use cases as they appeared in [CPS09]:

(1) Controlling *denial of service* (DOS) attacks: *"a user is allowed to send only three redirect messages to a server with an interval between the messages of no less than two time units"*.

(2) Used in the Travel Agency web service: *"a customer can change the date of his travel only two times and this must happen between one and five days of the initial reservation"*.

Observe that this pattern specifies the repetition of a given number of messages. We can then identify two ways in which this repetition is specified: (a) we can require the repetition to occur at a specified pace; i.e., requiring that messages can only be sent in intervals of time, or (b) specifying an overall time-frame for the messages to be sent.

We generalize these patterns next. Trying to be consistent with Fig. 1.7, we will use t for the interval pattern and t', for the overall time-frame pattern. Letters r and r' denote the upper bound of the time-frames—i.e., $t \leq i \leq r$ (resp. $t' \leq i \leq r'$), where i corresponds to the "safe time" for messages to be sent.

(a) *Interval:* A is allowed to send B at most k messages, and at time intervals of at least t and at most r time units.

(b) *Overall time-frame:* A is allowed to send B at most k messages in the overall time-frame of at least t' and at most r' time units.

3. *Action duration pattern:* This pattern sets a constraint on the delay between actions of the *same* participant (see Fig. 1.8).

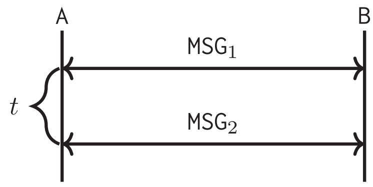


Figure 1.8: Action duration.

In the figure, double-headed arrows are used because, contrary to the request-response timeout, the action duration pattern expresses the requirement only in terms of A. For example, the requirement *"a user should not be inactive more than 30 minutes"* can be specified by this pattern. Neykova et al. [NBY17] state that this pattern can express *progress properties* verified by the UPPAAL model-checker [Uni]: *"a user is allowed to stay in the state for no more than three time units"*. We generalize the pattern below:

(a) The time elapsed between two actions of the same participant A must not exceed t time units.

4. *Repeated constraint pattern:* As its name implies, this pattern deals with repeated constraints that appear as loops in the protocol (see Fig. 1.9).

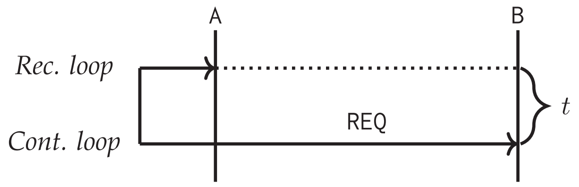


Figure 1.9: Repeated constraint.

According to Neykova et al. [NBY17], this pattern captures the requirements in *pull notification systems*, where the intervals at which a certain interaction should be repeated is fixed. For example: *"the email client should request the emails from the web service every 5 seconds"*. A generalized version of the pattern is:

(a) A must send (and unbounded number of) messages to B every t time units.

The main differences of this pattern with respect to the request-response timeout are: (1) the number of messages is unbounded, and (2) the time required to elapse between messages is *exactly* t.

We shall revisit these patterns in § 4.4, § 7.4, and § 10.6 to investigate how they can be represented in the context of our proposed unified view.

1.7 This Dissertation

We present the overall structure of the dissertation and list some publications and technical reports derived from the work herein presented.

1.7.1 Structure

This dissertation has been divided in four parts, described next. Some of the proofs are presented in the Appendix, which has been organized following the chapter structure in the main text.

- Part I contains chapters that lay down the foundations of our work.
  - Ch. 1 is the current chapter.
  - Ch. 2 introduces our encodability criteria, based on [Gor10]; a session π-calculus, called π [Vas12]; linear concurrent constraint programming (`lcc`) [Hae11]; ReactiveML [MP05]; and a multiparty session calculus, called MPST [CDPY15].
  - Ch. 3 introduces the variants and extensions of the languages in Part I that will serve as source and target languages for the translations presented further in the dissertation. These languages have been developed specifically to address the challenges in further chapters.
- Part II answers (Q1) and presents two translations from session calculi into linear concurrent constraint languages and their respective correctness proofs (cf. § 1.5.1).
  - Ch. 4 presents an encoding from a variant of π without *output races* ($\pi^{\lightning}_{\mathsf{OR}}$) into `lcc`. We also provide examples of timed communication patterns represented in our translation.
  - Ch. 5 presents an encoding from an extension of $\pi^{\lightning}_{\mathsf{OR}}$ with explicit session establishments and locations, called π_{E}, into an `lcc` extension, called $\mathtt{lcc}^{\mathsf{p}}$, that allows abstractions to have *private information*.
  - Ch. 6 gives some final remarks and mention some additional related work.
- Part III answers (Q2) and introduces two translations from session-based calculi into ReactiveML (cf. § 1.5.2).
  - Ch. 7 presents a translation from a variant of π without output and input races ($\pi^{\lightning}_{\mathsf{R}}$) into ReactiveML. We show the translation correctness and present some examples of timed communication patterns in our translation.
  - Ch. 8 presents a translation from an asynchronous session-based π-calculus called $\mathtt{a}\pi$, a fragment of the calculus presented in [KYHH16], into a queue-based variant of RML, called RMLq. We present the correctness properties and give some small examples.
  - Ch. 9 gives final remarks and mention some additional related work.

- Part IV answers (Q3) and presents a multiparty session-based calculus with constructs for synchronous reactive programming, called MRS, and its type system (see Ch. 10). In Ch. 11 we give some final remarks and related work (cf. § 1.5.3).
- Part V concludes this dissertation by giving some closing remarks and future work in Ch. 12.

1.7.2 *Origin of the Results*

The results in this dissertation supersede peer-reviewed publications and technical reports co-authored by the author.

- Part II supersedes:
  - Mauricio Cano, Camilo Rueda, Hugo A. López, and Jorge A. Pérez. Declarative interpretations of session-based concurrency. In *Proc. of the Int. Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 67–78. ACM, 2015.
- Part III supersedes:
  - Mauricio Cano, Jaime Arias, and Jorge A. Pérez. Session-based concurrency, reactively. In *Proc. of the Int. Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 74–91, 2017.
  - Mauricio Cano, Jaime Arias, and Jorge A. Pérez. A reactive interpretation session-based concurrency. Workshop on Reactive and Event-based Languages & Systems (REBLS), co-located with the ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH), 2016.
  - Jaime Arias, Mauricio Cano, and Jorge A. Pérez. Towards A practical model of reactive communication-centric software. In *Proc. of the Italian Conference on Theoretical Computer Science (ICTCS).*, volume 1720 of *CEUR Workshop Proceedings*, pages 227–233. CEUR-WS.org, 2016.
- Part IV supersedes:
  - Mauricio Cano, Ilaria Castellani, Cinzia Di Giusto, and Jorge A. Pérez. Multiparty Reactive Sessions. Research Report 9270, INRIA, April 2019.

2 Preliminaries

In this chapter we present the technical background needed in our work. First, we formally define the correctness properties that we shall require in our encodings (cf. § 2.1). Next, we introduce a session π-calculus, called π [Vas12], used as a representative language for session-based concurrency (cf. § 2.2). This process calculus is the foundation for $\pi^{\lightning}_{\mathsf{OR}}$ (cf. § 3.1.1) and $\pi^{\lightning}_{\mathsf{R}}$ (cf. § 3.1.2); i.e., the source languages for two of our translations (presented in Ch. 4 and Ch. 7, respectively). Similarly, we introduce two of our target languages: `lcc` [Hae11] in § 2.3 and ReactiveML [MP05] in § 2.4. These languages are used for the encodings in Ch. 4 and Ch. 7, respectively. Finally, we introduce multiparty sessions (MPST) [CDPY15] in § 2.5. This calculus is used as the basis for the results presented in Ch. 10.

2.1 Relative Expressiveness

In this section we define the correctness properties we desire for our translations. A key reference for this section comes from Gorla's criteria for *valid encodings* [Gor10].

In § 2.1.1 we formally introduce languages and translations. In § 2.1.2, we introduce the correctness criteria used for our correct translations. Finally, in § 2.1.3, we introduce a refined formulation of operational correspondence, needed to bridge the differences between declarative and operational languages.

2.1.1 Languages and Translations

We first formally define *languages* and *translations*.

Definition 2.1 (Languages and Translations). We define:

- A *language* $\mathcal{L}$ is a triplet $\langle \mathsf{P}, \longrightarrow, \lessapprox \rangle$, where P is a set of terms (i.e., expressions, processes), $\longrightarrow$ is a relation on P that denotes an operational semantics, and $\lessapprox$

is a pre-order on P. We use $\Longrightarrow$ to denote the reflexive-transitive closure of $\longrightarrow$.

- A *translation* from $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \lesssim_s \rangle$ into $\mathcal{L}_t = \langle \mathsf{P}_t, \longrightarrow_t, \lesssim_t \rangle$ (each with countably infinite sets of variables V_s and V_t, respectively) is a pair $\langle [\![\cdot]\!], \psi_{[\![\cdot]\!]} \rangle$, where $[\![\cdot]\!] : \mathsf{P}_s \rightarrow \mathsf{P}_t$ is defined as a mapping from source to target terms, and $\psi_{[\![\cdot]\!]} : \mathsf{V}_s \rightarrow \mathsf{V}_t$ is a *renaming policy* for $[\![\cdot]\!]$, which maps source variables to target variables.

Some informal intuitions follow. In language triplets, P represents a set containing all the terms of the language, and finitely represented as a formal grammar, which gives the formation rules for terms. Next, the operational semantics are given as relations between terms, finitely denoted by sets of rules. In an operational semantics, $P \longrightarrow P'$ represents a pair (P, P') that is included in the relation; we call each one of these pairs a *step*. Each step represents the fact that term P *reduces* to term P'. For the rest of this section, we will refer to $P \longrightarrow P'$ as a reduction step. Moreover, we use $P \Longrightarrow P'$ to say that P reduces to P' in 0 or more steps (i.e., a "multi-step" reduction). Finally, and differing from [Gor10], we have $\lesssim$, which denotes a pre-order between terms used to compare processes inside the language. We use a pre-order, rather than the equivalence relation used in [Gor10], to account for the more general setting in this dissertation. Indeed, in our work we will instantiate $\lesssim$ as a symmetric pre-order denoting a structural congruence (cf. Def. 2.11), as a bisimilarity (cf. Def. 2.33), and as a non-symmetric pre-order (cf. Def. 2.42).

Regarding the pairs that represent translations, we have that the mapping $[\![\cdot]\!]$ assigns each source term a corresponding target term. Notice that this mapping is usually defined inductively over the structure of source terms. The renaming policy $\psi_{[\![\cdot]\!]}$ is in charge of translating variables. Notice that we do not use the more general version of a renaming policy, given in [Gor10], since it is not necessary—in all our translations, variables are translated to themselves.

When referring to translations, we often use $[\![\cdot]\!]$ (i.e., the mapping), instead of the pair notation. We now introduce some terminology regarding translations.

Notation 2.2. Let $\langle [\![\cdot]\!], \psi_{[\![\cdot]\!]} \rangle$ be a translation from $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \lesssim_s \rangle$ into $\mathcal{L}_t = \langle \mathsf{P}_t, \longrightarrow_t , \lesssim_t \rangle$.

- We will refer to $\mathcal{L}_s$ and $\mathcal{L}_t$ as *source* and *target* languages of the translation, respectively. Whenever it does not creates any confusion, we will only refer to source and target languages as *source* and *target*.
- We say that any process $S \in \mathsf{P}_s$ is a *source term*.
- Similarly, given a source term S, we say that any process $T \in \mathsf{P}_t$ that is reachable from $[\![S]\!]$ using $\longrightarrow_t$ is a *target term*.

2.1.2 Correctness Criteria

Since we are interested in meaningful translations, we define a set of properties which determine whether said translation is *valid* or not. Following [Gor10], we shall be interested in four properties: *name invariance*, *compositionality*, *operational completeness*, and *operational soundness*. We refer to these properties as *correctness criteria*.

Definition 2.3 (Valid Encoding). Let $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \lessapprox_s \rangle$ and $\mathcal{L}_t = \langle \mathsf{P}_t, \longrightarrow_t, \lessapprox_t \rangle$ be languages. Also, let $\langle [\![\cdot]\!], \psi_{[\![\cdot]\!]} \rangle$ be a translation between them (cf. Def. 2.1). We say that such a translation is a *valid encoding* if it satisfies the following criteria:

1. **Name invariance**: For all $S \in \mathsf{P}_s$ and substitution σ, there exists σ' such that $[\![S\sigma]\!] = [\![S]\!]\sigma'$, with $\psi_{[\![\cdot]\!]}(\sigma(x)) = \sigma'(\psi_{[\![\cdot]\!]}(x))$, for any $x \in \mathsf{V}_s$.

2. **Compositionality**: Let $\mathsf{res}_s(\cdot,\cdot)$ and $\mathsf{par}_s(\cdot,\cdot)$ (resp. $\mathsf{res}_t(\cdot,\cdot)$ and $\mathsf{par}_t(\cdot,\cdot)$) denote restriction and parallel composition operators in P_s (resp. P_t). Then, we define: $[\![\mathsf{res}_s(x, P)]\!] = \mathsf{res}_t(\psi_{[\![\cdot]\!]}(x), [\![P]\!])$ and $[\![\mathsf{par}_s(P, Q)]\!] = \mathsf{par}_t([\![P]\!], [\![Q]\!])$.

3. **Operational Completeness:** For every $S, S' \in \mathsf{P}_s$ such that $S \Longrightarrow_s S'$, it holds that $[\![S]\!] \Longrightarrow_t T$ and $T \lessapprox_t [\![S']\!]$, for some $T \in \mathsf{P}_t$.

4. **Operational Soundness:** For every $S \in \mathsf{P}_s$ and $T \in \mathsf{P}_t$ such that $[\![S]\!] \Longrightarrow_t T$, there exists S', T' such that $S \Longrightarrow_s S'$ and $T \Longrightarrow_t T'$ and $T' \lessapprox_t [\![S']\!]$.

Above, name invariance and compositionality correspond to the so-called *static criteria*. Name invariance ensures that substitutions are well-behaved in translated terms. The condition $\psi_{[\![\cdot]\!]}(\sigma(x)) = \sigma'(\psi_{[\![\cdot]\!]}(x))$ ensures that for every variable substituted in the source term (i.e., $\sigma(x)$), there exists a substitution σ' such that the translation of x (i.e., $\psi_{[\![\cdot]\!]}(x)$) is substituted by the translation of $\sigma(x)$.

Compositionality ensures that the translation of compound terms depends on their sub-terms. These sub-terms should then be combined in a target language context that ensures their interactions are preserved. In this work, rather than the more general version presented in [Gor10], we focus on compositionality at the level of parallel composition and restriction. While the focus on parallel composition is expected (we are dealing with languages that describe concurrent programs), our focus on restriction comes from the fact that, as we shall see later, the hiding operator in the π-calculus does not entirely correspond to the restriction operator in our target languages.

Together, operational completeness and soundness form the *operational correspondence* criterion, which can be considered a *dynamic correctness criterion*, as it deals with preservation and reflection of behavior. Intuitively, operational completeness ensures that the behavior of the source semantics is preserved by the target semantics. Operational soundness, on the other hand, ensures that the target semantics *does not* introduce extraneous steps that do not correspond to anything the source can do. More in details, operational completeness requires that for every multi-step reduction in the source language there exists a multi-step reduction in the target language that simulates the source. The pre-order $\lessapprox_t$ then ensures that the target term obtained by the multi-step reduction in the target terms preserves the same behavior as the translation of the reduced source term. Similarly, operational soundness requires that every reduction in the target language corresponds to a reduction in the source, where $\lessapprox_t$ ensures that the reduced target term has the same behavior as the reduced source term.

Remark 2.4 (*Gorla's Criteria*). Besides name invariance, compositionality, and operational correspondence, Gorla [Gor10] advocates for two additional correctness criteria (*divergence reflection* and *success sensitiveness*), which we do not consider in this work.

Success Sensitiveness assumes a "success" predicate on source processes (denoted $S \Downarrow$) that is also definable on target processes (denoted $T \Downarrow$). In the name-passing calculi considered in [Gor10], this predicate can be naturally assimilated to the notion of *observable* (or *barb*). The corresponding encodability criterion is then defined as

"A translation $[\![\cdot]\!] : \mathcal{L}_s \to \mathcal{L}_t$ is success sensitive if, for every S, it holds that $S\Downarrow$ if and only if $[\![S]\!]\Downarrow$."

We do not consider success sensitiveness because the session π-calculi we will present in this work do not have a natural notion of observable behavior: as we will see, the semantics of these languages are closed under restriction—there are no free names we can observe (let alone preserve by a translation).

Divergence Reflection ensures that every infinite sequence of reductions in a target term corresponds to some infinite sequence of reductions in its associated source term. Let us write $S \longrightarrow_s^\omega$ (resp. $T \longrightarrow_t^\omega$) whenever the source term S (resp. target term T) has such an infinite sequence of reductions. The corresponding encodability criterion is then defined as:

"A translation $[\![\cdot]\!] : \mathcal{L}_s \to \mathcal{L}_t$ reflects divergence if, for every S such that $[\![S]\!] \longrightarrow_t^\omega$ then $S \longrightarrow_s^\omega$."

In our setting, this criterion is not very meaningful because the semantics of target languages can be very different to the ones of source languages—see below. Hence, there may not be a clear correspondence between infinite behavior between the source and target languages. Moreover, in most of the session π-calculi we use as source languages, infinite behavior will only come from (input-guarded) replicated processes ("servers"), which are triggered by a corresponding output prefix ("client requests"). As we will see later, the type systems introduced for these source languages will ensure that well-typed processes contain a finite number of server requests, which rules out the possibility of having infinite reduction sequences in source processes.

2.1.3 A Refined Notion of Operational Correspondence

Arguably, the most interesting correctness criterion of the ones mentioned above is operational correspondence: it attests that a translation preserves the intended behavior of a source term, without adding or removing any meaningful behaviors found in the source language. While this intuition is clear, formalizing operational correspondence may be delicate: depending on the semantics of both source and target languages, we may be interested in variants of the operational criteria stated above.

To illustrate this point, we consider an extended example based on μCCS: this is the (tiny) fragment of CCS [Mil80] without prefixes, restriction, or relabeling, given by the grammar below. We assume sets $\mathcal{A}$ and $\overline{\mathcal{A}}$, ranged over $a, a', \ldots$ and $\overline{a}, \overline{a}', \ldots$, respectively. These sets contain the actions that can synchronize.

$$s ::= \checkmark \mid a \mid \overline{a}$$
$$P ::= s \mid P_1 \parallel P_2$$

Above, s denotes actions $a, \overline{a}$, or the terminated action $\checkmark$. Then, P denotes the parallel composition of actions (i.e., processes). Let us now consider the following semantics for μCCS:

$$\lfloor \textsc{Sync} \rfloor\ \frac{s_1 = a \quad s_2 = \overline{a}}{s_1 \parallel s_2 \longrightarrow \checkmark} \qquad \lfloor \textsc{Par} \rfloor\ \frac{P_1 \longrightarrow P_1'}{P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2} \qquad \lfloor \textsc{Str} \rfloor\ \frac{P_1 \doteq P_1' \longrightarrow P_2' \doteq P_2}{P_1 \longrightarrow P_2}$$

where Rule $\lfloor$Sync$\rfloor$ synchronizes complementary actions a and $\overline{a}$; this is acknowledged by evolving into $\checkmark$. Rule $\lfloor$Par$\rfloor$ allows reduction of parallel components. Finally, Rule (Str) allows to use $\doteq$, a syntactic equivalence extended with commutativity and associativity of the parallel composition operator, as the structural congruence in μCCS. The formal language is then given by $\mathcal{L}_\mu = \langle \mu\text{CCS}, \longrightarrow, \doteq \rangle$.

To compare semantics with different speeds, we define an alternative synchronous (in the sense of synchronous programming) semantics for μCCS. The goal of this synchronous semantics is to execute *all* possible synchronizations in a process instantaneously—i.e., in a single step. To define such semantics, we take into account two important observations:

- In μCCS, a *redex* is the parallel composition of two complementary actions: $a \parallel \overline{a}$. We use $R_1, R_2, \ldots$ to range over redexes.
- In μCCS, a process can be represented as the finite parallel composition of redexes and actions which, up to the structural congruence $\doteq$, can be written as: $R_1 \parallel \cdots \parallel R_n \parallel s_1 \parallel \cdots \parallel s_m$, with $n, m \geq 1$.

Using the previous observations, we now present a synchronous semantics for μCCS, given by the following reduction rule:

$$\lfloor \textsc{SSync} \rfloor\ \frac{\forall i, j \in \{1, \ldots, m\}.(i \neq j \wedge s_i \parallel s_j \text{ not a redex})}{R_1 \parallel \cdots \parallel R_n \parallel s_1 \parallel \cdots \parallel s_m \Downarrow \checkmark_1 \parallel \cdots \parallel \checkmark_n \parallel s_1 \parallel \cdots \parallel s_m}$$

In the semantics above, we assume that all the processes have already been pre-organized by applying $\doteq$. This is done for simplicity, as we would require to add a rule similar to (Str), otherwise. Intuitively, Rule $\lfloor$SSync$\rfloor$ synchronizes all the redexes R_i contained in a process, while leaving the actions that cannot form redexes (i.e., s_i) intact. The semantics above induces a formal language: $\mathcal{L}_\mu^\# = \langle \mu\text{CCS}, \Downarrow, \doteq \rangle$. Notice that even in this simple setting, establishing operational completeness for a concrete translation is not obvious.

Example 2.5. Let us consider the identity translation of $\mathcal{L}_\mu$ into $\mathcal{L}_\mu^\#$, i.e., $[\![P]\!] = P$. Then, consider process $P = a \parallel \overline{a} \parallel a' \parallel \overline{a}'$. With the semantics for $\mathcal{L}_\mu$, it can be shown that a possible reduction is $P \longrightarrow (\checkmark \parallel a' \parallel \overline{a}') = P'$. In contrast, using the semantics for $\mathcal{L}_\mu^\#$, the only possible reduction for the translation is $[\![P]\!] \Downarrow (\checkmark \parallel \checkmark)$. Therefore, it can be observed that criterion (3) in Def. 2.3 does not hold: using the semantics of $\mathcal{L}_\mu^\#$, there does not exist a T such that $[\![P]\!] \Downarrow T$, where $T \doteq [\![P']\!]$. △

The example above shows that, in a single step, the semantics of $\mathcal{L}^{\#}_{\mu}$ allows more synchronizations than the semantics for $\mathcal{L}_{\mu}$. Still, it can be shown that the translation is semantically correct: although $\mathcal{L}^{\#}_{\mu}$ cannot match exactly the reductions in $\mathcal{L}_{\mu}$, it can be shown that for every $\longrightarrow$ step there is a corresponding $\Downarrow$. This correspondence, however, is not so obvious: it can be shown that for every step $P \longrightarrow P'$, there exists a reduction $P \Downarrow P''$ such that $P' \longrightarrow P''$. In other words, it can be shown that for every source reduction, there exists a target reduction which corresponds to a sequence of source reductions starting from the reduced source term. For example, in Ex. 2.5, we have that $P \longrightarrow \checkmark \parallel a' \parallel \overline{a}' = P'$, that $[\![P]\!] \Downarrow \checkmark \parallel \checkmark$, but also, $P' \longrightarrow \checkmark \parallel \checkmark$.

To formalize the previous idea, let us consider in detail the formulation of completeness in Gorla's framework (cf. Def. 2.3(3)):

"For every $S, S' \in \mathsf{P}_s$ such that $S \Longrightarrow_s S'$, it holds that $[\![S]\!] \Longrightarrow_t T$ and $T \gtrsim_t [\![S']\!]$, for some $T \in \mathsf{P}_t$."

In cases such as the translation of $\mathcal{L}_{\mu}$ into $\mathcal{L}^{\#}_{\mu}$, this property does not appropriately capture the connection between source terms and their translated target terms. We should then relax this formulation in such a way that it allows target terms to be "ahead" with respect to the behavior in their corresponding source terms. A refined formulation for completeness is the following:

"For every $S, S' \in \mathsf{P}_s$ such that $S \Longrightarrow_s S'$, it holds that $[\![S]\!] \Longrightarrow_t T$, $S' \Longrightarrow_s S''$ and $T \gtrsim_t [\![S'']\!]$, for some $T \in \mathsf{P}_t$ and some $S'' \in \mathsf{P}_s$."

The difference is in the second line: since the target language "moves faster" than the source language, the process T that is obtained from $[\![S]\!]$ is not related to S', which has been obtained using the (slower) source semantics; to compensate for this, additional steps from S' are required, until reaching some S''. Then, we consider the correspondence between $[\![S'']\!]$ and T, rather than between $[\![S']\!]$ and T.

In our results, this difference becomes apparent in Ch. 7, as ReactiveML has a synchronous semantics. Therefore, the semantics of ReactiveML indeed "goes faster" than that of π: in the semantics of ReactiveML each step represents a complete instant and therefore, several actions will be executed in a single execution step. These different speeds create a mismatch with respect to the (standard) reduction semantics of π. For these reasons, we establish the correctness of our translations with respect to the following definition of *refined encoding*, which adopts our revised formulation of completeness:

Definition 2.6 (Refined Encoding). Let $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \gtrsim_s \rangle$ and $\mathcal{L}_t = \langle \mathsf{P}_t, \longrightarrow_t, \gtrsim_t \rangle$ be languages; also let $\langle [\![\cdot]\!], \psi_{[\![\cdot]\!]} \rangle$ be a translation between them (cf. Def. 2.1). We say that such a translation is a *refined encoding* if it satisfies name invariance (cf. Def. 2.3(1)), compositionality (cf. Def. 2.3(2)), operational soundness (cf. Def. 2.3(4)), and the following criterion for operational completeness:

(3′) **Operational Completeness:** For every $S, S' \in \mathsf{P}_s$ such that $S \Longrightarrow_s S'$, it holds that $[\![S]\!] \Longrightarrow_t T$, $S' \Longrightarrow_s S''$ and $T \gtrsim_t [\![S'']\!]$, for some $T \in \mathsf{P}_t$ and some $S'' \in \mathsf{P}_s$.

Throughout the rest of this work we shall focus on both valid and refined encodings as needed. In this sense, we will find it illustrative to consider the connection between them. To this end, we establish results of *semantic correspondence* that connect two different semantics for the same process language.

Semantic Correspondence

Notice that the differences between valid encodings and refined encodings are induced by, what we have dubbed, the "semantic speed" of both the source and target languages. In our message-passing setting, semantic speed refers to the number of parallel synchronizations that a term can execute in a single step. For example, consider the following π-calculus process:

$$P = (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(x\langle v_1\rangle.\mathbf{0} \mid w\langle v_2\rangle.\mathbf{0} \mid y(z_1).\mathbf{0} \mid z(z_2).\mathbf{0})$$

As we shall see later (cf. § 2.2), the semantics of π (cf. Fig. 2.1) will only let P execute a single synchronization in each reduction. This means that P will reduce to either $(\boldsymbol{\nu} wz)(w\langle v_2\rangle.\mathbf{0} \mid z(z_2).\mathbf{0})$ or $(\boldsymbol{\nu} xy)(x\langle v_1\rangle.\mathbf{0} \mid y(z_1).\mathbf{0})$. This behavior will contrast with the synchronizations in ReactiveML as we will see later (cf. § 2.4), since the following ReactiveML program:

$$e_4 = \texttt{signal}\ x, x'\ \texttt{in}\ (\texttt{emit}\ x\ 42 \parallel \texttt{await}\ x(y)\ \texttt{in}\ y \parallel \texttt{emit}\ x'\ 56 \parallel \texttt{await}\ x'(z)\ \texttt{in}\ z)$$

will reduce in a single step into $42 \parallel 56$. This means that the semantics of ReactiveML allows e_4 to make more than one parallel synchronization in a single step. Thus, according to our intuitive definition of semantic speed, ReactiveML is faster than π.

Remark 2.7. Notice that the operational completeness from valid encodings (Def. 2.3) implies the refined operational completeness presented in Def. 2.6. This means that our operational correspondence relaxes Gorla's operational correspondence by allowing the source language to be "behind" the target language during a sequence of reductions.

Considering the previous remark, it is natural to question whether given a refined encoding (cf. Def. 2.6), there exists an alternative semantics for the source language which makes the encoding valid (cf. Def. 2.3). In our setting, as we shall show in this dissertation, this query can be answered positively. Nonetheless, the alternative semantics should have a strong connection with the initial semantics, as we do not want to add or remove behavior in the source language. We call this relation a *semantic correspondence* between the two semantics. Notice that the idea of semantic speed is orthogonal to semantic correspondence, as the correspondence just ensures that the semantics allows the same behavior in a term. Intuitively, two semantics are *semantically corresponding* if whenever one of them goes ahead of the other, it is possible to the semantics that is behind to "catch up" and reach the same term. Below we give both the definition of semantic correspondence in a single-step and a multi-step version.

Definition 2.8 (Semantic Correspondence (Single-Step)). Given a language $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \lesssim_s\rangle$ and an alternative semantics for $\mathcal{L}_s$, denoted $\longrightarrow_{s'}$. We say that $\longrightarrow_s$ and $\longrightarrow_{s'}$ are semantically correspondent if for every $S_1, S_2 \in \mathsf{P}_s$ the following holds:

1. If $S_1 \longrightarrow_s S_2$ then there exists $S_2' \in \mathsf{P}_s$ such that $S_1 \longrightarrow_{s'} S_2'$ and $S_2 \Longrightarrow_s S_2'$.
2. If $S_1 \Longrightarrow_{s'} S_2$ then $S_1 \Longrightarrow_s S_2$.

$$\lfloor\textsc{Com}\rfloor\ \ (\boldsymbol{\nu}xy)(x\langle v\rangle.P \mid y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q\{v/z\} \mid R)$$

$$\lfloor\textsc{Sel}\rfloor\ \ (\boldsymbol{\nu}xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i{:}Q_i\}_{i\in I} \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q_j \mid R)\ (j \in I)$$

$$\lfloor\textsc{Rep}\rfloor\ \ (\boldsymbol{\nu}xy)(x\langle v\rangle.P \mid {*}\,y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q\{v/z\} \mid {*}\,y(z).Q \mid R)$$

$$\lfloor\textsc{IfT}\rfloor\ \ \texttt{tt}?\,(P):(Q) \longrightarrow P \qquad \lfloor\textsc{IfF}\rfloor\ \ \texttt{ff}?\,(P):(Q) \longrightarrow Q$$

$$\lfloor\textsc{Str}\rfloor\ \frac{P \equiv_{\mathsf{S}} P' \quad P' \longrightarrow Q' \quad Q' \equiv_{\mathsf{S}} Q}{P \longrightarrow Q} \qquad \lfloor\textsc{Par}\rfloor\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \lfloor\textsc{Res}\rfloor\ \frac{P \longrightarrow P'}{(\boldsymbol{\nu}x)P \longrightarrow (\boldsymbol{\nu}x)P'}$$

Figure 2.1: Reduction relation for π processes.

Definition 2.9 (Semantic Correspondence (Multi-Step)). Given a language $\mathcal{L}_s = \langle \mathsf{P}_s, \longrightarrow_s, \lessapprox_s \rangle$ and an alternative semantics for $\mathcal{L}_s$, denoted $\longrightarrow_{s'}$. We say that $\longrightarrow_s$ and $\longrightarrow_{s'}$ are semantically correspondent if for every $S_1, S_2 \in \mathsf{P}_s$ the following holds:

1. If $S_1 \Longrightarrow_s S_2$ then there exists $S_2' \in \mathsf{P}_s$ such that $S_1 \Longrightarrow_{s'} S_2'$ and $S_2 \Longrightarrow_s S_2'$.
2. If $S_1 \Longrightarrow_{s'} S_2$ then $S_1 \Longrightarrow_s S_2$.

2.2 The Session π-Calculus (π)

In this section we present a session π-calculus, simply called π. As a variant of the π-calculus, π is message-passing process calculus, used to model communicating systems. In π, communication is synchronous: every output action is blocked until a complementary input action can receive the sent message. In π, *binary sessions* are established over *communication channels*. Each communication channel in π is characterized by a pair of *channel endpoints*. Then, a communicating party can send messages over its corresponding channel endpoint and receive messages by reading its complementary endpoint. Channel endpoints can also be called *session endpoints*. Whenever the context allows it, we use "endpoints" to refer to both session and channel endpoints.

We closely follow the presentation of Vasconcelos [Vas12]. In § 2.2.1, we summarize the syntax and semantics of π. Next, in § 2.2.2, we summarize the type system for π. Finally, we summarize the guarantees ensured by typing and present some examples of typing derivations in § 2.2.3.

2.2.1 *Syntax and Semantics*

We assume a basic set of variables, represented as a countably infinite set $\mathcal{V}_s$, ranged over by $x, y, \ldots$. Channels are represented as pairs of variables, called *co-variables*. Messages are then represented by *values*, ranged over by $v, v', u, u', \ldots$ and whose base set is called $\mathcal{U}_s$. Values can be both variables and the boolean constants $\texttt{tt}, \texttt{ff}$. We also use $l, l', \ldots$ to range over a countably infinite set of *labels*, denoted $\mathcal{B}_\pi$. We write $\widetilde{x}$ to denote a finite sequence of variables $x_1, \ldots, x_n$ with $n \geq 0$ (and similarly for sequences of other elements).

Below, we present the construction rules for π *processes* as a formal grammar. We will often refer to the set of all processes as π.

Definition 2.10 (π)**.** The grammar below defines the construction rules for π processes:

$$\begin{array}{llll}
(\textit{Processes}) & P, Q ::= & x\langle v\rangle.P & (\textit{Output}) \\
 & \mid & x(y).P & (\textit{Input}) \\
 & \mid & x \triangleleft l.P & (\textit{Selection}) \\
 & \mid & x \triangleright \{l_i : P_i\}_{i \in I} & (\textit{Branching}) \\
 & \mid & b?\,(P):(Q) & (\textit{Conditional}) \\
 & \mid & *\,x(y).P & (\textit{Replicated input}) \\
 & \mid & (\boldsymbol{\nu} xy)P & (\textit{Restriction}) \\
 & \mid & P \mid Q & (\textit{Parallel composition}) \\
 & \mid & \mathbf{0} & (\textit{Inaction})
\end{array}$$

Process $x\langle v\rangle.P$ sends value v over x and then continues as P; dually, process $x(y).Q$ expects a value v on x that will replace all free occurrences of y in Q. Processes $x \triangleleft l_j.P$ and $x \triangleright \{l_i : Q_i\}_{i\in I}$ define a labeled choice mechanism, with labels indexed by the finite set I: given $j \in I$, process $x \triangleleft l_j.P$ uses x to select l_j from $x \triangleright \{l_i : Q_i\}_{i\in I}$ and triggering process Q_j. We assume pairwise distinct labels. The conditional process $v?\,(P):(Q)$ behaves as P if v evaluates to $\texttt{tt}$; otherwise it behaves as Q. Process $*\,x(y).P$ denotes a replicated input process, which allows us to express infinite server behaviors. The restriction $(\boldsymbol{\nu} xy)P$ binds together x and y in P, thus indicating that they are two endpoints of the same channel; i.e., the same session. Parallel composition and inaction are standard. We often write $\prod_{i=1}^{n} P_i$ to stand for $P_1 \mid \cdots \mid P_n$. Furthermore, we say that the parallel sub-processes of $\prod_{i=1}^{n} P_i$ are *threads*. Moreover, whenever ' $\mid$ ' does not occurs at the top-level of a process, we say that process is a *sequential thread*. In $x(y).P$ and $*\,x(y).P$ (resp. $(\boldsymbol{\nu} yz)P$) occurrences of y (resp. y, z) are bound with scope P. The set of free variables of P, denoted $\mathsf{fv}_\pi(P)$, is defined as expected.

The operational semantics for π is given as a *reduction relation* $\longrightarrow$, the smallest relation generated by the rules in Fig. 2.1. Reduction expresses the computation steps that a process performs on its own. It relies on a *structural congruence* on processes, given below.

Definition 2.11. The structural congruence relation for π processes is the smallest congruence relation $\equiv_{\mathsf{S}}$ that satisfies the following axioms and identifies processes up to renaming of bound variables (i.e., α-conversion). The renaming of bound variables is denoted $\equiv_\alpha$.

$$\begin{array}{lll}
(\mathrm{SC}_\pi{:}1) & (\mathrm{SC}_\pi{:}2) & (\mathrm{SC}_\pi{:}3) \\
P \mid \mathbf{0} \equiv_{\mathsf{S}} P & P \mid Q \equiv_{\mathsf{S}} Q \mid P & (P \mid Q) \mid R \equiv_{\mathsf{S}} P \mid (Q \mid R)
\end{array}$$

$$\begin{array}{ll}
(\mathrm{SC}_\pi{:}4) & (\mathrm{SC}_\pi{:}5) \\
(\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)P \equiv_{\mathsf{S}} (\boldsymbol{\nu} wz)(\boldsymbol{\nu} xy)P & (\boldsymbol{\nu} xy)\mathbf{0} \equiv_{\mathsf{S}} \mathbf{0}
\end{array}$$

$$(\mathrm{SC}_\pi{:}6)\ \frac{P \equiv_\alpha Q}{P \equiv_{\mathsf{S}} Q} \qquad (\mathrm{SC}_\pi{:}7)\ \frac{x, y \notin \mathsf{fv}_\pi(Q)}{(\boldsymbol{\nu} xy)P \mid Q \equiv_{\mathsf{S}} (\boldsymbol{\nu} xy)(P \mid Q)}$$

We briefly comment on the rules in Fig. 2.1. Reduction requires an enclosing restriction $(\boldsymbol{\nu} xy)(\cdots)$; this represents the fact that a session connecting endpoints x and y has been already established. Hence, communication cannot occur on free variables, as there is no way to tell what is the pair of interacting co-variables. In Rules $\lfloor\text{Com}\rfloor$, $\lfloor\text{Sel}\rfloor$, and $\lfloor\text{Rep}\rfloor$, the restriction is *persistent* after each reduction, to allow further synchronizations on co-variables x and y. Moreover, in the same rules, process R collects all the threads that may share variables x and y.

Rule $\lfloor\text{Com}\rfloor$ represents the synchronous communication of value v through endpoint x to endpoint y. Furthermore, Rule $\lfloor\text{Sel}\rfloor$ formalizes a labeled choice mechanism, in which communication of a label l_j is used to choose which of the Q_i will be executed, Rule $\lfloor\text{Rep}\rfloor$ is similar to Rule $\lfloor\text{Com}\rfloor$, and used to spawn a new copy of Q, available as a replicated server. Rules $\lfloor\text{IfT}\rfloor$ and $\lfloor\text{IfF}\rfloor$ are self-explanatory. Rules for reduction within parallel and restriction contexts, together with the reduction up to $\equiv_S$ are as expected.

To reason compositionally about the syntactic structure of π processes, we introduce the idea of (*evaluation*) *contexts* for π. A π context represents a process with a "hole", which may be filled by another π process. Formally:

Definition 2.12 (Contexts for π). The syntax of (evaluation) contexts in π is given by the following grammar:

$$E ::= - \;\big|\; E \mid P \;\big|\; P \mid E \;\big|\; (\boldsymbol{\nu} xy)(E)$$

where P is a π process and '$-$' represents a hole. We write $C[-]$ to range over contexts $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(-)$. We also write $E[P]$ (resp. $C[P]$) denote the process obtained by filling '$-$' with P.

2.2.2 Type System

We now summarize the type system presented in [Vas12]. We use $q, q', \ldots$, to range over *qualifiers*; $p, p', \ldots$, to range over *pre-types*; $T, T', U, U', \ldots$ to range over *types*, and $\Gamma, \Gamma', \ldots$ to range over *typing environments*—i.e., sets that contains pairs $x : T$, where x is a variable and T is a type.

Definition 2.13 (Syntax). The syntax of types and typing contexts is given in Fig. 2.2. Given two typing environments Γ_1 and Γ_2, we write Γ_1, Γ_2 to denote their concatenation. We write $dom(\Gamma)$ to denote the domain of Γ.

Intuitively, pre-types represent pure communication behavior (e.g., send, receive, selection, and branching), which can then be assigned a qualifier to indicate whether the behavior is *unrestricted* or *linear* (see below). Pre-type $!T_1.T_2$ represents a protocol that sends a value of type T_1 and then continues according to type T_2. Dually, pre-type $?T_1.T_2$ represents a protocol that receives a value of type T_1 and then proceeds according to type T_2. Pre-types $\oplus\{l_i : T_i\}_{i\in I}$ and $\&\{l_i : T_i\}_{i\in I}$ denote labeled selection (internal choice) and branching (external choice), respectively.

Qualifiers give additional information about the behavior represented by pre-types. Briefly, linearly qualified pre-types can only be assigned to variables that *do not* appear shared among threads (i.e., they only occur in exactly one sequential thread), whereas unrestricted pre-types may be assigned to variables shared among different threads. Types can be one of the following: (1) `bool`, used for constants and

(*Qualifiers*)	q	$::=$	$\mathtt{lin}$	(*Linear*)
		$\mid$	un	(*Unrestricted*)
(*Pre-types*)	p	$::=$	$?T.T$	(*Receive*)
		$\mid$	$!T.T$	(*Send*)
		$\mid$	$\oplus\{l_i : T_i\}_{i\in I}$	(*Select*)
		$\mid$	$\&\{l_i : T_i\}_{i\in I}$	(*Branch*)
(*Types*)	T	$::=$	$\mathtt{bool}$	(*Boolean*)
		$\mid$	end	(*Termination*)
		$\mid$	qp	(*Qualified Pre-type*)
		$\mid$	a	(*Type Variable*)
		$\mid$	$\mu\mathsf{a}.T$	(*Recursive Type*)
(*Typing environments*)	Γ	$::=$	$\emptyset$	(*Empty Environment*)
		$\mid$	$\Gamma, x : T$	(*Assumption, with $x \notin dom(\Gamma)$*)

Figure 2.2: Session Types: Qualifiers, Pre-types, Types, and Typing Environments.

variables; (2) end, which indicates a terminated behavior; (3) qualified pre-types, which are assigned to variables and represent communication patterns (see below); or (4) recursive types for disciplining potentially infinite communication patterns. In π, recursive types are considered equi-recursive; i.e., types are equal to their unfolding because are equated to the same regular infinite trees, and contractive; i.e., containing no subexpression of the form $\mu\mathbf{t}_1 \ldots \mu\mathbf{t}_n.\mathbf{t}_1$ [Pie02]. Following [Vas12], we omit end at the end of types whenever it is not needed; we also write recursive types $\mu\mathsf{a}.\,\mathsf{un}!T.\mathsf{a}$ and $\mu\mathsf{a}.\,\mathsf{un}?T.\mathsf{a}$ as $*\,!T$ and $*\,?T$, respectively.

Observe that $\mathtt{bool}$ and end are always assumed to be unrestricted. Similarly, the qualifier of a recursive type $T = \mu\mathsf{a}.T'$is obtained via unfolding, by assigning the qualifier of the body T' to type T.

Following [Vas12], we define predicates over types to indicate which types can be assigned to shared variables and which not:

Definition 2.14 (Predicates for Session Types). Let T be a session type (cf. Fig. 2.2). We define $q(T)$ and $q(\Gamma)$ for each qualifier $q \in \{\mathsf{un}, \mathtt{lin}\}$:

- $\mathtt{lin}(T)$ if and only if true.
- $\mathsf{un}(T)$ if and only if $T = \mathtt{bool}$ or $T = \mathsf{end}$ or $T = \mathsf{un}\,p$.
- $q(\Gamma)$ if and only if $x : T \in \Gamma$ implies $q(T)$.

Predicate $\mathtt{lin}(T)$ is true for all types, whereas predicate $\mathsf{un}(\cdot)$ only holds for the terminated type, the ground type $\mathtt{bool}$ and pre-types qualified with un. Predicates $\mathsf{un}(\cdot)$ and $\mathtt{lin}(\cdot)$ are extended to typing environments as expected.

Session type systems depend on *type duality* to relate two types with complementary (or opposite) behaviors: e.g., the dual of input is output (and vice versa); branching is the dual of selection (and vice versa). Following [Vas12], we define duality as an inductive function on the structure of types. We write $\overline{T}$ to denote the dual of type T.

Definition 2.15 (Duality of Session Types). For every type T except bool, we define duality by:

$$\overline{\mathsf{end}} \stackrel{\mathsf{def}}{=} \mathsf{end} \quad \overline{\mathsf{a}} \stackrel{\mathsf{def}}{=} \mathsf{a} \quad \overline{!T.U} \stackrel{\mathsf{def}}{=} ?T.\overline{U} \quad \overline{?T.U} \stackrel{\mathsf{def}}{=} !T.\overline{U} \quad \overline{\oplus\{l_i : T_i\}_{i\in I}} \stackrel{\mathsf{def}}{=} \&\{l_i : \overline{T_i}\}_{i\in I}$$
$$\overline{\&\{l_i : T_i\}_{i\in I}} \stackrel{\mathsf{def}}{=} \oplus\{l_i : \overline{T_i}\}_{i\in I} \quad \overline{\mu\mathsf{a}.T} \stackrel{\mathsf{def}}{=} \mu\mathsf{a}.\overline{T}$$

We use a *typing environment splitting* operator on typing environments, denoted '$\circ$', to maintain the linearity invariant for variables on typing derivations.

Definition 2.16 (Typing Environment Splitting [Vas12]). Let Γ_1 and Γ_2 be two typing environments. The (typing) environment splitting of Γ_1 and Γ_2, written $\Gamma_1 \circ \Gamma_2$, is defined as follows:

$$\emptyset \circ \emptyset = \emptyset \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathsf{un}(T)}{(\Gamma_1, x : T) \circ (\Gamma_2, x : T) = \Gamma, x : T}$$
$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathtt{lin}(T)}{(\Gamma_1, x : T) \circ \Gamma_2 = \Gamma, x : T} \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathtt{lin}(T)}{\Gamma_1 \circ (\Gamma_2, x : T) = \Gamma, x : T}$$

We also define a '+' operation to correctly update the typing environment during derivations.

$$\frac{x : T \notin \Gamma}{\Gamma + x : T = \Gamma, x : T} \qquad \frac{\mathsf{un}(T)}{(\Gamma, x : T) + x : T = \Gamma, x : T}$$

Given a typing environment Γ and a process P, typing judgments are of the form $\Gamma \vdash P$. Fig. 2.3 gives the typing rules for π processes; some intuitions follow (see [Vas12] for full details).

Rules (T:Bool) and (T:Var) are for variables; in both cases, we check that all variables assigned to types satisfying $\mathtt{lin}(\cdot)$ are consumed, by requiring $\mathsf{un}(\Gamma)$. Rule (T:In) types an input process: it checks whether x has the right type and checks the continuation; it also adds variable y with type T and updates x in Γ with type U. To type-check a process $x\langle v\rangle.P$, Rule (T:Out) splits the typing environment in three parts: the first is used to check the type of the sent object v; the second is used to check the type of subject x; the third is used to check the continuation P. Rules (T:Bra) and (T:Sel) type-check label branching and label selection processes, and work similarly to Rules (T:In) and (T:Out), respectively. Rule (T:Rin) types a replicated input $* x(y).P$; it requires the environment Γ to be unrestricted (hence, no split is needed). The rule also requires that the environment contains an input qualified with un, and that the continuation P is typed with an environment that contains $y : T$ and $x : U$.

Rule (T:Par) types parallel composition using the (environment) splitting operation to divide resources among the two threads. Rule (T:Res) types the restriction operator by performing a duality check on the types of the co-variables. Rule (T:If) type-checks the conditional process. Finally, Rule (T:Nil) types the inactive process **0**; it also checks that the typing environment only contains unrestricted variables.

2.2.3 *Typing Properties*

As a way to introduce the fundamental guarantees expected from the type system of a session calculus, we summarize the most prominent properties in [Vas12].

$$(\text{T:Bool})\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathtt{ff}, \mathtt{tt} : \mathtt{bool}} \qquad (\text{T:Var})\ \frac{\mathsf{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T}$$

$$(\text{T:In})\ \frac{\Gamma_1 \vdash x : q?T.U \quad (\Gamma_2 + x : U), y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x(y).P}$$

$$(\text{T:Out})\ \frac{\Gamma_1 \vdash x : q!T.U \quad \Gamma_2 \vdash v : T \quad \Gamma_3 + x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x\langle v\rangle.P}$$

$$(\text{T:Sel})\ \frac{\Gamma_1 \vdash x : q\oplus\{l_i : T_i\}_{i\in I} \quad \Gamma_2 + x : T_j \vdash P \quad j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P}$$

$$(\text{T:Bra})\ \frac{\Gamma_1 \vdash x : q\&\{l_i : T_i\}_{i\in I} \quad \forall i \in I.\, \Gamma_2 + x : T_i \vdash P_i}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i\in I}}$$

$$(\text{T:RIn})\ \frac{\mathsf{un}(\Gamma) \quad \Gamma \vdash x : \mathsf{un}?T.U \quad (\Gamma + x : U), y : T \vdash P}{\Gamma \vdash * x(y).P}$$

$$(\text{T:Par})\ \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad (\text{T:Res})\ \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)P}$$

$$(\text{T:If})\ \frac{\Gamma_1 \vdash v : \mathtt{bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash v ?\, (P) \!:\! (Q)} \qquad (\text{T:Nil})\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}$$

Figure 2.3: Session types: typing rules for π processes.

Subject congruence proves that typing is preserved by structural congruence. In other words, if P is typable with some environment Γ, then all of its structural congruent processes Q are also required to be typable with Γ. This result is useful for proving *subject reduction*.

Lemma 2.17 (Subject Congruence [Vas12]). *If* $\Gamma \vdash P$ *and* $P \equiv_{\mathsf{S}} Q$ *then* $\Gamma \vdash Q$.

Subject reduction ensures that typing is preserved by the reduction relation given in Fig. 2.1. Notice that due to Rule $\lfloor\text{Str}\rfloor$, subject congruence becomes necessary to prove this statement.

Theorem 2.18 (Subject Reduction [Vas12]). *If* $\Gamma \vdash P$ *and* $P \longrightarrow^* Q$ *then* $\Gamma \vdash Q$.

Next, we collect results that concern process structure: often, we refer to these statements collectively as *type safety* results; they aim to show that the type system rules out *communication errors*. We summarize these statements as they were presented in [Vas12]. We require auxiliary notions for *pre-redexes*, *redexes*, and *well-formed processes*, given next:

Definition 2.19 (Pre-redexes and Redexes). We shall use the following terminology:

- We say $x\langle v\rangle.P$, $x(y).P$, $x \triangleleft l.P$, $x \triangleright \{l_i : P_i\}_{i\in I}$, and $* x(y).P$ are *pre-redexes* (*at variable* x).
- A *redex* is a process R such that $(\boldsymbol{\nu} xy)R \longrightarrow$ and:

1. $R = v?\,(P):(Q)$ with $v \in \{\mathtt{tt}, \mathtt{ff}\}$ (or)
2. $R = x\langle v\rangle.P \mid y(z).Q$ (or)
3. $R = x\langle v\rangle.P \mid *\,y(z).Q$ (or)
4. $R = x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$, with $j \in I$.

Moreover, a redex R is either *conditional* (if $R = v?\,(P):(Q)$) or *communicating* (otherwise).

In a session calculus like π, type safety results often depend on a *well-formedness* property, such as the one presented below. The following well-formed property in particular, showcases the flexibility that recursive unrestricted types introduce for the type system presented in [Vas12].

Definition 2.20 (Well-Formed Process). A π process P_0 is *well-formed* if for each of its structural congruent processes $P_0 \equiv_{\mathsf{S}} (\boldsymbol{\nu} x_1 y_1)\dots(\boldsymbol{\nu} x_n y_n)(P \mid Q \mid R)$, with $n \geq 0$, the following conditions hold:

1. If $P \equiv_{\mathsf{S}} v?\,(P'):(P'')$ then $v = \mathtt{tt}$ or $v = \mathtt{ff}$.
2. If P and Q are prefixed on the same variable, then they are of the same nature (input, output, selection, or branching).
3. If P is prefixed at x_i and Q is prefixed at y_i, $1 \leq i \leq n$, then $P \mid Q$ is a redex.

Before stating type safety, we introduce a useful notation to characterize *programs*: an important class of π processes.

Notation 2.21 ((Typable) Programs). A process P such that $\mathsf{fv}_\pi(P) = \emptyset$ is called *a program*. Therefore, program P is typable if it is well-typed under the empty environment ($\vdash P$).

Type safety, stated below, ensures that all typable programs are well-formed. Therefore, type safety ensures that no communication mismatches occur (absence of communication errors).

Theorem 2.22 (Type Safety [Vas12]). *If $\vdash P$ then P is well-formed.*

The following corollary shows that well-formedness is preserved under reduction for well-typed programs. It follows from Thm. 2.18 and Thm. 2.22.

Corollary 2.23. *If $\vdash P$ and $P \longrightarrow^* Q$ then Q is well-formed.*

We now introduce a useful notation that allows us to refer to both inputs and replicated inputs at the same time, whenever we do not wish to distinguish between them.

Notation 2.24. Let P be a π process. We will write $P = \diamond\, y(z).P'$, to represent whenever $P = y(z).P'$ or $P = *\,y(z).P'$.

We close this section by presenting some well-typed processes, which exhibits the flexibility of recursive types, and their corresponding typing derivations. These examples serve as illustration of the type system.

Example 2.25 (Well-Typed Processes). Let us consider the following well-typed π processes:

$$\begin{aligned}
P_1 &= (\boldsymbol{\nu} xy)(x\langle v_1\rangle.Q_1 \mid x\langle v_2\rangle.Q_2 \mid *\, y(z).Q_3)\\
P_2 &= (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(x\langle \texttt{ff}\rangle.\mathbf{0} \mid *\, y(u).\mathbf{0} \mid *\, y(u').(w\langle \texttt{tt}\rangle.\mathbf{0} \mid z(u'').\mathbf{0}))\\
P_3 &= (\boldsymbol{\nu} wz)(\boldsymbol{\nu} xy)(x\langle z\rangle.w(u').\mathbf{0} \mid *\, y(u).u\langle \texttt{tt}\rangle.\mathbf{0})\\
P_4 &= (\boldsymbol{\nu} xy)(x\langle \texttt{tt}\rangle.\mathbf{0} \mid x\langle \texttt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle \texttt{tt}\rangle.\mathbf{0})
\end{aligned}$$

These processes are meant to give the reader an idea of the kind of processes that are well-typed with unrestricted types in π. In particular, all the processes above, except for P_3, exhibit some form on nondeterminism in terms of reductions. For example, process P_1 has the following possible reductions:

$$\begin{aligned}
P_1 &\longrightarrow (\boldsymbol{\nu} xy)(Q_1 \mid x\langle v_2\rangle.Q_2 \mid Q_3\{{}^{v_1}\!/z\} \mid *\, y(z).Q_3)\\
P_1 &\longrightarrow (\boldsymbol{\nu} xy)(x\langle v_1\rangle.Q_1 \mid Q_2 \mid Q_3\{{}^{v_2}\!/z\} \mid *\, y(z).Q_3)
\end{aligned}$$

Similarly, P_2 also has two possible reductions:

$$\begin{aligned}
P_2 &\longrightarrow (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(*\, y(u).\mathbf{0} \mid *\, y(u').(w\langle \texttt{tt}\rangle.\mathbf{0} \mid z(u'').\mathbf{0}))\\
P_2 &\longrightarrow (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(*\, y(u).\mathbf{0} \mid w\langle \texttt{tt}\rangle.\mathbf{0} \mid z(u'').\mathbf{0} \mid *\, y(u').(w\langle \texttt{tt}\rangle.\mathbf{0} \mid z(u'').\mathbf{0}))
\end{aligned}$$

Notice that although both P_1 and P_2 are similar in the sense that both allow two nondeterministic reductions, each of them represents a very different scenario. The intuitive model behind P_1 corresponds to two clients attempting to interact with a single replicated server, whereas in P_2, we observe two replicated servers ready to receive a message from a single client. Throughout this work, these two scenarios will be known as *output* and *input* races, respectively, and will be detailed in further chapters.

Below we present the typing derivation trees of P_3 and P_4, as examples of how recursive, unrestricted types are used. Notice that we only mention the application of typing rules for processes, while omitting the sub-trees corresponding to variables and constants. For P_2 we have:

$$\text{(T:Res)}\ \dfrac{\text{(T:Par)}\ \dfrac{D_1 \qquad \text{(T:RIn)}\ \dfrac{\text{(T:Out)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(\Gamma_1, u : \mathsf{end})}{\Gamma_1, u : \mathsf{end} \vdash \mathbf{0}}}{\Gamma_1, u : !\texttt{bool} \vdash u\langle \texttt{tt}\rangle.\mathbf{0}}}{\Gamma_1 \vdash *\, y(u).u\langle \texttt{tt}\rangle.\mathbf{0}}}{\underbrace{x : *!(!\texttt{bool}), y : *?(!\texttt{bool})}_{\Gamma_1}, \underbrace{w : ?\texttt{bool}, z : !\texttt{bool}}_{\Gamma_2} \vdash \begin{array}{l} x\langle z\rangle.w(u').\mathbf{0} \mid \\ *\, y(u).u\langle \texttt{tt}\rangle.\mathbf{0}\end{array}}}{\vdash (\boldsymbol{\nu} wz)(\boldsymbol{\nu} xy)(x\langle z\rangle.w(u').\mathbf{0} \mid *\, y(u).u\langle \texttt{tt}\rangle.\mathbf{0})}$$

where typing derivation sub-tree D_1 corresponds to:

$$\text{(T:Out)}\ \dfrac{\text{(T:In)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(\Gamma_1, u' : \texttt{bool}, w : \mathsf{end})}{\Gamma_1, u' : \texttt{bool}, w : \mathsf{end} \vdash \mathbf{0}}}{\Gamma_1, w : ?\texttt{bool} \vdash w(u').\mathbf{0}}}{\Gamma_1, \Gamma_2 \vdash x\langle z\rangle.w(u').\mathbf{0}}$$

A highlight from the typing derivation above is the fact that the typing environment (i.e., $\Gamma = \Gamma_1, \Gamma_2$) displays both linear and unrestricted types. It can be seen that $\mathsf{un}(\Gamma_1)$ holds, hence we see Γ_1 appear throughout the derivation, whereas $\mathtt{lin}(\Gamma_2)$ holds, and thus, Γ_2 is only used in D_1.

Finally, we show the typing derivation tree for P_4. It is also important to notice that in this case, the typing environment only contains unrestricted types and therefore, it is unchanged during the derivation.

$$\text{(T:Res)}\ \dfrac{\text{(T:Par)}\ \dfrac{\text{(T:Out)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}}{\Gamma \vdash x\langle \mathtt{tt}\rangle.\mathbf{0}} \qquad \text{(T:Par)}\ \dfrac{D_2 \quad D_3}{\Gamma \vdash x\langle \mathtt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle \mathtt{tt}\rangle.\mathbf{0}}}{\underbrace{x : *!\mathsf{bool}, y : *?\mathsf{bool}}_{\Gamma} \vdash x\langle \mathtt{tt}\rangle.\mathbf{0} \mid x\langle \mathtt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle \mathtt{tt}\rangle.\mathbf{0}}}{\vdash (\boldsymbol{\nu} xy)(x\langle \mathtt{tt}\rangle.\mathbf{0} \mid x\langle \mathtt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle \mathtt{tt}\rangle.\mathbf{0})}$$

where the typing derivation sub-trees D_2 and D_3 are, respectively:

$$\text{(T:Out)}\ \dfrac{\text{(T:In)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(\Gamma, u : \mathsf{bool})}{\Gamma, u : \mathsf{bool} \vdash \mathbf{0}}}{\Gamma \vdash y(u).\mathbf{0}}}{\Gamma \vdash x\langle \mathtt{tt}\rangle.y(u).\mathbf{0}} \qquad \text{(T:In)}\ \dfrac{\text{(T:Out)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(\Gamma, u' : \mathsf{bool})}{\Gamma, u' : \mathsf{bool} \vdash \mathbf{0}}}{\Gamma, u' : \mathsf{bool} \vdash x\langle \mathtt{tt}\rangle.\mathbf{0}}}{\Gamma \vdash y(u').x\langle \mathtt{tt}\rangle.\mathbf{0}}$$

△

2.3 Linear Concurrent Constraint Programming (lcc)

In this section we introduce the first target language: linear concurrent constraint calculus (lcc). It is a variant of ccp tightly related to linear logic [Gir87]. As in ccp, concurrent processes in lcc interact via a (*global*) *constraint store* that defines a synchronization mechanism for *tell* and *ask* operations.

The relation with linear logic follows in lcc due to the fact that the store is *non-monotonic*, contrary to ccp. A non-monotonic store ensures that constraints can be consumed whenever information is asked from it. Therefore, lcc is a formalism that enables the specification and analysis of concurrent systems with partial information and *linear resources* [FRS01, Hae11]. In particular, reasoning techniques over processes based on observational equivalences [Hae11] and phase semantics [FRS01] have been developed.

We follow the presentation of lcc given in [Hae11]. First introduce the syntax and semantics of lcc (cf. § 2.3.1). Next, we introduce some important observational equivalences that will be used throughout this work and we conclude with some examples (cf. § 2.3.2).

2.3.1 *Syntax and Semantics*

Variables are ranged over by $x, y, \ldots$ and belong to the countably infinite set $\mathcal{V}_l$. Similarly, we assume that Σ_c and Σ_f correspond to sets of predicate and function symbols, respectively. First-order terms, built from $\mathcal{V}_l$ and Σ_f, will be denoted by $t, t', \ldots$. An arbitrary predicate in Σ_c is denoted $\varphi(\widetilde{t})$.

Definition 2.26 (Syntax). The syntax for `lcc` is given by the following grammar:

$$
\begin{array}{llll}
(\mathit{Constraints}) & c, d & ::= \mathtt{ff} & (\mathit{False}) \\
 & & \mid \mathtt{tt} & (\mathit{True}) \\
 & & \mid \varphi(\widetilde{t}) & (\mathit{Predicates}) \\
 & & \mid c \otimes d & (\mathit{Linear\ Conjunction}) \\
 & & \mid \exists \widetilde{x}.c & (\mathit{Existential}) \\
 & & \mid\, !\, c & (\mathit{Bang}) \\
(\mathit{Guards}) & G, G' & ::= \forall \widetilde{x}(c \rightarrow P) & (\mathit{Abstraction}) \\
 & & \mid G + G' & (\mathit{Nondeterministic\ Sum}) \\
(\mathit{Processes}) & P, Q & ::= \overline{c} & (\mathit{Tell}) \\
 & & \mid G & (\mathit{Guards}) \\
 & & \mid P \parallel Q & (\mathit{Parallel}) \\
 & & \mid \exists \widetilde{x}.\, P & (\mathit{Hiding}) \\
 & & \mid\, !\, P & (\mathit{Replication})
\end{array}
$$

Constraints represent the pieces of information that can be posted to the store, as well as be asked from it. Constant `tt`, the multiplicative identity, denotes truth; constant `ff` denotes falsehood. Logic connectives used as constructors include the multiplicative conjunction ($\otimes$), bang ($!$), and the existential quantifier ($\exists \widetilde{x}$). Notation $c\{\widetilde{t}/\widetilde{x}\}$ denotes the constraint obtained by the (capture-avoiding) substitution of the free occurrences of x_i for t_i in c, with $|\widetilde{t}| = |\widetilde{x}|$ and pairwise distinct x_i's. Process substitution $P\{\widetilde{t}/\widetilde{x}\}$ is defined analogously.

The syntax for guards includes nondeterministic choices, denoted $G_1 + G_2$, and *parametric asks*, denoted $\forall \widetilde{x}(c \rightarrow P)$, which spawns process $P\{\widetilde{t}/\widetilde{x}\}$ if the current constraint store entails constraint $c\{\widetilde{t}/\widetilde{x}\}$; the exact operational semantics for parametric ask operators (and its interplay with linear constraints) is detailed below. When $\widetilde{x}$ is empty (a parameterless ask), $\forall \widetilde{x}(c \rightarrow P)$ is written $\forall \epsilon(c \rightarrow P)$.

Besides guards, the syntax of processes includes the *tell operator* $\overline{c}$ that adds constraint c to the current store. Moreover, process constructs include parallel composition $P \parallel Q$, which has the expected reading, hiding $\exists \widetilde{x}.\, P$, which declares x as being local to P, and replication $!\, P$, that provides infinitely many copies of P. We use notation $\prod_{1 \leq i \leq n} P_i$ (with $n \geq 1$) to stand for process $P_1 \parallel \cdots \parallel P_n$. Universal quantifiers in ask operators and existential quantifiers in hiding operators bind their respective variables. Given this, the set of free variables in constraints and processes is defined as expected, and denoted $\mathsf{fv}(\cdot)$.

We follow closely the semantics for `lcc` processes given in [Hae11], which is parametric in a *constraint system*, as defined next.

Definition 2.27 (Constraint System). A constraint system is a triplet $(\mathcal{C}, \Sigma, \vdash)$, where Σ contains Σ_c (i.e., the set of predicates) and Σ_f (i.e., the set of functions and constants). Also, $\mathcal{C}$ is the set of constraints obtained by using the grammar in Def. 2.26

$$
(\text{Ax})\ \frac{}{c \vdash c} \qquad (\text{T1})\ \frac{}{\vdash \mathtt{tt}} \qquad (\text{Cut})\ \frac{\Gamma, c \vdash d \quad \Delta \vdash c}{\Gamma, \Delta \vdash d} \qquad (\text{T2})\ \frac{\Gamma \vdash c}{\Gamma, \mathtt{tt} \vdash c}
$$

$$
(\text{L}_{\otimes})\ \frac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} \qquad (\text{R}_{\otimes})\ \frac{\Gamma \vdash c_1 \quad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2}
$$

$$
(\text{L}_{\exists})\ \frac{\Gamma, c \vdash d}{\Gamma, \exists x.c \vdash d}\ x \notin \mathsf{fv}(\Gamma, d) \qquad (\text{R}_{\exists})\ \frac{\Gamma \vdash c\{^{t}\!/\!_{x}\}}{\Gamma \vdash \exists x.c}
$$

$$
(!_1)\ \frac{\Gamma, c \vdash d}{\Gamma, !\,c \vdash d} \qquad (!_2)\ \frac{!\,\Gamma \vdash d}{!\,\Gamma \vdash !\,d} \qquad (!_3)\ \frac{\Gamma \vdash d}{\Gamma, !\,c \vdash d} \qquad (!_4)\ \frac{\Gamma, !\,c, !\,c \vdash d}{\Gamma, !\,c \vdash d}
$$

Figure 2.4: Intuitionistic linear sequents for `lcc` (cf. Def. 2.27).

and Σ. Relation $\Vdash$ is a subset of $\mathcal{C} \times \mathcal{C}$ that defines the non-logical axioms of the constraint system. Relation $\vdash$ is the least subset of $\mathcal{C}^* \times \mathcal{C}$ containing $\Vdash$ and closed by the rules in Fig. 2.4. We write $c \dashv\vdash d$ whenever both $c \vdash d$ and $d \vdash c$ hold.

The semantics of `lcc` processes is defined as a Labeled Transition System (LTS), that relies on a structural congruence on processes, given next.

Definition 2.28 (Structural Congruence). The structural congruence relation for `lcc` processes is the smallest congruence relation $\equiv$ that satisfies α-renaming of bound variables, commutativity and associativity for parallel composition and summation, together with the following identities:

$$
\begin{array}{cccc}
(\text{SC}_1{:}1) & (\text{SC}_1{:}2) & (\text{SC}_1{:}3) & (\text{SC}_1{:}4) \\
P \parallel \overline{\mathtt{tt}} \equiv P & \exists z.\, \overline{\mathtt{tt}} \equiv \overline{\mathtt{tt}} & \exists x.\, \exists y.\, P \equiv \exists y.\, \exists x.\, P & !\,P \equiv P \parallel !\,P
\end{array}
$$

$$
\begin{array}{cccc}
(\text{SC}_1{:}5) & (\text{SC}_1{:}6) & (\text{SC}_1{:}7) & (\text{SC}_1{:}8) \\
\dfrac{c \otimes d \dashv\vdash e}{\overline{c} \parallel \overline{d} \equiv \overline{e}} & \dfrac{P \equiv P'}{P \parallel Q \equiv P' \parallel Q} & \dfrac{z \notin \mathsf{fv}(P)}{P \parallel \exists z.\, Q \equiv \exists z.\, (P \parallel Q)} & \dfrac{P \equiv P'}{\exists x.\, P \equiv \exists x.\, P'}
\end{array}
$$

As customary, a (strong) transition $P \xrightarrow{\alpha}_1 P'$ denotes the evolution of process P to P' by performing the action denoted by the transition label α:

$$
\alpha ::= \tau \mid c \mid (\widetilde{x})\overline{c}
$$

Label τ denotes a silent (internal) action. Label $c \in \mathcal{C}$ denotes a constraint "received" as an input action (but see below) and $(\widetilde{x})\overline{c}$ denotes an output (tell) action in which $\widetilde{x}$ are extruded variables and $c \in \mathcal{C}$. We write $ev(\alpha)$ to refer to these extruded variables.

Before discussing the transition rules (cf. Fig. 2.5), we introduce a key notion: the *most general choice* predicate:

Definition 2.29 (Most General Choice (mgc) [Hae11]). Let c, d, e be constraints, $\widetilde{x}, \widetilde{y}$ be vectors of variables and $\widetilde{t}$ be a vector of terms. We write

$$
\mathbf{mgc}\big(c, \exists \widetilde{y}.(d\{^{\widetilde{t}}\!/\!_{\widetilde{x}}\} \otimes e)\big)
$$

whenever for any constraint e', all terms $\widetilde{t'}$ and all variables $\widetilde{y'}$, if $c \vdash \exists \widetilde{y'}.(d\{^{\widetilde{t'}}\!/\!_{\widetilde{x}}\} \otimes e')$ and $\exists \widetilde{y'}.e' \vdash \exists \widetilde{y}.e$ hold, then $\exists \widetilde{y}.(d\{^{\widetilde{t}}\!/\!_{\widetilde{x}}\}) \vdash \exists \widetilde{y'}.(d\{^{\widetilde{t'}}\!/\!_{\widetilde{x}}\})$ and $\exists \widetilde{y}.e \vdash \exists \widetilde{y'}.e'$.

$$\lfloor \text{C:Out} \rfloor \quad \frac{c \vdash \exists \widetilde{x}.(d \otimes e) \quad \exists \widetilde{x}.d \vdash \exists \widetilde{x'}.d' \quad \mathbf{mgc}\big(c, \exists \widetilde{x}.(d \otimes e)\big) \quad (\widetilde{x} \cup \widetilde{x'}) \cap \mathsf{fv}(c) = \emptyset}{\overline{c} \xrightarrow{(\widetilde{x'})\overline{d'}}_1 \overline{e}}$$

$$\lfloor \text{C:Sync} \rfloor \quad \frac{c \vdash \exists \widetilde{y}(d\{\widetilde{t}/\widetilde{x}\} \otimes e) \qquad \widetilde{y} \cap \mathsf{fv}(c, d, P) = \emptyset \qquad \mathbf{mgc}\big(c, \exists \widetilde{y}.(d\{\widetilde{t}/\widetilde{x}\} \otimes e)\big)}{\overline{c} \parallel \forall \widetilde{x}(d \rightarrow P) \xrightarrow{\tau}_1 \exists \widetilde{y}.\, (P\{\widetilde{t}/\widetilde{x}\} \parallel \overline{e})}$$

$$\lfloor \text{C:In} \rfloor \quad \frac{}{\mathtt{tt} \xrightarrow{c}_1 \overline{c}} \qquad \lfloor \text{C:Comp} \rfloor \quad \frac{P \xrightarrow{\alpha}_1 P' \quad ev(\alpha) \cap \mathsf{fv}(Q) = \emptyset}{P \parallel Q \xrightarrow{\alpha}_1 P' \parallel Q} \qquad \lfloor \text{C:Sum} \rfloor \quad \frac{P \parallel G_i \xrightarrow{\alpha}_1 P' \quad i \in \{1, 2\}}{P \parallel G_1 + G_2 \xrightarrow{\alpha}_1 P'}$$

$$\lfloor \text{C:Ext} \rfloor \quad \frac{P \xrightarrow{(\widetilde{x})\overline{c}}_1 Q}{\exists y.\, P \xrightarrow{(y\widetilde{x})\overline{c}}_1 Q} \qquad \lfloor \text{C:Res} \rfloor \quad \frac{P \xrightarrow{\alpha}_1 P' \quad y \notin \mathsf{fv}(\alpha)}{\exists y.\, P \xrightarrow{\alpha}_1 \exists y.\, P'} \qquad \lfloor \text{C:Cong} \rfloor \quad \frac{P \equiv P' \quad P' \xrightarrow{\alpha}_1 Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha}_1 Q}$$

Figure 2.5: Labeled Transition System (LTS) for `lcc` processes.

Intuitively, the **mgc** predicate allows us to refer formally to decompositions of a constraint c (seen as "linear resources") that do not "lose" or "forget" information in c. This is essential in the presence of linear constraints. For example, assuming that $c \vdash d \otimes e$ holds, we can see that $\mathbf{mgc}(c, d \otimes e)$ holds too, because c is the precise amount of information necessary to obtain $d \otimes e$. However, $\mathbf{mgc}(c \otimes f, d \otimes e)$ does not hold, assuming $f \neq \mathtt{tt}$, since $c \otimes f$ produces more information than the necessary to obtain $d \otimes e$.

We briefly discuss the rules of Fig. 2.5. Rule $\lfloor$C:Out$\rfloor$ formalizes asynchronous tells: using the **mgc** predicate, the emitted constraint is decomposed in two parts: the first one is actually sent (as recorded in the label); the second part is kept as a continuation. (In the rule, these two parts are denoted as d' and e, respectively.) Rule $\lfloor$C:Sync$\rfloor$ formalizes the synchronization between a tell (i.e., an output) and an ask. The constraint mentioned in the tell is decomposed using the **mgc** predicate: in this case, here the first part is used (consumed) to "trigger" the processes guarded by the ask, while the second part is the remaining continuation. Rule $\lfloor$C:In$\rfloor$ asynchronously receives a constraint; it represents the separation between observing an output and its (asynchronous) reception, which is not directly observable. Rule $\lfloor$C:Comp$\rfloor$ enables the parallel composition of two processes P and Q, provided that the variables extruded in an action by P are disjoint from the free variables of Q. Rule $\lfloor$C:Sum$\rfloor$ enables non-deterministic choices at the level of guards. Rules $\lfloor$C:Ext$\rfloor$ and $\lfloor$C:Res$\rfloor$ formalize hiding: the former rule makes local variables explicit in the transition label; the latter rule avoids the hiding of free variables in the label. Finally, Rule $\lfloor$C:Cong$\rfloor$ closes transitions under structural congruence (cf. Def. 2.28).

We shall write $\xrightarrow{\tau}_1^{*}$ to denote the reflexive-transitive closure of `lcc` τ-transitions, i.e., a sequence of zero or more τ-labeled transitions. Whenever the number $k \geq 1$ of τ-transitions is fixed, we write $\xrightarrow{\tau}_1^{k}$. Moreover, when explicit τ-labels are unnecessary we shall write $\longrightarrow_1$, $\longrightarrow_1^*$, and $\longrightarrow_1^k$ to refer to $\xrightarrow{\tau}_1$, $\xrightarrow{\tau}_1^*$, and $\xrightarrow{\tau}_1^k$, respectively. *Weak transitions* are standardly defined: we write $P \overset{\tau}{\Longrightarrow}_1 Q$ if and only if $(P \xrightarrow{\tau}_1^* Q)$ and $P \overset{\alpha}{\Longrightarrow}_1 Q$ if and only if $(P \xrightarrow{\tau}_1^* P' \xrightarrow{\alpha}_1 P'' \xrightarrow{\tau}_1^* Q)$. We use the terms τ-transition and *reduction* interchangeably to refer to τ-labeled transitions.

2.3.2 Observational Equivalences

We now present some useful *observational equivalences* for lcc processes. We require the following auxiliary definition from [Hae11]:

Definition 2.30 ($\mathcal{D}$-Accessible Constraints). Let $\mathcal{D} \subset \mathcal{C}$, where $\mathcal{C}$ is the set of all constraints. The observables of an lcc process P are the set of all $\mathcal{D}$-accessible constraints defined as follows:

$$\mathcal{O}^{\mathcal{D}}(P) \stackrel{\text{def}}{=} \{(\exists \widetilde{x}.c) \in \mathcal{D} \,|\, \text{there exists } P'.\, P \overset{\tau}{\Longrightarrow}_1 \exists \widetilde{x}.(P' \parallel \overline{c})\}$$

We now present a notion of equivalence for lcc processes in the form of a *weak barbed congruence*, as in [Hae11]. We first introduce a notation to parameterize processes in terms of the constraints they can tell and ask. Then, we introduce (evaluation) contexts for lcc.

Notation 2.31 ($\mathcal{DE}$-Processes). Let $\mathcal{D}, \mathcal{E}$ be subsets of the constraint set $\mathcal{C}$. Also, let P be an lcc process (cf. Def. 2.26).

- P is *$\mathcal{D}$-ask restricted* if for every sub-processs $\forall \widetilde{x}(c \rightarrow P')$ in P, we have $\exists \widetilde{z}.c \in \mathcal{D}$.
- P is *$\mathcal{E}$-tell restricted* if for every sub-process $\overline{c}$ in P, we have $\exists \widetilde{z}.c \in \mathcal{E}$.
- If P is both $\mathcal{D}$-ask restricted and $\mathcal{E}$-tell restricted then we call P a *$\mathcal{DE}$-process*.

Definition 2.32 (Contexts in lcc). Let E be the evaluation contexts for lcc as given by the following grammar, where P is an lcc process and '$-$' represents a hole in said process:

$$E ::= - \;\big|\; P \parallel E \;\big|\; E \parallel P \;\big|\; \exists \widetilde{x}.E$$

Given an evaluation context $E[-]$, we write $E[P]$ to denote the lcc process that results from filling in the occurrences of the hole with process P. In lcc, we will say that a context is a *$\mathcal{DE}$-context*, ranged over $C, C', \ldots$, if it is formed only by $\mathcal{DE}$-processes.

Using the previous auxiliary definitions we now present a notion of weak barbed bisimulation and a weak barbed congruence:

Definition 2.33 (Weak $\mathcal{DE}$-Barbed Bisimulation). Let $\mathcal{D} \subseteq \mathcal{C}$ and $\mathcal{E} \subseteq \mathcal{C}$. A *symmetric* relation $\mathcal{R}$ is a $\mathcal{DE}$-barbed bisimulation if, for $\mathcal{DE}$-processes P and Q, $(P, Q) \in \mathcal{R}$ implies:

(1) $\mathcal{O}^{\mathcal{D}}(P) = \mathcal{O}^{\mathcal{D}}(Q)$ (and),

(2) whenever $P \xrightarrow{\tau}_1 P'$ there exists Q' such that $Q \overset{\tau}{\Longrightarrow}_1 Q'$ and $P' \mathcal{R} Q'$.

The largest weak barbed $\mathcal{DE}$-bisimulation is called $\mathcal{DE}$-bisimilarity and is denoted by $\approx_{\mathcal{DE}}$.

Definition 2.34 (Weak $\mathcal{DE}$-Barbed Congruence). We will say that two `lcc` processes P, Q are weakly barbed $\mathcal{DE}$-congruent, denoted by $P \cong_{\mathcal{DE}} Q$, if for every $\mathcal{DE}$-context $E[-]$ we have that $E[P] \approx_{\mathcal{DE}} E[Q]$. We define the weak barbed $\mathcal{DE}$-congruence $\cong_{\mathcal{DE}}$ as the largest $\mathcal{DE}$-congruence that is a weak barbed $\mathcal{DE}$-bisimilarity.

We conclude this section by showing some examples of `lcc` processes and their transitions. We also show some equivalences to help the reader to grasp the notions presented above.

Example 2.35. We use `lcc` to model a simple vending machine that requires a number of coins to return either tea or coffee. First, we assume the nullary functions (i.e., constants) $\Sigma_f = \{\mathsf{coin}/0, \mathsf{coffee}/0, \mathsf{tea}/0\}$ and an empty set of predicates (i.e., $\Sigma_c = \emptyset$). We let the constraint system then be the triple $(\mathcal{C}, \Sigma, \vdash)$, where $\Sigma = \Sigma_f \cup \Sigma_c = \Sigma_f$; $\mathcal{C}$ is obtained from Σ and the formation rules in Def. 2.26; and $\vdash$ is given solely by the deduction rules in Fig. 2.27. We also consider sets $\mathcal{D} = \mathcal{E} = \mathcal{C}$ (cf. Not. 2.31). With the previous assumptions, we can then define our vending machine as:

$$Q_1 = !\,\forall\epsilon\big(\mathsf{coin} \otimes \mathsf{coin} \to \overline{\mathsf{coffee}}\big) \parallel !\,\forall\epsilon\big(\mathsf{coin} \to \overline{\mathsf{tea}}\big)$$

In Q_1 above, we have two replicated abstractions: the leftmost one requires two coins to return a coffee, whereas the rightmost one requires a single coin to return tea. The process above is *suspended*, as no process is providing information to the store. Hence, we require a context that adds the necessary coins to buy a coffee or tea.

Assume a context $E[-] = \overline{\mathsf{coin}} \parallel -$. Then, we have that our complete system can be modeled as $E[Q_1]$. Since we only have available one coin, by applying Rule $\lfloor$C:SYNC$\rfloor$, we have that:

$$E[Q_1] \longrightarrow_1 \overline{\mathsf{tea}} \parallel !\,\forall\epsilon\big(\mathsf{coin} \otimes \mathsf{coin} \to \overline{\mathsf{coffee}}\big) \parallel !\,\forall\epsilon\big(\mathsf{coin} \to \overline{\mathsf{tea}}\big)$$

Note that this synchronization is enabled between the coin and the sub-process in charge of returning tea. More interestingly, the process above allows us to showcase our behavioral equivalences. Consider $Q_2 = !\,\forall\epsilon\big(\mathsf{coin} \to \overline{\mathsf{tea}}\big)$. Then, it can be shown that $E[Q_1] \approx_{\mathcal{DE}} E[Q_2]$. However, notice that $E[Q_1] \not\cong_{\mathcal{DE}} E[Q_2]$: a possible $\mathcal{DE}$-context to be tested could add an additional $\overline{\mathsf{coin}}$ in parallel, which would enable the abstraction that returns a coffee in Q_1. This means that $E[Q_1]$ and $E[Q_2]$ are not bisimilar in that particular context, which also means that these two processes do not satisfy Def. 2.34. △

2.4 ReactiveML (RML)

In this section we introduce ReactiveML [MP05], a synchronous reactive extension of OCaml, based on the model for SRP given in [BdS96]. ReactiveML allows the dynamic creation of processes and unbounded time response for processes. Since it is based on the SRP introduced by Boussinot and De Simone, ReactiveML avoids all the causality issues present in other SRP approaches, such as ESTEREL [BG92].

ReactiveML extends OCaml with *processes*: state machines whose behavior can be executed through several *time instants* (or simply, instants). Processes are the reactive counterpart of OCaml functions, which ReactiveML executes instantaneously. In ReactiveML, synchronization is based on *signals*: events that occur in a single instant.

Signals can trigger reactions in processes, which can be run instantaneously or in the next instant, depending on the process definition. Also, signals carry values and can be emitted by different processes in the same instant.

Below we present the syntax and the semantics of ReactiveML, as they are given in [MP14] (cf. § 2.4.1). We use this work as our source, as it presents the most up-to-date formal specification for the ReactiveML compiler. Next, in § 2.4.2 we present some examples and considerations necessary to use ReactiveML as a target language in our work.

2.4.1 Syntax and Semantics

We assume countable infinite sets of variables $\mathcal{V}_r$ and signal names $\mathcal{S}_r$ (ranged over by x_1, x_2 and n_1, n_2, respectively). To simplify some of our expressiveness results, we assume that $\mathcal{S}_r \subseteq \mathcal{V}_r$. The free variables of a ReactiveML expression, written $\mathsf{fv}(e)$, are defined as expected (i.e., signal declarations, recursive declarations, λ- and let-expressions are binders).

Definition 2.36 (Syntax). The set RML of ReactiveML expressions is defined by the grammar below:

$$
\begin{array}{lcll}
\text{(Values)} & v, v' & ::= & c \mid (v, v') \mid n \mid \lambda x.e \mid \mathtt{process}\ e \mid () \\
\text{(Expressions)} & e, e' & ::= & x \mid c \mid (e, e') \mid \lambda x.e \mid e\ e' \\
 & & & \mid \mathtt{rec}\ x = v \mathtt{match}\ e\ \mathtt{with}\ \{c_i \to e_i\}_{i \in I} \\
 & & & \mid \mathtt{let}\ x = e\ \mathtt{and}\ x = e'\ \mathtt{in}\ e'' \mid \mathtt{run}\ e \mid \mathtt{loop}\ e \\
 & & & \mid \mathtt{signal}_{e'}\ x : e\ \mathtt{in}\ e'' \mid \mathtt{emit}\ e'\ e'' \mid \mathtt{pause} \mid \mathtt{process}\ e \\
 & & & \mid \mathtt{present}\ e ? e' : e'' \mid \mathtt{do}\ e\ \mathtt{when}\ e' \mid \mathtt{do}\ e\ \mathtt{until}\ e'(x) \to e''
\end{array}
$$

The set of values $\mathcal{U}_r$, ranged over $v, v', \ldots$, includes constants c (booleans and the unit value $()$), pairs, names, abstractions, and also includes the process declaration operator $\mathtt{process}\ e$.

The syntax of expressions ranged over by $e, e', \ldots$ extends a standard functional substrate with matching expressions and parallel let definitions (i.e., let/and constructs) with process- and signal-related constructs. Expressions $\mathtt{run}\ e$ and $\mathtt{loop}\ e$ follow the expected intuitions. Expression $\mathtt{signal}_g\ x : d\ \mathtt{in}\ e$ declares a signal x with default value d, bound in e; here g denotes a *gathering function* that collects the distinct values emitted alongside signal x in one instant. When d and g are unimportant (e.g., when the signal will only be emitted once in a single instant), we write simply $\mathtt{signal}\ x\ \mathtt{in}\ P$. We also write $\mathtt{signal}\ x_1, \ldots, x_n\ \mathtt{in}\ e$ when declaring $n > 1$ distinct signals in e. Moreover, expression $\mathtt{emit}\ e_1\ e_2$ emits a signal carrying the value from the instantaneous execution of e_2, only whenever e_1 evaluates to a signal name n. ReactiveML also has an explicit pausing construct, $\mathtt{pause}$, which postpones execution until the next instant. The conditional expression $\mathtt{present}\ e_1 ? e_2 : e_3$ checks the presence of a signal: if e_1 evaluates to some signal name n, which has been emitted in that instant, then, e_2 is run in the same instant; otherwise, if n is not present, e_3 runs in the next instant. Expression $\mathtt{do}\ e\ \mathtt{when}\ e_1$ executes e only when e_1 evaluates a signal name n, with n present in the current instant, and suspends its execution otherwise. Expression $\mathtt{do}\ e_1\ \mathtt{until}\ e(x) \to e_2$ executes e_1 until e evaluates into a signal name n,

$$\begin{aligned}
e_1 \parallel e_2 &\overset{\mathsf{def}}{=} \mathtt{let}\ \_ = e_1\ \mathtt{and}\ \_ = e_2\ \mathtt{in}\ () \\
e_1; e_2 &\overset{\mathsf{def}}{=} \mathtt{let}\ \_ = ()\ \mathtt{and}\ \_ = e_1\ \mathtt{in}\ e_2 \\
\mathtt{await}\ e_1(x)\ \mathtt{in}\ e_2 &\overset{\mathsf{def}}{=} \mathtt{do}\ \mathtt{loop}\ \mathtt{pause}\ \mathtt{until}\ e_1(x) \to e_2 \\
\begin{array}{l}\mathtt{let\ rec\ process} \\ \quad f\ x_1 \ldots x_n = e_1\ \mathtt{in}\ e_2\end{array} &\overset{\mathsf{def}}{=} \mathtt{let}\ f = (\mathtt{rec}\ f = \lambda x_1 \ldots \lambda x_n.\mathtt{process}\ e_1)\ \mathtt{in}\ e_2 \\
\mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 &\overset{\mathsf{def}}{=} \mathtt{match}\ e_1\ \mathtt{with}\ \{\mathtt{tt} \to \mathtt{pause}\,; e_2, \mathtt{ff} \to \mathtt{pause}\,; e_3\}
\end{aligned}$$

Figure 2.6: Derived ReactiveML expressions. Assume $n \geq 1$.

with n present, and (possibly) carrying a value which will substitute x. If this occurs, the execution of e_1 stops at the end of the instant and e_2 is executed in the next one, applying the substitution of x for the values contained in n.

Finally, notice that the constructor for recursive expressions $\mathtt{rec}\ x = v$ only allows for values to be used in the right hand side. This restriction is done to allow only λ-terms or process declarations to be used as the definition of a recursive expression. We sometimes say processes when referring to expressions, in particular, when said expression contains timed subexpressions.

Using these basic constructs, we may obtain the useful derived expressions reported in Fig. 2.6. The first three are the most useful ones: the parallel composition $e_1 \parallel e_2$, the sequential composition $e_1; e_2$, and the awaiting operator $\mathtt{await}\ s(x)\ \mathtt{in}\ e$, which waits until signal s is present to execute e in the next instant. In the awaiting construct, x is a bound variable which is used to extract the values contained in signal s. The last two derived constructs are shortcuts for declaring recursive processes and conditional operators. We say that an expression with no parallel composition operator at top level is a *thread*.

Following [MP14], we define a big-step operational semantics for ReactiveML. Each step in this semantics intuitively corresponds to a single instant in the ReactiveML computation model. Notice that in ReactiveML, every pure OCaml expression is considered to be instantaneous. Therefore, the authors introduce a set of *well-formation* rules in [MP14] to separate instantaneous expressions from reactive expressions. These well-formations rules are summarized in Fig. 2.7.

In the figure, a *well-formation judgment* is of the form $\omega \vdash e$, where ω is either 0 or 1: if $\omega = 1$ then the expression is reactive, otherwise (i.e., $\omega = 0$) then the expression is instantaneous. The most important takeaways from this figure are: (1) the body of OCaml functions *has* to be instantaneous, whereas the body of ReactiveML processes *may be* reactive and (2) the matching operator requires e to be instantaneous. For further details see [MP05, MP14].

We now proceed to define the semantics of ReactiveML. We first require some auxiliary definitions for *signal environments* and *events*. Below, $\uplus$ and $\sqsubseteq$ denote the usual multiset union and inclusion, respectively. Intuitively, signal environments refer to functions that assign signal names to triplets of default values, gathering functions, and a multiset that contains all the other emitted values in that signal.

$$\omega \vdash x \qquad \omega \vdash c \qquad 1 \vdash \texttt{pause}$$

$$\frac{0 \vdash e_1 \quad 0 \vdash e_2}{\omega \vdash (e_1, e_2)} \qquad \frac{0 \vdash e}{\omega \vdash \lambda x.e} \qquad \frac{0 \vdash e_1 \quad 0 \vdash e_2}{\omega \vdash e_1\ e_2} \qquad \frac{0 \vdash v}{\omega \vdash \texttt{rec}\ x = v}$$

$$\frac{\forall i \in I \quad 0 \vdash e \quad \omega \vdash e_i}{\omega \vdash \texttt{match}\ e\ \texttt{with}\ \{c_i \to e_i\}_{i \in I}} \qquad \frac{\omega \vdash e \quad \omega \vdash e_1 \quad \omega \vdash e_2}{\omega \vdash \texttt{let}\ x = e\ \texttt{and}\ x = e_1\ \texttt{in}\ e_2}$$

$$\frac{0 \vdash e}{1 \vdash \texttt{run}\ e} \qquad \frac{1 \vdash e}{1 \vdash \texttt{loop}\ (e)} \qquad \frac{0 \vdash e \quad 0 \vdash e_1 \quad \omega \vdash e_2}{\omega \vdash \texttt{signal}_{e_1}\ x : e\ \texttt{in}\ e_2}$$

$$\frac{0 \vdash e_1 \quad 0 \vdash e_2}{\omega \vdash \texttt{emit}\ e_1\ e_2} \qquad \frac{1 \vdash e}{\omega \vdash \texttt{process}\ e} \qquad \frac{0 \vdash e \quad 1 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{present}\ e ?\, e_1 : e_2} \qquad \frac{1 \vdash e_1 \quad 0 \vdash e_2}{1 \vdash \texttt{do}\ e_1\ \texttt{when}\ e_2}$$

$$\frac{1 \vdash e \quad 0 \vdash e_1 \quad 1 \vdash e_2}{1 \vdash \texttt{do}\ e\ \texttt{until}\ e_1(x) \to e_2}$$

Figure 2.7: Well-formation rules for ReactiveML expressions.

Definition 2.37 (Signal Environment). Let $\mathcal{D}, \mathcal{G}, \mathcal{M}$ be sets of default values, gathering functions, and multisets, respectively. A *signal environment* is a function $S : \mathcal{S}_r \to (\mathcal{D} \times \mathcal{G} \times \mathcal{M})$, denoted $S \overset{\text{def}}{=} [(d_1, g_1, m_1)/n_1, \ldots, (d_k, g_k, m_k)/n_k]$, with $k \geq 1$.

We use the following notations: $S^d(n_i) = d_i$, $S^g(n_i) = g_i$, and $S^m(n_i) = m_i$. Also, $S^v(n_i) = fold\ g_i\ m_i\ d_i$ where $fold$ recursively gathers multiple emissions of different values in the same signal; see [MP05, MP14] for details. An event, similarly to a signal environment, is a function E that associates a signal name n_i to a multiset m_i that represents the values emitted during the current instant.

Definition 2.38 (Events). An *event* is defined as a function $E : \mathcal{S}_r \to \mathcal{M}$, i.e., $E \overset{\text{def}}{=} [m_1/n_1, \ldots, m_k/n_k]$, with $k \geq 1$. Given events E_1 and E_2, we say that E_1 is *included* in E_2 (written $E_1 \sqsubseteq_{\mathsf{E}} E_2$) if and only if $\forall n \in Dom(E_1) \cup Dom(E_2) \Rightarrow E_1(n) \sqsubseteq E_2(n)$. The *union* E_1 and E_2 (written $E_1 \sqcup_{\mathsf{E}} E_2$) is defined for all $n \in Dom(E_1) \cup Dom(E_2)$ as $(E_1 \sqcup_{\mathsf{E}} E_2)(n) = E_1(n) \uplus E_2(n)$.

The big-step semantics for ReactiveML is given in Fig. 2.8 and Fig. 2.9. Intuitively, a big-step reduction captures reactions within a single instant, and is of the form $e \xrightarrow[S]{E,b} e'$, where S stands for the smallest signal environment (with respect to $\sqsubseteq_{\mathsf{E}}$ and S^m) containing input, output, and local signals; E is the event made of signals emitted during the reaction; $b \in \{\texttt{tt}, \texttt{ff}\}$ is a boolean value that indicates termination: b is false if e is stuck during that instant and true otherwise. At each instant i, the program reads an input I_i and produces an output O_i. The reaction of an expression obeys three conditions:

(C1) $(I_i \sqcup_{\mathsf{E}} E_i) \sqsubseteq_{\mathsf{E}} S_i^m$ (i.e., S must contain the inputs and emitted signals);

(C2) $O_i \sqsubseteq_{\mathsf{E}} E_i$ (i.e., the output signals are included in the emitted signals);

(C3) $S_i^d \subseteq S_{i+1}^d$ and $S_i^g \subseteq S_{i+1}^g$ (i.e., default values and gathering functions are preserved throughout instants).

$$\lfloor\textsc{L-Par}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,b_1} e_1' \quad e_2 \xrightarrow[S]{E_2,b_2} e_2' \quad b_1 \wedge b_2 = \mathtt{ff}}{\mathtt{let}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}} \mathtt{let}\ x_1 = e_1'\ \mathtt{and}\ x_2 = e_2'\ \mathtt{in}\ e_3}$$

$$\lfloor\textsc{L-Done}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\mathtt{tt}} v_1 \quad e_2 \xrightarrow[S]{E_2,\mathtt{tt}} v_2 \quad e_3\{{}^{v_1,v_2}/_{x_1,x_2}\} \xrightarrow[S]{E_3,b} e_3'}{\mathtt{let}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3,b} e_3'}$$

$$\lfloor\textsc{Run}\rfloor\ \frac{e \xrightarrow[S]{E_1,\mathtt{tt}} \mathtt{process}\ e' \quad e' \xrightarrow[S]{E_2,b} e''}{\mathtt{run}\ e \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} e''} \qquad \lfloor\textsc{Lp-Stu}\rfloor\ \frac{e \xrightarrow[S]{E,\mathtt{ff}} e'}{\mathtt{loop}\ e \xrightarrow[S]{E,\mathtt{ff}} e';\mathtt{loop}\ e} \qquad \lfloor\textsc{Lp-Un}\rfloor\ \frac{e \xrightarrow[S]{E_1,\mathtt{tt}} v \quad \mathtt{loop}\ e \xrightarrow[S]{E_2,b} e'}{\mathtt{loop}\ e \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} e'}$$

$$\lfloor\textsc{Sig-Dec}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\mathtt{tt}} v_1 \quad e_2 \xrightarrow[S]{E_2,\mathtt{tt}} v_2 \quad e_3\{{}^{n}/_{x}\} \xrightarrow[S]{E_3,b} e_3' \quad n \text{ fresh} \quad S(n) = (v_1, v_2, m)}{\mathtt{signal}_{e_2}\ x : e_1\ \mathtt{in}\ e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3,b} e_3'}$$

$$\lfloor\textsc{Emit}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\mathtt{tt}} n \quad e_2 \xrightarrow[S]{E_2,\mathtt{tt}} v}{\mathtt{emit}\ e_1\ e_2 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} [\{v\}/n],\mathtt{tt}} ()} \qquad \lfloor\textsc{Pause}\rfloor\ \frac{}{\mathtt{pause} \xrightarrow[S]{\emptyset,\mathtt{ff}} ()}$$

$$\lfloor\textsc{Sig-P}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\mathtt{tt}} n \quad n \in S \quad e_2 \xrightarrow[S]{E_2,b} e_2'}{\mathtt{present}\ e_1 ?\, e_2 : e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} e_2'} \qquad \lfloor\textsc{Sig-NP}\rfloor\ \frac{e_1 \xrightarrow[S]{E,\mathtt{tt}} n \quad n \notin S}{\mathtt{present}\ e_1 ?\, e_2 : e_3 \xrightarrow[S]{E,\mathtt{ff}} e_3}$$

$$\lfloor\textsc{DU-End}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \quad e_1 \xrightarrow[S]{E_1,\mathtt{tt}} v}{\mathtt{do}\ e_1\ \mathtt{until}\ e_2(x) \to e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{tt}} v} \qquad \lfloor\textsc{DU-P}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \quad n \in S \quad e_1 \xrightarrow[S]{E_1,\mathtt{ff}} e_1'}{\mathtt{do}\ e_1\ \mathtt{until}\ e_2(x) \to e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}} e_3\{{}^{S^v(n)}/_{x}\}}$$

$$\lfloor\textsc{DU-NP}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \quad n \notin S \quad e_1 \xrightarrow[S]{E_1,\mathtt{ff}} e_1'}{\mathtt{do}\ e_1\ \mathtt{until}\ e_2(x) \to e_3 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}} \mathtt{do}\ e_1'\ \mathtt{until}\ n(x) \to e_3}$$

$$\lfloor\textsc{DW-NS}\rfloor\ \frac{e_2 \xrightarrow[S]{E,\mathtt{tt}} n \quad n \notin S}{\mathtt{do}\ e_1\ \mathtt{when}\ e_2 \xrightarrow[S]{E,\mathtt{ff}} \mathtt{do}\ e_1\ \mathtt{when}\ n} \qquad \lfloor\textsc{DW-Int}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \quad n \in S \quad e_1 \xrightarrow[S]{E_1,\mathtt{ff}} e_1'}{\mathtt{do}\ e_1\ \mathtt{when}\ e_2 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}} \mathtt{do}\ e_1'\ \mathtt{when}\ n}$$

$$\lfloor\textsc{DW-End}\rfloor\ \frac{e_2 \xrightarrow[S]{E_2,\mathtt{tt}} n \quad n \in S \quad e_1 \xrightarrow[S]{E_1,\mathtt{tt}} v}{\mathtt{do}\ e_1\ \mathtt{when}\ e_2 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{tt}} v}$$

Figure 2.8: Big-step semantics for ReactiveML expressions (Part 1).

Intuitions on the big-step reduction rules follow. Rules ⌊L-Par⌋ and ⌊L-Done⌋

$$\lfloor\textsc{Val}\rfloor\ \frac{}{v \xrightarrow[S]{\emptyset,\mathtt{tt}} v} \qquad \lfloor\textsc{Pair}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\mathtt{tt}} v_1 \quad e_2 \xrightarrow[S]{E_2,\mathtt{tt}} v_2}{(e_1, e_2) \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{tt}} (v_1, v_2)} \qquad \lfloor\textsc{Recur}\rfloor\ \frac{v\{\mathtt{rec}\ x = v/x\} \xrightarrow[S]{E,\mathtt{tt}} v'}{\mathtt{rec}\ x = v \xrightarrow[S]{E,\mathtt{tt}} v'}$$

$$\lfloor\textsc{Appl}\rfloor\ \frac{e_1 \xrightarrow[S]{E_1,\mathtt{tt}} \lambda x.e_3 \quad e_2 \xrightarrow[S]{E_2,\mathtt{tt}} v' \quad e_3\{v'/x\} \xrightarrow[S]{E_3,\mathtt{tt}} v}{e_1\ e_2 \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3,\mathtt{tt}} v}$$

$$\lfloor\textsc{Case}\rfloor\ \frac{e \xrightarrow[S]{E_1,\mathtt{tt}} c_j \quad j \in I \quad e_j \xrightarrow[S]{E_2,b} e'_j}{\mathtt{match}\ e\ \mathtt{with}\ \{c_i \to e_i\}_{i \in I} \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} e'_j}$$

Figure 2.9: Big-step semantics for ReactiveML expressions (Part 2).

handle let expressions, distinguishing when (a) at least one of the parallel branches has not yet terminated, and (b) both branches have terminated and their resulting values can be used. Rule ⌊Run⌋ ensures that declared processes can only be executed while they are preceded by a run construct. Rules ⌊Lp-Stu⌋ and ⌊Lp-Un⌋ handle loop expressions: the former decrees that a loop will stop executing when the termination boolean of its body becomes ff; the latter executes a loop until Rule ⌊Lp-Stu⌋ is applied. Rule ⌊Sig-Dec⌋ declares a signal by instantiating it with a fresh name in the continuation; its default value and gathering function must be instantaneous expressions. Rule ⌊Emit⌋ governs signal emission. Rule ⌊Pause⌋ suspends the process for an instant. Rules ⌊Sig-P⌋ and ⌊Sig-NP⌋ check for presence of a signal n: when n is currently present, the body e_2 is run in the same instant; otherwise, e_3 is executed in the next instant. Rules ⌊DU-End⌋, ⌊DU-P⌋, and ⌊DU-NP⌋ handle expressions do e_1 until $e_2(x) \to e_3$. Rule ⌊DU-End⌋ says that if e_1 terminates instantaneously, then the whole expression terminates. Rule ⌊DU-P⌋ says that if e_2 reduces to the name of a currently present signal n, then e_3 is executed in the next instant, substituting x with the values gathered in n. Rule ⌊DU-NP⌋ executes e_1 as long as e_2 does not reduce to a currently present signal. Rule ⌊Val⌋ allows values to execute instantaneously. Finally, Rules ⌊Pair⌋, ⌊Recur⌋, ⌊Appl⌋ and ⌊Case⌋ are the usual functional programming rules for pairs, recursion, application and case operators.

To simplify notation, we use $e \longmapsto e'$ to represent $e \xrightarrow[S]{E,b} e'$ whenever E, S, and b are unimportant, or can be clearly derived from the context. Similarly, we write $e \longmapsto^* e'$, whenever there exists $E_1, \ldots, E_n$, $S_1, \ldots, S_n$, and $b_1, \ldots, b_n$, with $n \geq 0$ such that $e \xrightarrow[S_1]{E_1,b_1} e_1 \xrightarrow[S_2]{E_2,b_2} \ldots \xrightarrow[S_n]{E_n,b_n} e'$.

2.4.2 *Equivalences for ReactiveML Expressions*

Before providing some examples of ReactiveML expressions and their execution, we point out that the original formulation in [MP14] does not provide a method to compare processes, either syntactically nor behaviorally. Such tools are required for our results. Therefore, we introduce a structural congruence for ReactiveML expressions: we write $\equiv_{\mathsf{R}}$ to denote the smallest equivalence on expressions that satisfies the following axioms: (i) $e \parallel () \equiv_{\mathsf{R}} e$; (ii) $e_1 \parallel e_2 \equiv_{\mathsf{R}} e_2 \parallel e_1$; (iii) $(e_1 \parallel e_2) \parallel e_3 \equiv_{\mathsf{R}} e_1 \parallel (e_2 \parallel e_3)$.

Using the previous structural congruence as a base, we propose an useful extension of $\equiv_{\mathsf{R}}$ that allows us to consume matching constructs whenever the compared value is already a constants. For example, we would like to equate the following expression:

$$\mathtt{match}\ c_2\ \mathtt{with}\ \{c_1 \to \lambda x.x, c_2 \to \lambda x.\lambda y.(x+y)\}$$

with $\lambda x.\lambda y.(x+y)$. Therefore, we shall extend $\hookrightarrow_{\mathsf{R}}$ to denote the following pre-order:

Definition 2.39 (Equality with Case Normalization). Let $\hookrightarrow_{\mathsf{R}}$ denote the extension of $\equiv_{\mathsf{R}}$ with the axiom $\mathtt{match}\ c_j\ \mathtt{with}\ \{c_i \to P_i\}_{i\in I} \hookrightarrow_{\mathsf{R}} P_j$, where c_j is a constant and $j \in I$.

Another glaring difference between ReactiveML and the calculi introduced so far (π in § 2.2 and lcc in § 2.3) is the fact that ReactiveML does not have a construct similar to the restriction construct. Indeed, in ReactiveML, signal, variable, and recursive declarations disappear during execution.

Thus, considering the fact that the objective of this work is to encode session π-calculus in ReactiveML, it is clear that we must find a way to ensure that the translation of restriction operators is preserved. We achieve the previous goal by extending the pre-order in Def. 2.39 (i.e., $\hookrightarrow_{\mathsf{R}}$) to include all the missing signal declarations in a ReactiveML expression. As an example, consider the following expression:

$$e_1 = \mathtt{signal}\ x\ \mathtt{in}\ (\mathtt{emit}\ x \parallel \mathtt{pause}\,;\mathtt{emit}\ x) \tag{2.1}$$

Observe that $e_1 \xrightarrow[S]{E,\mathtt{ff}} () \parallel \mathtt{emit}\ x$, with $E = [\mathtt{tt}/x]$, and $S = [(\mathtt{tt}, \lambda x.x, \mathtt{tt})/x]$, and that $() \parallel \mathtt{emit}\ x \xrightarrow[S]{E,\mathtt{tt}} () \parallel ()$. Moreover, notice that it also holds that $\mathtt{signal}\ x\ \mathtt{in}\ (() \parallel \mathtt{emit}\ x) \xrightarrow[S]{E,\mathtt{tt}} () \parallel ()$. It becomes then clear that, intuitively, the two ReactiveML expressions above are equivalent up to certain missing declarations.

Considering this, we shall compare expressions up-to their signal declarations. To this end, we define a special context which will contain the missing signal declarations: a so-called *signal declaration context*.

Definition 2.40. Let $\widetilde{x}$ be a set of variables and $D_{\widetilde{x}}$ be a ReactiveML process with a hole '$-$', given by the following grammar:

$$D_{\widetilde{x}} ::= - \ \big|\ \mathtt{signal}\ \widetilde{x}\ \mathtt{in}\ -$$

We call $D_{\widetilde{x}}$ a signal declaration context and we write $D_{\widetilde{x}}[e]$ to denote the replacement of the hole by expression e. Furthermore, we will say that $D_{\widetilde{x}}$ is *empty* whenever $\widetilde{x}$ is the empty sequence. Finally, whenever $D_{\widetilde{x}}$ is empty, we will say that $D_{\widetilde{x}}[e] = e$.

Remark 2.41. Notice that for the syntax of expressions (cf. Def. 2.36) we assumed that the set of signal names is a subset of the set of variables: $\mathcal{S}_r \subseteq \mathcal{V}_r$. Therefore, a signal declaration $\mathtt{signal}\ \widetilde{n}\ \mathtt{in}\ e$ can bind all the signal names $\widetilde{n}$ in e. This is simply done out of convenience to reduce the number of substitutions we need to apply to ReactiveML expressions and does not affect the expressiveness results. To understand what would happen if we relax this condition, observe that Rule $\lfloor$Sig-Dec$\rfloor$ (cf. Fig. 2.8), applied to expression $e = \texttt{signal}\ x\ \texttt{in}\ e$, substitutes x by a fresh signal name n in Q (i.e., $e\{^{n}\!/x\}$). Thus, to obtain the correct signal declaration $\texttt{signal}\ x\ \texttt{in}\ e$ it would be necessary to substitute back some variable x in e (i.e., $(e\{^{n}\!/x\})\{^{x}\!/n\}$).

To compare ReactiveML expressions that include signal declaration contexts, we extend $\hookrightarrow_{\mathsf{R}}$ into the following pre-order:

Definition 2.42 (Equality with Case Normalization and Signal Declaration). We will say that two ReactiveML expressions e_1 and e_2 are equivalent up-to signal declarations, written $e_1 \lesssim e_2$, whenever $e_1 = e_1' \parallel \cdots \parallel e_n'$ with $n \geq 1$ and there exist, possibly empty, signal declaration contexts $D_{\widetilde{x}}, D_{\widetilde{x_1}}, \ldots, D_{\widetilde{x_n}}$ such that:

$$D_{\widetilde{x}}[D_{\widetilde{x_1}}[e_1'] \parallel \cdots \parallel D_{\widetilde{x_n}}[e_n']] \hookrightarrow_{\mathsf{R}} e_2$$

When presenting our translations we will delve deeper on how to apply the pre-orders presented above in our results.

We conclude this section by giving some examples of interesting ReactiveML expressions and their big-step reductions.

Example 2.43. Consider the following expressions:

$$\begin{aligned}
e_1 &= \texttt{signal}\ x\ \texttt{in}\ (\texttt{emit}\ x\ 42 \parallel \texttt{await}\ x(y)\ \texttt{in}\ y)\\
e_2 &= \texttt{do loop}\ (\texttt{emit}\ x\ 42; \texttt{pause})\ \texttt{until}\ w \to e\\
e_3 &= \texttt{signal}\ x, w\ \texttt{in}\ (e_2 \parallel \texttt{pause}; \texttt{do emit}\ w\ \texttt{when}\ x; \texttt{await}\ x(y)\ \texttt{in}\ y)
\end{aligned}$$

In expression e_1, we showcase a synchronization between two expressions: the leftmost sub-expression inside the signal declaration (i.e., $\texttt{emit}\ x\ 42$) is emitting signal x, whereas the rightmost sub-expression is awaiting for signal x to be emitted and retrieve its value, assigning it to variable y. The big-step reduction of this expression is $e_1 \xrightarrow[S]{E,\texttt{ff}} 42$, where $E = [42/x]$, and $S = [(1, \lambda x.x, 1)/x]$ (assuming a default value of 1). An important thing to notice is that the termination boolean is $\texttt{ff}$—the expression does not terminate in the current instant. Indeed, the await construct executes the continuation at the next time instant, whenever the signal is present.

Expression e_2 showcases the fact that signal emissions are asynchronous (i.e., non-blocking). Notice that for this expression we assume that signals x, and w have already been declared. In e_2 we have a loop that is emitting signal x with value 42 exactly once in each instant until signal w is emitted. However, there is no process emitting said signal, hence, this expression will loop indefinitely:

$$e_2 \xrightarrow[S]{E,\texttt{ff}} e_2 \xrightarrow[S]{E,\texttt{ff}} e_2 \xrightarrow[S]{E,\texttt{ff}} \ldots$$

where $E = [42/x]$, and $S = [(1, \lambda x.x, 1)/x, (\texttt{tt}, \lambda x.x, \texttt{tt})/w]$.

In expression e_3, we find two sub-expressions in parallel inside a signal declaration. The leftmost sub-expression (i.e., e_2) behaves as previously shown. Analogously, the rightmost sub-expression is awaiting for signal x, before emitting signal

w. The big-step reductions for e_3 is given below:

$$e_3 \xrightarrow[S]{E_1,\mathtt{ff}} e_2 \parallel \mathtt{do\ emit}\ w\ \mathtt{when}\ x; \mathtt{await}\ x(y)\ \mathtt{in}\ y \xrightarrow[S]{E_2,\mathtt{ff}} e \parallel 42$$

where $E_1 = [42/x]$, $S_1 = [(1, \lambda x.x, 1)/x, (\mathtt{tt}, \lambda x.x, \mathtt{tt})/w]$, and $E_2 = [42/x, \mathtt{tt}/w]$.

△

2.5 Multiparty Session Types (MPSTs)

We summarize the theory of *multiparty asynchronous session types* [HYC08, CDPY15]. Briefly, multiparty session types is a generalization of the binary type discipline presented in § 2.2, used to certify asynchronous communicating systems that allow *multiple parties* to interact. We closely follow the presentation given in [CDPY15]. In § 2.5.1 we introduce the syntax and semantics of MPST. Next, in § 2.5.2, we introduce the type systems. Finally, in § 2.5.6 we present the main guarantees ensured by typed MPST processes.

2.5.1 *Syntax and Semantics*

In MPST, processes are ranged over by $P, Q, \ldots$ and expressions are ranged over by $e, e', \ldots$. Besides processes and expressions, we also introduce *message queues* and *runtime channels*. These new elements will only appear during process execution and are therefore referred to as *runtime syntax*. Some additional naming conventions follow: (1) we use $a, b, \ldots$ to range over *service names*, (2) and $s, s', \ldots$ to range over *session names*; (3) similarly, we use $x, x', \ldots$ to range over *value variables*, (4) $y, z, t, \ldots$ to range over *channel variables*, (5) and $X, Y, \ldots$ to range over *process variables*; (6) finally, we use $l, l', \ldots$ to range over a countably infinite set of labels.

Definition 2.44 (Syntax). The syntax of MPST processes is given by the grammar in Fig. 2.10. Runtime processes appear shaded in the figure.

In the figure, processes $\overline{u}[\mathsf{p}](y).P$ and $u[\mathsf{p}](y).P$ enable the *initiation* (or *establishment*) of a multiparty session through some service u. *Session participants* are ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$ and are identified by successive numbers. As a design choice, we let the participant with the highest number correspond to p in $\overline{u}[\mathsf{p}](y).P$. Coincidentally, participant p gives the total number of participants needed to establish a session. In both session initiation constructs, variable y appears bound and is used as a placeholder for the channel that will be used in the communication. Once the session has been established each channel place holder will be replaced by a *channel with a role*, denoted $s[\mathsf{p}]$. This construct represents the channel of participant p in session s at runtime. Inside a established session processes can communicate in three ways:

(1) *Value exchanging*, governed by primitives $c!\langle \mathsf{p}, e\rangle.P$ and $c?(\mathsf{p}, x).P$, which correspond to the sending of a value and the receiving of a value, respectively. Therefore, $c!\langle \mathsf{p}, e\rangle.P$ represents the sending of a value on channel c to participant p; similarly $c?(\mathsf{p}, x).P$ represents the reception of some value on channel c; variable x is bound and is used as a placeholder for the value to be received.

$$
\begin{array}{lcll}
P, Q & ::= & \overline{u}[\mathsf{p}](y).P & (\textit{Multicast Request}) \\
& \mid & u[\mathsf{p}](y).P & (\textit{Accept}) \\
& \mid & c!\langle \mathsf{p}, e\rangle.P & (\textit{Value Sending}) \\
& \mid & c?(\mathsf{p}, x).P & (\textit{Value Reception}) \\
& \mid & c!\langle\langle \mathsf{p}, c\rangle\rangle.P & (\textit{Channel Delegation}) \\
& \mid & c?((\mathsf{p}, y)).P & (\textit{Channel Reception}) \\
& \mid & \oplus c\langle \mathsf{p}, l\rangle.P & (\text{Selection}) \\
& \mid & \& c(\mathsf{p}, \{l_i : P_i\}_{i\in I}) & (\textit{Branching}) \\
& \mid & e?\,(P):(Q).P & (\textit{Conditional}) \\
& \mid & P \mid Q & (\textit{Parallel}) \\
& \mid & \mathbf{0} & (\textit{Inaction}) \\
& \mid & (\boldsymbol{\nu} a)P & (\textit{Service Hiding}) \\
& \mid & \mathsf{Def}\ D\ \mathsf{in}\ P & (\textit{Recursion}) \\
& \mid & X\langle e, c\rangle & (\textit{Process Call}) \\
& \mid & (\boldsymbol{\nu} s)P & (\textit{Session Hiding}) \\
& \mid & s : h & (\textit{Message Queue}) \\
\\
D & ::= & X(x, y) & (\textit{Declarations})
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{E} & ::= & [-] \mid P \mid (\boldsymbol{\nu} a)\mathcal{E} & (\textit{Context}) \\
& \mid & (\boldsymbol{\nu} s)\mathcal{E} \mid \mathsf{Def}\ D\ \mathsf{in}\ \mathcal{E} & \\
& \mid & \mathcal{E} \mid \mathcal{E} & \\
u & ::= & x \mid a & (\textit{Identifier}) \\
v & ::= & \mathtt{tt} \mid \mathtt{ff} \mid a & (\textit{Value}) \\
e & ::= & v \mid x & (\textit{Expression}) \\
& \mid & e\ \mathsf{and}\ e \mid \mathsf{not}\ e & \\
c & ::= & y \mid s[\mathsf{p}] & (\textit{Channel}) \\
m & ::= & (\mathsf{q}, \mathsf{p}, v) & (\textit{Message in} \\
& \mid & (\mathsf{q}, \mathsf{p}, s[\mathsf{p}']) & \textit{Transit}) \\
& \mid & (\mathsf{q}, \mathsf{p}, l) & \\
h & ::= & h \cdot m \mid \emptyset & (\textit{Queue})
\end{array}
$$

Figure 2.10: Syntax for MPST processes, declarations, (evaluation) contexts, identifiers, values, expressions, channels, and messages. Shaded constructs only appear at *runtime*.

(2) *Session delegation*, governed by constructs $c!\langle\langle \mathsf{p}, c\rangle\rangle.P$ and $c?((\mathsf{p}, y)).P$. Delegation represents the exchange of a channel and the constructs behave similarly to the output and receive.

(3) *Selection*, governed by constructs $\oplus c\langle \mathsf{p}, l\rangle.P$ and $\& c(\mathsf{p}, \{l : P_i\}_{i\in I})$ and which represents an interaction which allows the selection construct to pick one of the choices offered by $\& c(\mathsf{p}, \{l_i : P_i\}_{i\in I})$. The labels $l_i \in I$ are assumed to be pairwise distinct.

In the sequel, we will consider *output actions* to correspond to value sending, channel delegation or label selection; moreover, we will say that an *output process* is a process whose first action is an output action. Dually, we let *input actions* correspond to value reception, session reception or label branching; then, an *input process* is a process whose first action is an input action. We also use the term *communication action* to refer to either an output and input actions.

Evaluation contexts, denoted by $\mathcal{E}$, correspond to processes with holes (written $[-]$), which can be filled in by replacing them with a process. We write $\mathcal{E}[P]$ to denote the substitution of $[-]$ by process P.

Message queues are used to model asynchronous communications in which the message order is preserved and sending is non-blocking. Messages contained inside queues are characterized by the object they send; hence, a *value message* $(\mathsf{q}, \mathsf{p}, v)$ represents a message from q intended for p which contains value v. *Label messages* are interpreted in the in a similar way. Finally, a *channel message* $(\mathsf{q}, \mathsf{p}, s[\mathsf{p}'])$ indicates that q delegates to p the role if p' on session s (i.e., $s[\mathsf{p}']$). Empty queues are represented by $\emptyset$. The construct $s : h$ denotes the queue h of session s.

Remark 2.45 (*Runtime Syntax*). The constructs that appear shaded in Fig. 2.10 are exclusively generated by the operational semantics of the calculus (cf. Fig. 2.11). Moreover, whenever a process does not contain message queues, we will call it *pure*.

We find many binders in Fig. 2.10: as mentioned above, constructs for request and accept bind channel variables; moreover, value receptions bind value variables, channel receptions bind channel variables, declarations bind value and channel variables, recursions bind process variables, and hidings bind service and session names. Also notice that the session hiding $(\boldsymbol{\nu} s)P$ also binds all free occurrences of $s[\mathsf{p}]$ and queue s in P. Finally, a process P is *closed* if its free names (written $\mathsf{fn}(P)$) only contains service names.

The operational semantics of MPST is given as a reduction relation $\longrightarrow$ over processes. As usual, the semantics is given up to structural congruence, which we define below. In the sequel, we write $\mathsf{fpv}(P)$ to denote the free process variables in P, and $\mathsf{dpv}(D)$ to denote the set of process variable declared in D. Moreover, we assume that free names are extended to declarations as expected. Throughout the section we also assume that $r ::= a \mid s$ and $\zeta ::= v \mid s[\mathsf{p}] \mid l$.

Definition 2.46 (Structural Congruence). The structural congruence relation on M-PST processes, written $\equiv$, is given by the smallest relation generated by α-conversion and the following rules:

$$(\textsc{Str}{:}1)\ P \mid \mathbf{0} \equiv P \quad (\textsc{Str}{:}2)\ P \mid Q \equiv Q \mid P \quad (\textsc{Str}{:}3)\ (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\textsc{Str}{:}4)\ \mathsf{Def}\ D'\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0} \quad (\textsc{Str}{:}5)\ (\boldsymbol{\nu} r)(\boldsymbol{\nu} r')P \equiv (\boldsymbol{\nu} r')(\boldsymbol{\nu} r)(P \mid Q)$$

$$(\textsc{Str}{:}6)\ (\boldsymbol{\nu} a)0 \equiv \mathbf{0} \quad (\textsc{Str}{:}7)\ (\boldsymbol{\nu} a)0 \equiv (\boldsymbol{\nu} s)(s : \emptyset \equiv \mathbf{0})$$

$$(\textsc{Str}{:}8)\ \frac{r \notin \mathsf{fn}(Q)}{(\boldsymbol{\nu} r)P \mid Q \equiv (\boldsymbol{\nu} r)(P \mid Q)} \qquad (\textsc{Str}{:}9)\ \frac{r \notin \mathsf{fn}(D)}{\mathsf{Def}\ D\ \mathsf{in}\ (\boldsymbol{\nu} r)P \equiv (\boldsymbol{\nu} r)\mathsf{Def}\ D\ \mathsf{in}\ P}$$

$$(\textsc{Str}{:}10)\ \frac{\mathsf{dpv}(D) \cap \mathsf{fpv}(Q) = \emptyset}{\mathsf{Def}\ D\ \mathsf{in}\ (\boldsymbol{\nu} r)P \mid Q \equiv (\boldsymbol{\nu} r)\mathsf{Def}\ D\ \mathsf{in}\ P \mid Q}$$

$$(\textsc{Str}{:}11)\ \frac{(\mathsf{dpv}(D) \cup \mathsf{fpv}(D)) \cap \mathsf{dpv}(D') = \mathsf{dpv}(D) \cap (\mathsf{dpv}(D' \cup \mathsf{fpv}(D')))}{\mathsf{Def}\ D\ \mathsf{in}\ \mathsf{Def}\ D'\ \mathsf{in}\ P = \mathsf{Def}\ D'\ \mathsf{in}\ \mathsf{Def}\ D\ \mathsf{in}\ P}$$

$$(\textsc{Str}{:}12)\ \frac{\mathsf{p} \neq \mathsf{p}' \vee \mathsf{q} \neq \mathsf{q}'}{s : h \cdot (\mathsf{q}, \mathsf{p}, \zeta) \cdot (\mathsf{q}', \mathsf{p}', \zeta') \cdot h' \equiv s : h \cdot (\mathsf{q}', \mathsf{p}', \zeta') \cdot (\mathsf{q}, \mathsf{p}, \zeta) \cdot h'}$$

Above, the first eleven rules are standard for the π-calculus [Mil99]. In contrast, the last rule (Rule (Str:12)) is unique to MPST: it allows to rearrange messages in a queue whenever the senders or receivers are not the same. The rules that generate the reduction relation on MPST processes are given in Fig. 2.11. We also use the shorthands $\longrightarrow^*$ and $\longrightarrow^k$ to denote the reflexive-transitive closure of $\longrightarrow$ and a reduction sequence of $k \geq 0$ steps, respectively. Intuitively, Rule $\lfloor\textsc{Init}\rfloor$ describes the session establishment phase: n participants synchronize over name a, where the last participant, distinguished by the bar over its service name, specifies n. During the

$$\lfloor\textsc{Init}\rfloor \; \begin{array}{l} a[1](y).P_1\{^{s[1]}/_{y}\} \mid \ldots \mid a[n-1](y).P_{n-1} \mid \overline{a}[n](y).P_n \\ \longrightarrow P_1 \mid \ldots \mid P_{n-1}\{^{s[n-1]}/_{y}\} \mid P_n\{^{s[n]}/_{y}\} \mid s : \emptyset \end{array}$$

$$\lfloor\textsc{Send}\rfloor \; \frac{e \downarrow v}{s[\mathsf{p}]!\langle \mathsf{q}, e\rangle.P \mid s : h \longrightarrow P \mid s : h \cdot (\mathsf{p}, \mathsf{q}, v)}$$

$$\lfloor\textsc{Rcv}\rfloor \; s[\mathsf{p}]?(\mathsf{q}, x).P \mid s : (\mathsf{q}, \mathsf{p}, v) \cdot h \longrightarrow P\{^{v}/_{x}\} \mid s : h$$

$$\lfloor\textsc{Deleg}\rfloor \; s[\mathsf{p}]!\langle\langle \mathsf{q}, s'[\mathsf{p}']\rangle\rangle.P \mid s : h \longrightarrow P \mid s : h \cdot (\mathsf{p}, \mathsf{q}, s'[\mathsf{p}'])$$

$$\lfloor\textsc{SRcv}\rfloor \; s[\mathsf{p}]?((\mathsf{q}, y)).P \mid s : (\mathsf{q}, \mathsf{p}, s'[\mathsf{p}']) \cdot h \longrightarrow P\{^{s'[\mathsf{p}']}/_{y}\} \mid s : h$$

$$\lfloor\textsc{Sel}\rfloor \; \oplus s[\mathsf{p}]\langle \mathsf{q}, l\rangle.P \mid s : h \longrightarrow P \mid s : h \cdot (\mathsf{p}, \mathsf{q}, l)$$

$$\lfloor\textsc{Bra}\rfloor \; \frac{j \in I}{\& s[\mathsf{p}](\mathsf{q}, \{l_i : P_i\}_{i \in I}) \mid s : (\mathsf{q}, \mathsf{p}, l_j) \cdot h \longrightarrow P_j \mid s : h}$$

$$\lfloor\textsc{ProcCall}\rfloor \; \frac{e \downarrow v}{\mathsf{Def}\ X(x, y) = P\ \mathsf{in}\ X\langle e, s[\mathsf{p}]\rangle \mid Q \longrightarrow \mathsf{Def}\ X(x, y) = P\ \mathsf{in}\ P\{^{v,\, s[\mathsf{p}]}/_{x,\, y}\} \mid Q}$$

$$\lfloor\textsc{IfF}\rfloor \; \frac{e \downarrow \mathtt{tt}}{e?\,(P)\!:\!(Q) \longrightarrow P} \qquad \lfloor\textsc{IfF}\rfloor \; \frac{e \downarrow \mathtt{ff}}{e?\,(P)\!:\!(Q) \longrightarrow Q} \qquad \lfloor\textsc{Ctxt}\rfloor \; \frac{P \longrightarrow P'}{\mathcal{E}[P] \longrightarrow \mathcal{E}[P']} \qquad \lfloor\textsc{Str}\rfloor \; \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Figure 2.11: Reduction rules for MPST.

establishment, private session name s and the queue associated with s (which is initially empty), are shared among participants. Then, the variable y is replaced in each participant with the corresponding channel with role $s[\mathsf{p}]$. Once a session has been established, Rules $\lfloor$Send$\rfloor$, $\lfloor$Deleg$\rfloor$, and $\lfloor$Sel$\rfloor$ are used to enqueue values, channels, and labels, respectively, into the queue associated with session s. In Rule $\lfloor$Send$\rfloor$, $e \downarrow v$ denotes the evaluation of expression e to value v. Dually, Rules $\lfloor$Rcv$\rfloor$, $\lfloor$SRcv$\rfloor$, and $\lfloor$Branc$\rfloor$ are used to dequeue these values and use them in their continuations. All the other rules are as expected.

2.5.2 *Global Types and Session Types*

We now introduce the type system of MPST. With it, we can check the communication safety of processes.

In MPST, *global types* describe the overall communication protocol by providing a global view of the interactions between participants. On the other hand, *local* or *session types* describe the individual contributions of each participant to the protocol. They are obtained by *projecting* the global types and therefore, they correspond to the types of pure processes. The syntax of types is given in Fig. 2.12.

The basic types are *sorts*, ranged over by $S, S', \ldots$; they are associated to values or *closed* global types G (i.e., global types without free type variables). Similarly, *exchange types*, ranged over by U, U', consist of sorts and *closed* session types T (i.e.,

$$
\begin{array}{llcll}
\text{(Sorts)} & S & ::= & \mathtt{bool} \mid \ldots \mid \mathsf{G} & \\
\text{(Exchange Types)} & U & ::= & S \mid \mathsf{T} & \\
 & & & & \\
\text{(Global types)} & G & ::= & \mathsf{p} \to \mathsf{q} : \langle S \rangle.G & (\textit{Value Exchange}) \\
 & & \mid & \mathsf{p} \to \mathsf{q} : \langle \mathsf{T} \rangle.G & (\textit{Channel Exchange}) \\
 & & \mid & \mathsf{p} \to \mathsf{q} : \langle \{l_i : G_i\}_{i \in I} \rangle & (\textit{Branching}) \\
 & & \mid & \mathbf{t} & (\textit{Recursive Variable}) \\
 & & \mid & \mu\mathbf{t}.G \mid \mathsf{end} & (\textit{Recursion/End}) \\
 & & & & \\
\text{(Local Types)} & T & ::= & !\langle \mathsf{p}, S \rangle.T & (\textit{Send Value}) \\
 & & \mid & !\langle \mathsf{p}, \mathsf{T} \rangle.T & (\textit{Send Channel}) \\
 & & \mid & ?(\mathsf{p}, U).T & (\textit{Receive}) \\
 & & \mid & \oplus\langle \mathsf{p}, \{l_i : T_i\}_{i \in I} \rangle & (\textit{Selection}) \\
 & & \mid & \&(\mathsf{p}, \{l_i : T_i\}_{i \in I}) & (\textit{Branching}) \\
 & & \mid & \mathbf{t} & (\textit{Recursive Variable}) \\
 & & \mid & \mu\mathbf{t}.T \mid \mathsf{end} & (\textit{Recursion/End})
\end{array}
$$

Figure 2.12: Syntax of MPST types.

local types without variables).

Intuitively, the global type $\mathsf{p} \to \mathsf{q} : \langle S \rangle.G$ says that participant p is sending a value of sort S to participant q, provided that $\mathsf{p} \neq \mathsf{q}$; then, the interactions in G follow. Similarly, global type $\mathsf{p} \to \mathsf{q} : \langle \mathsf{T} \rangle.G$ denotes the delegation of a channel of type T to participant q, then continuing as G. Whenever it is not important, we write $\mathsf{p} \to \mathsf{q} : \langle U \rangle.G$ to refer to both $\mathsf{p} \to \mathsf{q} : \langle S \rangle.G$ and $\mathsf{p} \to \mathsf{q} : \langle \mathsf{T} \rangle.G$.

The global type $\mathsf{p} \to \mathsf{q} : \langle \{l_i : G_i\}_{i \in I} \rangle$ represents an interaction where participant p sends one of the labels l_i to participant q. If l_j is sent, then the interactions in G_j are executed next. Type $\mu\mathbf{t}.G$ represents a recursive types, where recursive variables are ranged over by $\mathbf{t}, \mathbf{t}', \ldots$ and where these variables are assumed to always appear under some prefix. Once again, we take an equi-recursive view of types and let $\mu\mathbf{t}.G$ be equal to its unfolding $\mu\mathbf{t}.G\{^{\mu\mathbf{t}.G}/\mathbf{t}\}$ because they represent the same regular infinite trees [Pie02]. Finally, type end represents the terminated session.

As hinted before, session types represent the input and output actions performed by each individual participant. Thus, the send types $!\langle \mathsf{p}, S \rangle.T$ and $!\langle \mathsf{p}, \mathsf{T} \rangle.T$ represent the sending of a value of sort S to participant p and the delegation of a channel of type T to participant p, respectively. In both cases, the communication then follows as T. The selection type $\oplus\langle \mathsf{p}, \{l_i : T_i\}_{i \in I} \rangle$ represents the transmission of a label l_i by participant p, chosen from the set indexing I. The type will continue with the communication described in T_i. The receive and branching types are complementary of send and selection types, respectively. Finally, recursive types and the type end are handled as in global types.

The relation between global types and local types is given by a *projection* function on global types. Intuitively, the projection function extracts the individual contributions of a participant to the overall protocol to construct the local type. The definition of projection is given below.

Definition 2.47 (Projection). The projection of a global type G onto the local type of participant q, written $G \upharpoonright \mathsf{q}$, is defined inductively on G:

$$(\mathsf{p} \to \mathsf{p}' : \langle U \rangle.G') \upharpoonright \mathsf{q} = \begin{cases} !\langle \mathsf{p}', U \rangle.(G' \upharpoonright \mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p} \\ ?(\mathsf{p}, U).(G' \upharpoonright \mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p}' \\ G' \upharpoonright \mathsf{q} & \text{otherwise} \end{cases}$$

$$(\mathsf{p} \to \mathsf{p}' : \langle \{l_i : G_i\}_{i \in I} \rangle.G') \upharpoonright \mathsf{q} = \begin{cases} \oplus\langle \mathsf{p}', \{l_i : T_i\}_{i \in I} \rangle.(G' \upharpoonright \mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p} \\ \&(\mathsf{p}, \{l_i : G_i \upharpoonright \mathsf{q}\}_{i \in I}).(G' \upharpoonright \mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p}' \\ & \text{where } i_0 \in I \\ G_{i_0} \upharpoonright \mathsf{q} & \text{if } \mathsf{q} \neq \mathsf{p}, \mathsf{q} \neq \mathsf{p}' \text{ and} \\ & G_i \upharpoonright \mathsf{q} = G_j \upharpoonright \mathsf{q} \\ & \forall i, j \in I \end{cases}$$

$$(\mu \mathbf{t}.G) \upharpoonright \mathsf{q} = \begin{cases} \mu \mathbf{t}.G \upharpoonright \mathsf{q} & \text{if } G \upharpoonright \mathsf{q} \neq \mathbf{t} \\ \mathsf{end} & \text{otherwise} \end{cases} \qquad \mathbf{t} \upharpoonright \mathsf{q} = \mathbf{t} \quad \mathsf{end} \upharpoonright \mathsf{q} = \mathsf{end}$$

In the sequel, we assume that all global types are *well-formed*, meaning that $G \upharpoonright \mathsf{q}$ is defined for all participants q that occur in G.

2.5.3 *Typing Pure Processes*

The typing rules are split in three categories: typing rules for pure processes, typing rules for message queues, and typing rules for processes (runtime and pure). We first present the rules for typing pure processes. The typing judgments for expressions and pure processes are:

$$\Gamma \vdash e : S \qquad \Gamma \vdash P \triangleright \Delta$$

where Γ and Δ are defined as mappings given by the grammar below:

$$\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, a : \mathsf{G} \mid \Gamma, X : S\,T \qquad \Delta ::= \emptyset \mid \Delta, c : T$$

Intuitively, Γ corresponds to the so-called *standard environment* which associates variables to sorts, service names to closed global types, and process variables to *pairs* of sort types and session types. On the other hand, Δ is called the *session environment* which associates channels to session types. As usual, we assume $dom(\Gamma)$ (resp. $dom(\Delta)$) represents the domain of Γ (resp, Δ). Notice that the notation $\Gamma, x : S$ is only well-defined whenever $x \notin dom(\Gamma)$. This convention extends to $a : \mathsf{G}$, $X : S\,T$, and $c : T$. The latter implies that Δ, Δ' is only well-defined if $\Delta \cap \Delta' = \emptyset$. Then, the typing rules are given in Fig. 2.13. In the figure, $\mathsf{mp}(\mathsf{G})$ corresponds to the participant with the highest number assigned, also called the *maximum participant* in G. Some intuitions on the typing rules follow:

- Rules (Name), (Bool), and (And) are as expected.
- Rules (MCast) and (MAcc) deal with request and accept constructs, respectively. In the former rule, Γ must contain $u : \mathsf{G}$ and p must be the maximum participant of G. Then, the projection of G onto p is added to the session environment, assigned to y, to type continuation P. The latter rule is similar, except for the fact that the number assigned to p is required to be less than $\mathsf{mp}(\mathsf{G})$.

$$\text{(\textsc{Name})}\ \Gamma, u : S \vdash u : S \qquad \text{(\textsc{Bool})}\ \Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathsf{bool} \qquad \text{(\textsc{And})}\ \frac{\Gamma \vdash e_i : \mathsf{bool} \quad i \in \{1,2\}}{\Gamma \vdash e_1 \ \mathsf{and}\ e_2 : \mathsf{bool}}$$

$$\text{(\textsc{MCast})}\ \frac{\Gamma \vdash u : \mathsf{G} \quad \Gamma \vdash P \triangleright \Delta, y : \mathsf{G} \upharpoonright \mathsf{p} \quad \mathsf{p} = \mathsf{mp}(\mathsf{G})}{\Gamma \vdash \overline{u}[\mathsf{p}](y).P \triangleright \Delta}$$

$$\text{(\textsc{MAcc})}\ \frac{\Gamma \vdash u : \mathsf{G} \quad \Gamma \vdash P \triangleright \Delta, y : \mathsf{G} \upharpoonright \mathsf{p} \quad \mathsf{p} < \mathsf{mp}(\mathsf{G})}{\Gamma \vdash u[p](y).P \triangleright \Delta}$$

$$\text{(\textsc{Send})}\ \frac{\Gamma \vdash e : S \quad \Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle \mathsf{p}, e\rangle.P \triangleright \Delta, c : !\langle \mathsf{p}, S\rangle.T} \qquad \text{(\textsc{Rcv})}\ \frac{\Gamma, x : S \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c?(\mathsf{p}, x).P \triangleright \Delta, c : ?(\mathsf{p}, S).T}$$

$$\text{(\textsc{Deleg})}\ \frac{\Gamma \vdash P \triangleright \Delta, c : T}{\Gamma \vdash c!\langle\!\langle \mathsf{p}, c' \rangle\!\rangle.P \triangleright \Delta, c : !\langle \mathsf{p}, \mathsf{T}\rangle.T, c' : \mathsf{T}} \qquad \text{(\textsc{SRcv})}\ \frac{\Gamma \vdash P \triangleright \Delta, y : \mathsf{T}, c : T}{\Gamma \vdash c?(\!(\mathsf{p}, x)\!).P \triangleright \Delta, c : ?(\mathsf{p}, \mathsf{T}).T}$$

$$\text{(\textsc{Sel})}\ \frac{\Gamma \vdash P \triangleright \Delta, c : T_j \quad j \in I}{\Gamma \vdash \oplus c\langle \mathsf{p}, l_j\rangle.P \triangleright \Delta, c : \oplus\langle \mathsf{p}, \{l_i : T_i\}_{i \in I}\rangle}$$

$$\text{(\textsc{Branch})}\ \frac{\Gamma \vdash P_i \triangleright \Delta, c : T_i \quad \forall i \in I}{\Gamma \vdash \& c(\mathsf{p}, \{l_i : P_i\}_{i \in I}) \triangleright \Delta, c : \&(\mathsf{p}, \{l_i : T_i\}_{i \in I})}$$

$$\text{(\textsc{Par})}\ \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'} \qquad \text{(\textsc{If})}\ \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash e?\,(P) : (Q) \triangleright \Delta}$$

$$\text{(\textsc{End})}\ \frac{\Delta \ \mathsf{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \text{(\textsc{NRes})}\ \frac{\Gamma, a : \mathsf{G} \vdash P \triangleright \Delta}{\Gamma \vdash (\boldsymbol{\nu} a)P \triangleright \Delta}$$

$$\text{(\textsc{Var})}\ \frac{\Gamma \vdash e : S \quad \Delta \ \mathsf{end\ only}}{\Gamma, X : S\, T \vdash X\langle e, c\rangle \triangleright \Delta, c : T} \qquad \text{(\textsc{Def})}\ \frac{\Gamma, x : S\, \mathbf{t}, x : S \vdash P \triangleright y : T \quad \Gamma, X : S\, \mu\mathbf{t}.T \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{Def}\ X(x, y) = P\ \mathsf{in}\ Q \triangleright \Delta}$$

Figure 2.13: Typing rules for pure MPST processes.

- Rules (Send) and (Deleg) are reminiscing of Rules (T:Out) and (T:In) in Fig. 2.3 for π. The only difference is that, in MPST, there are no qualifiers, and we split the rules: Rule (Send) deals exclusively with the sending of values and Rule (Deleg) with the sending of channels.
- Rules (Rcv), (SRcv) are complementary to rules (Send) and (Deleg).
- Rules (Sel) and (Branch) deal with selection and branching, and are analogous to rules (T:Sel) and (T:Bra) in Fig. 2.3.
- Rules (Par), (If), (End) are defined as expected. In Rule (Par), the session environments Δ and Δ' must be disjoint. Also, in Rule (End), we require ses-

$$
\begin{array}{llcll}
\text{Message Types} & M & ::= & !\langle \mathsf{p}, U\rangle & (\textit{Message Send}) \\
 & & \mid & \oplus\langle \mathsf{p}, l\rangle & (\textit{Message Selection}) \\
 & & \mid & M; M & (\textit{Message Sequence}) \\
\\
\text{Generalized Types} & \tau & ::= & T & (\textit{Session}) \\
 & & \mid & M & (\textit{Message}) \\
 & & \mid & M; T & (\textit{Continuation})
\end{array}
$$

Figure 2.14: Syntax of message and generalized MPST types.

sion environments in which every channel is associated with type `end` (i.e., the condition "Δ end only").

- In Rule (NRes), the service name a must be added to the standard environment Γ assigned to a closed global type G.
- Rules (Var) and (Def) deal with recursion. Notice that the recursion rule presented in here forces process variables to be associated with μ-types—for details, see [CDPY15].

2.5.4 Typing Queues

We now present the second set of typing rules for MPST mentioned in § 2.5.3: rules for typing message queues. We start by defining the syntax of *message* and *generalized* types. The former correspond to the types assigned to message queues, while the latter is defined as the "composition" of message types and local types (see below). The syntax of message and generalized types is given in Fig. 2.14.

Intuitively, the message send type $!\langle \mathsf{p}, U\rangle$ indicates that a queue contains an element of type U that is to be communicated to participant p. The message selection type, on the other hand, signals the communication of a label l to participant p. Finally, type $M; M'$ denotes the sequencing of message types. Notice that we assume ";" to be associative. For example, a message $(\mathsf{q}, \mathsf{p}, \mathtt{tt})$ has type $!\langle \mathsf{p}, \mathtt{bool}\rangle$.

Generalized types, as mentioned above, represent a composition of session types and message types. Intuitively, a generalized type can be either a session type, a message type, or a message type followed by a session type. In a sense, it can be said that generalized types are used to reconstruct the overall type that governs the behavior of one participant. Thus, type $M; T$ represents that the continuation of the type M associated to some queue is T, which must be associated to a pure process.

Example 2.48 (Generalized Types). To explain the subtle difference between '.' and ';' let us consider the following two types:

$$!\langle \mathsf{p}, \mathtt{bool}\rangle.?(\mathsf{p}, \mathtt{bool}).\mathtt{end} \qquad !\langle \mathsf{p}, \mathtt{bool}\rangle; ?(\mathsf{p}, \mathtt{bool}).\mathtt{end}$$

where the leftmost type is a session type and the rightmost is a generalized type. Intuitively, the first type indicates that participant p is *ready* to send a message of type `bool` to participant q. In contrast, the second one represents a type where participant p has already send a message with type `bool`, but this message has not yet been received by participant q, and thus, the message is still in the queue. △

$$\text{(QINIT)}\quad \Gamma \vdash_{\emptyset} s : \emptyset \triangleright \emptyset \qquad \text{(QSEND)}\quad \frac{\Gamma \vdash_{\{s\}} s : h \triangleright \Delta \quad \Gamma \vdash v : S}{\Gamma \vdash_{\{s\}} s : h(\mathsf{p}, \mathsf{q}, v) \triangleright \Delta; \{s[\mathsf{q}] : !\langle \mathsf{p}, S\rangle\}}$$

$$\text{(QDEL)}\quad \frac{\Gamma \vdash_{\{s\}} s : h \triangleright \Delta}{\Gamma \vdash_{\{s\}} s : h(\mathsf{p}, \mathsf{q}, s'[\mathsf{q}']) \triangleright (\Delta; \{s[\mathsf{q}] : !\langle \mathsf{p}, \mathsf{T}\rangle\}), s'[\mathsf{p}'] : \mathsf{T}}$$

$$\text{(QSEL)}\quad \frac{\Gamma \vdash_{\{s\}} s : h \triangleright \Delta}{\Gamma \vdash_{\{s\}} s : h(\mathsf{p}, \mathsf{q}, l) \triangleright \Delta; \{s[\mathsf{q}] : \oplus\langle \mathsf{p}, l\rangle\}}$$

Figure 2.15: Typing rules for single queues in MPST.

We can now explain the rules for typing *single* queues in Fig. 2.15. Typing judgments for single queues are similar to judgments for pure processes. The main differences are that '$\vdash$' is decorated with the singleton $\{s\}$, where s is the session name of the current queue, and that session environments now map channels to message types.

Intuitively, Rule (QINIT) assigns the empty session environment to the empty queue. Then, Rules (QSEND), (QDELEG), and (QSEL) check that the queue of session s contains the corresponding output types, before checking the rest of the queue. The sequencing operator ';' is extended to environments by letting:

$$\Delta; \{s[\mathsf{p}] : M\} = \begin{cases} \Delta', s[\mathsf{q}] : M'; M & \text{if } \Delta = \Delta', s[\mathsf{q}] : M' \\ \Delta, s[\mathsf{q}] : M & \text{otherwise} \end{cases}$$

2.5.5 Typing Runtime Processes

We now present the full type system of MPST. Hence, we consider pure processes in parallel with queues; therefore, we should use generalized types in session environments, and to add new rules.

We start by defining an equivalence, denoted $\approx$, between message types that will allow us to take into account the structural congruence between queues (cf. Def. 2.46, Rule (STR:11)). The equivalence is induced by the rule below:

$$M; \natural\langle \mathsf{p}, Z\rangle; \natural\langle \mathsf{p}', Z\rangle; M' \approx M'; \natural\langle \mathsf{p}', Z\rangle; \natural\langle \mathsf{p}, Z\rangle; M$$

where $\natural \in \{!, \oplus\}$ and $Z \in \{U, l\}$. In the rest of this section we consider message types modulo $\approx$. We then extend the equivalence to generalized types:

$$M \approx M' \implies M; \tau \approx M'; \tau$$

We then say that two session environments Δ and Δ' are equivalent (written $\Delta \approx \Delta'$) whenever:

$$c : \tau \in \Delta \wedge \tau \neq \mathsf{end} \implies c : \tau' \in \Delta' \wedge \tau \approx \tau'$$

Above, type end is ignored because the type system allows weakening for this kind of types [CDPY15].

We must now define the composition of two session environments which may contain the same channel with role $s[\mathsf{p}]$. Moreover, in one of these environments, $s[\mathsf{p}]$ may be assigned a message type, and in the other a session type. Hence, we define a *partial* composition between generalized types, which allows to "sequentialize" the types:

$$\tau * \tau' = \begin{cases} \tau; \tau' & \text{if } \tau \text{ is a message type} \\ \tau'; \tau & \text{if } \tau' \text{ is a message type} \end{cases}$$

The composition above is partial because is only defined whenever τ or τ' is a message type. We then extend this composition to environments:

$$\Delta * \Delta' = \Delta \setminus dom(\Delta') \cup \Delta' \setminus dom(\Delta) \cup \{c : \tau * \tau' \mid c : \tau \in \Delta \wedge c : \tau' \in \Delta'\}$$

The definition of $*$ induces a commutativity property: $\Delta * \Delta' = \Delta' * \Delta$.

To type processes with queues we introduce the notion of *session environment consistency*. Intuitively, this property ensures that each individual participant in a multiparty session performs its communication actions in a consistent way—in a sense, this property generalizes the guarantees given by duality in π (cf. Def. 2.15).

To define consistency we need two ingredients: the *projection of generalized types* and *duality*, which are given below:

Definition 2.49 (Partial Projections of Generalized Types). The partial projections of generalized types, ranged over by $\mathcal{G}, \mathcal{G}, \ldots$ is given below:

$$(!\langle \mathsf{p}, U\rangle.T) \upharpoonright \mathsf{q} = \begin{cases} !U.T \upharpoonright \mathsf{q} & \text{if } \mathsf{q} = \mathsf{p} \\ T \upharpoonright \mathsf{p} & \text{otherwise} \end{cases} \qquad (?(\mathsf{p}, U).T) \upharpoonright \mathsf{q} = \begin{cases} ?U.T \upharpoonright \mathsf{q} & \text{if } \mathsf{q} = \mathsf{p} \\ T \upharpoonright \mathsf{p} & \text{otherwise} \end{cases}$$

$$(!\langle \mathsf{p}, U\rangle; \tau') \upharpoonright \mathsf{q} = \begin{cases} !U; \tau' \upharpoonright \mathsf{q} & \text{if } \mathsf{q} = \mathsf{p} \\ \tau' \upharpoonright \mathsf{p} & \text{otherwise} \end{cases} \qquad (\oplus\langle \mathsf{p}, l\rangle; \tau') \upharpoonright \mathsf{q} = \begin{cases} \oplus l; \tau' \upharpoonright \mathsf{q} & \text{if } \mathsf{q} = \mathsf{p} \\ \tau' \upharpoonright \mathsf{p} & \text{otherwise} \end{cases}$$

$$(\oplus\langle \mathsf{p}, \{l_i : T_i\}_{i\in I}\rangle) \upharpoonright \mathsf{q} = \begin{cases} \oplus\{l_i : T_i\}_{i\in I} & \text{if } \mathsf{q} = \mathsf{p} \\ T_{i_0} & \text{where } i_0 \in I \text{ if } \mathsf{p} \neq \mathsf{q} \text{ and} \\ & T_i \upharpoonright \mathsf{q} = T_j \upharpoonright \mathsf{q}\ \forall i, j \in I \end{cases}$$

$$(\&(\mathsf{p}, \{l_i : T_i\}_{i\in I})) \upharpoonright \mathsf{q} = \begin{cases} \&\{l_i : T_i\}_{i\in I} & \text{if } \mathsf{q} = \mathsf{p} \\ T_{i_0} & \text{where } i_0 \in I \text{ if } \mathsf{p} \neq \mathsf{q} \text{ and} \\ & T_i \upharpoonright \mathsf{q} = T_j \upharpoonright \mathsf{q}\ \forall i, j \in I \end{cases}$$

$$(\mu\mathbf{t}.T) \upharpoonright \mathsf{q} = \begin{cases} \mu\mathbf{t}.(T) \upharpoonright \mathsf{q} & \text{if } T \upharpoonright \mathsf{q} \neq \mathbf{t} \\ \mathsf{end} & \text{otherwise} \end{cases} \quad \mathbf{t} \upharpoonright q = \mathbf{t} \quad \mathsf{end} \upharpoonright \mathsf{q} = \mathsf{end}$$

Definition 2.50 (Duality). The duality relation on partial projections of generalized types is given below:

$$\mathsf{end} \bowtie \mathsf{end} \quad \mathbf{t} \bowtie \mathbf{t} \quad \mathcal{G} \bowtie \mathcal{G}' \implies \mu\mathbf{t}.\mathcal{G} \bowtie \mu\mathbf{t}.\mathcal{G}'$$

$$\mathcal{G} \bowtie \mathcal{G}' \implies !U.\mathcal{G} \bowtie ?U.\mathcal{G}' \quad \mathcal{G} \bowtie \mathcal{G}' \implies !U; \mathcal{G} \bowtie ?U.\mathcal{G}'$$

$$\forall i \in I\ \mathcal{G}_i \bowtie \mathcal{G}'_i \implies \oplus\{l_i : \mathcal{G}\}_{i\in I} \bowtie \&\{l_i : \mathcal{G}'\}_{i\in I}$$

$$\exists i \in I\ l = l_i \wedge \mathcal{G}_i \bowtie \mathcal{G}'_i \implies \oplus l; \mathcal{G} \bowtie \&\{l_i : \mathcal{G}'\}_{i\in I}$$

$$\text{(GInit)}\ \frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash_{\emptyset} P \triangleright \Delta} \qquad \text{(Equiv)}\ \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta' \quad \Delta \approx \Delta'}{\Gamma \vdash_{\Sigma} P \triangleright \Delta}$$

$$\text{(GPar)}\ \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \Sigma \cap \Sigma' = \emptyset}{\Gamma \vdash_{\Sigma \cup \Sigma'} P \mid Q \triangleright \Delta * \Delta'}$$

$$\text{(GSRes)}\ \frac{\Gamma \vdash_{\Sigma} P \triangleright \Delta \quad \mathsf{Co}(\Delta, s)}{\Gamma \vdash_{\Sigma \setminus s} (\boldsymbol{\nu} s)P \triangleright \Delta \setminus s} \qquad \text{(GNRes)}\ \frac{\Gamma, a : \mathsf{G} \vdash_{\Sigma} P \triangleright \Delta}{\Gamma \vdash_{\Sigma} (\boldsymbol{\nu} a)P \triangleright \Delta}$$

$$\text{(GDef)}\ \frac{\Gamma, x : S\,\mathbf{t}, x : S \vdash_{\Sigma} P \triangleright \{y : T\} \quad \Gamma, X : S\,\mu\mathbf{t}.T \vdash_{\Sigma} Q \triangleright \Delta}{\Gamma \vdash \mathsf{Def}\ X(x, y) = P\ \mathsf{in}\ Q \triangleright \Delta}$$

Figure 2.16: Typing rules for processes with queues in MPST.

Definition 2.51 (Consistency). A session environment Δ is consistent for the session s (written $\mathsf{Co}(\Delta, s)$) if $s[\mathsf{p}] : \tau \in \Delta$ and $s[\mathsf{q}] : \tau' \in \Delta$ imply $\tau \restriction \mathsf{q} \bowtie \tau' \restriction \mathsf{p}$. A session environment is consistent if it is consistent for all sessions which occur in it.

The following proposition guarantees that all the projection of the same global type are dual, and therefore the session environment obtained by projection global types are always consistent.

Proposition 2.52 (Duality of Global Type Projections [CDPY15]). *Let G be a global type and* $\mathsf{p} \neq \mathsf{q}$. *Then* $(G \restriction \mathsf{p}) \restriction \mathsf{q} \bowtie (G \restriction \mathsf{q}) \restriction \mathsf{p}$.

We are now ready to present the rest of the typing rule for MPST. For processes with queues, the typing judgments are of the form:

$$\Gamma \vdash_{\Sigma} P \triangleright \Delta$$

where Σ is now the set of all queues whose session name is in Δ. The rules are then given on Fig. 2.16.

Rule (GInit) states that a pure process types with $\Sigma = \emptyset$, provided that it is typable with the rules Fig. 2.13. Rule $\lfloor$Equiv$\rfloor$ introduces $\approx$ to the type system. Whenever two processes are composed in parallel, we require that each session name is associated with, at most, one queue. Hence, we require $\Sigma \cap \Sigma' = \emptyset$. Rules (GSRes) and (GNRes) deal with the restriction of session names and service names. Notice that session names can only be restricted whenever the environment is consistent for the session that is restricted. Rule $\lfloor$GDef$\rfloor$ remains as (Def).

2.5.6 *Typing Properties*

We now present the main typing property for the type system of MPST: subject reduction. Notice that since session types also represent ensuing communications, the session environments after a reduction may change and therefore, we need to define a reduction relation for session environments (given below). This contrasts with the type system with the subject reduction property in § 2.2.3 (cf. Thm. 2.18). The reason for this difference is due to the fact that the π semantics require communicating endpoints to be closed under restriction (cf. Fig. 2.1), whereas the semantics of MPST does allow communication between processes containing free channels with roles (cf. Fig. 2.11).

Definition 2.53 (Environment Reduction). The reduction relation for session environments, written $\Delta \Rightarrow \Delta'$, is given by the following rules:

- $\{s[\mathsf{p}] : M; !\langle \mathsf{q}, U\rangle.T\} \Rightarrow \{s[\mathsf{p}] : M; !\langle \mathsf{q}, U\rangle; T\}$.
- $\{s[\mathsf{p}] :!\langle \mathsf{q}, U\rangle; \tau, s[\mathsf{q}] : M; ?(\mathsf{p}, U).T\} \Rightarrow \{s[\mathsf{p}] : \tau, s[\mathsf{q}] : M; T\}$.
- $\{s[\mathsf{p}] : \oplus\langle \mathsf{p}, \{l_i : T_i\}_{i\in I}\rangle\} \Rightarrow \{s[\mathsf{p}] : M; \oplus\langle \mathsf{p}, l_j\rangle; T_j\}$ for $j \in I$.
- $\{s[\mathsf{p}] : \oplus\langle \mathsf{q}, l\rangle; \tau, s[\mathsf{q}] : M; \&(\mathsf{p}, \{l_i : T_i\}_{i\in I})\} \Rightarrow \{s[\mathsf{p}]; \tau, s[\mathsf{q}] : M; T_j\}$ if $l = l_j$ and $j \in I$.
- $\Delta, \Delta'' \Rightarrow \Delta', \Delta''$ if $\Delta \Rightarrow \Delta'$.

where M can be missing and message types are considered modulo $\approx$.

Intuitively, the first rule corresponds to putting a message with sender p, receiver q and with content of type U in the queue for session s (hence the session type is "transformed" into a message type). The second rule reads the message in the queue and therefore consumes it. The next two rules are similar, applied to labels; the last rule extends the reduction to larger session environments.

The authors in [CDPY15] observe that not all processes that can reduce are typed with consistent environments. For example, they present the judgment:

$$\Gamma \vdash_\Sigma \underbrace{s[1]?(2,x).s[1]?(2,x').\mathbf{0} \mid s : (2,1,\mathtt{tt})}_{P} \triangleright \underbrace{\begin{array}{l}\{s[1] :!\langle 2, \mathsf{bool}\rangle.!\langle 2, \mathsf{nat}\rangle.\mathsf{end},\\ s[2] :!\langle 1, \mathsf{bool}\rangle\}\end{array}}_{\Delta}$$

where $P \longrightarrow s[1]?(2,x').\mathbf{0} \mid s : \emptyset$ and Δ is not consistent. Given that the proof of subject congruence goes by induction on the length of the process reduction, we must prove that whenever a process is typed with a non-consistent session environment, which is consistent by composing it with another, then its reduction satisfies the same property. We then conclude this section by stating this statement followed by the subject reduction property.

Lemma 2.54 (Main Lemma [CDPY15]). *Let $\Gamma \vdash_\Sigma P \triangleright \Delta$ and $P \longrightarrow Q$ be obtained by any reduction rule different from $\lfloor$Ctxt$\rfloor$, $\lfloor$Str$\rfloor$, and $\Delta * \Delta_0$ be consistent, for some Δ_0. Then there is Δ' such that $\Gamma \vdash_\Sigma Q \triangleright \Delta'$ and $\Delta \Rightarrow \Delta'$ and $\Delta' * \Delta_0$ is consistent.*

Theorem 2.55 (Subject Reduction [CDPY15]). *If $\Gamma \vdash_\Sigma P \triangleright \Delta$ and $P \longrightarrow^* Q$, then $\Gamma \vdash_\Sigma Q \triangleright \Delta'$ for some consistent Δ' such that $\Delta \Rightarrow \Delta'$.*

3 Source and Target Languages

In this chapter we introduce the source languages for our translations, as well as additional target languages. First, we introduce our source languages:

- In § 3.1 we develop three variants of π (cf. § 2.2):

  1. In § 3.1.2 we introduce $\pi^{\lightning}_{\mathsf{OR}}$, a variant of π that disallows output races (see Ex. 2.25) by redefining the predicates $\mathtt{un}(\cdot)$ and $\mathtt{lin}(\cdot)$ for the π type system (cf. § 2.2.2).
  2. Next, in § 3.1.1 we present $\pi^{\lightning}_{\mathsf{R}}$, a variant of π that disallows both output and input races, while allowing infinite behavior given by replication.
  3. Finally, in § 3.1.3 we present an extension of $\pi^{\lightning}_{\mathsf{OR}}$, named π_{E}, that allows session establishment with *localities*.

- In § 3.2 we introduce $\mathsf{a}\pi$, an asynchronous variant of the language presented in [KYHH16].

After presenting the source languages, we present our remaining target languages:

- In § 3.3 we introduce an extension of `lcc` (cf. § 2.3) which allows for abstractions to have private, local information when resolving a query to the store.

- In § 3.4 we then present RMLq, an extension of ReactiveML that uses queues to keep track of the value of variables during execution (cf.).

In § 3.5 we close this chapter by summarizing all source and target languages and our encodability results.

3.1 Variants of the Session Calculus π

3.1.1 *A Session π Calculus without Output Races* ($\pi^{\lightning}_{\mathsf{OR}}$)

In this section we present $\pi^{\lightning}_{\mathsf{OR}}$, a variant of π (cf. § 2.2), induced by a variant of the type system in § 2.2.2. The goal of the new type system is to disallow output races. Intuitively, we say that π process has an output race whenever said process has two or more *output-like* (i.e., sending and branching constructs) prefixes on the same variable in parallel. We first motivate $\pi^{\lightning}_{\mathsf{OR}}$ by using some examples. Next, we introduce our specialized type system and present further examples to intuitively explain the details behind the taken design decisions. Finally, we present all the guarantees that the type system for $\pi^{\lightning}_{\mathsf{OR}}$ ensures.

Motivation

When devising translations for formal languages with different natures such as the ones present in this work, it is often necessary to narrow down the classes of processes considered in the source language, enabling the translation to satisfy stronger correctness criteria. Type systems, such as the one in § 2.2.2, can be used exactly for this purpose. Our first translation, as hinted in § 3.5, is an `lcc` embedding of a session-based π-calculus (cf. Ch. 4). Hence, to obtain a translation that enjoys strong correctness properties it is necessary to reconcile the differences between π and `lcc`. In particular, a prominent feature of π and its type system (cf. § 2.2.2) is its rather flexible support for π processes with infinite behavior, in the form of recursive session types that can be shared among multiple threads [Vas12]. This expressiveness contrasts with the more modest replication present in `lcc` (cf. § 2.3.1).

To smoothen these differences, we shall devise a specialization of the type system in § 2.2.2; this way we can identify a class of π processes that exhibits a tight correspondence with `lcc` processes—a correspondence made precise by the correctness properties of the translation in Ch. 4. At its core, this specialization rules out output races, which can be typed using the system in § 2.2.2 using recursive types. To illustrate the kind of processes we would like the new type system to rule out, let us recall process P_4 in Ex. 2.25:

$$P_4 = (\boldsymbol{\nu}xy)(x\langle\mathtt{tt}\rangle.\mathbf{0} \mid x\langle\mathtt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle\mathtt{tt}\rangle.\mathbf{0})$$

It can be shown that P_4 is typable by assigning the recursive type $*\,!\mathtt{bool}$ to x and its dual to y (cf. Ex. 2.25). Since our objective is ruling our processes such as P_4, it becomes clear that it is necessary to control the sharing of recursive types among the threads in a process.

Disallowing output races is not an stringent restriction on the session calculus. In fact, we can show that $\pi^{\lightning}_{\mathsf{OR}}$ can express an important class of processes with infinite behavior by using unrestricted input types and replication. As we show later, our type system allows processes such as P_2 (cf. Ex. 2.25) below:

$$P_2 = (\boldsymbol{\nu}xy)(\boldsymbol{\nu}wz)(x\langle\mathtt{ff}\rangle.\mathbf{0} \mid *\,y(u).\mathbf{0} \mid *\,y(u').(w\langle\mathtt{tt}\rangle.\mathbf{0} \mid z(u'').\mathbf{0}))$$

which represents multiple replicated servers offering services to a single client. Moreover, it can be argued that it is still possible to execute a *similar* behavior to the one induced by output races, where all the requests (outputs) have been *sequentialized*. This means that we could represent a process with output races such as:

$$P_5 = (\boldsymbol{\nu} x_1 y_1)(x_1\langle v_1\rangle.Q_1 \mid x_1\langle v_2\rangle.Q_2 \mid * y_1(z).Q_3) \tag{3.1}$$

as a process without output races by establishing a specific execution order (i.e, by saying which output interacts first), and creating new private channels to continue the interactions with Q_3:

$$\begin{aligned} P_5' = {} & (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3) \\ & (x_1\langle y_2\rangle.x_1\langle y_3\rangle.(x_2\langle v_1\rangle.Q_1\{^{x_2}/_{x_1}\} \mid x_3\langle v_2\rangle.Q_2\{^{x_3}/_{x_1}\}) \mid * y_1(z).z(z').Q_3) \end{aligned} \tag{3.2}$$

In the rest of this section we introduce our specialized type system. Note that it is necessary to prove again all the typing properties as a way to ensure that $\pi^{\not\,}_{\mathsf{OR}}$ still satisfies the guarantees given in § 2.2.2 for π.

Remark 3.1 (*Syntax and Semantics of* $\pi^{\not\,}_{\mathsf{OR}}$). The syntax and semantics of $\pi^{\not\,}_{\mathsf{OR}}$ will remain the same as the ones presented in § 2.2.1. Similarly, the syntax of types is as in Fig. 2.2. The only changes will appear in the type system—specifically, in the predicates $\mathsf{un}(\cdot)$ and $\mathtt{lin}(\cdot)$, used to define the environment splitting operation (cf. Def. 2.16).

Type System

To disallow output races we focus on the splitting operation for the type system in § 2.2.2. Recalling Def. 2.16, we focus on the types that satisfy $\mathsf{un}(\cdot)$, whose splitting is given by:

$$\frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathsf{un}(T)}{(\Gamma_1, x : T) \circ (\Gamma_2, x : T) = \Gamma, x : T}$$

In a nutshell, we would like for T to be a type that cannot be assigned to an output-like channel. Hence, we would like to completely exclude unrestricted output and selection types (i.e., $q!T.T'$ and $q \oplus \{l_i : T_i\}_{i\in I}$) from being shared among environments—even when they do not occur immediately; e.g., $?T.!T'.\mathsf{end}$. We accomplish this by redefining the predicate $\mathsf{un}(\cdot)$ (cf. Def. 2.14). First, we present an auxiliary predicate, which characterizes these output-like types:

Definition 3.2 (Output-Like Unrestricted Types). We define the predicate $\mathtt{out}(T)$ for types T inductively. We start by defining $\mathtt{out}(p)$ for pre-types:

$$\begin{aligned} \mathtt{out}(!T.U) &\overset{\mathsf{def}}{=} \mathtt{tt} & \mathtt{out}(?T.U) &\overset{\mathsf{def}}{=} \mathtt{out}(U) \\ \mathtt{out}(\oplus\{l_i : T_i\}_{i\in I}) &\overset{\mathsf{def}}{=} \mathtt{tt} & \mathtt{out}(\&\{l_i : T_i\}_{i\in I}) &\overset{\mathsf{def}}{=} \textstyle\bigvee_{i\in I} \mathtt{out}(T_i) \end{aligned}$$

then $\mathtt{out}(T)$ is given below:

$$\mathtt{out}(\mathsf{bool}) \overset{\mathsf{def}}{=} \mathtt{ff} \quad \mathtt{out}(\mathsf{end}) \overset{\mathsf{def}}{=} \mathtt{ff} \quad \mathtt{out}(\mathsf{a}) \overset{\mathsf{def}}{=} \mathtt{ff}$$
$$\mathtt{out}(qp) \overset{\mathsf{def}}{=} \mathtt{out}(p) \quad \mathtt{out}(\mu\mathsf{a}.T) \overset{\mathsf{def}}{=} \mathtt{out}(T)$$

Having defined output-like unrestricted types, we proceed to show the new predicates for the type system of $\pi^{\not\,}_{\mathsf{OR}}$ by replacing Def. 2.14 with the following definition:

$$\text{(T:WkNil)}\ \frac{\Gamma \vdash \mathbf{0} \quad \mathsf{uno}(T)}{\Gamma, x : T \vdash \mathbf{0}}$$

Figure 3.1: Additional weakening rule for the $\pi^{\lightning}_{\mathsf{OR}}$ type system.

Definition 3.3 (Refined Predicates for $\pi^{\lightning}_{\mathsf{OR}}$). Let T be a session type (cf. Fig. 2.2). We define $q(T)$ and $q(\Gamma)$ for each qualifier $q \in \{\mathsf{un}, \mathtt{lin}\}$:

- $\mathtt{lin}(T)$ if and only if true.
- $\mathsf{un}(T)$ if and only if $(T = \mathtt{bool}) \vee (T = \mathsf{end}) \vee (T = \mathsf{un}\, p \wedge \neg \mathtt{out}(p))$.
- $q(\Gamma)$ if and only if $x : T \in \Gamma$ implies $q(T)$.

Above, predicate $\mathtt{lin}(T)$ remains as presented in § 2.2.2. The main change can be found in predicate $\mathsf{un}(T)$, that is modified to avoid sharing output-like types by requiring that pre-types qualified with un do not satisfy $\mathtt{out}(\cdot)$. Furthermore, we will find it also useful to collect all pre-types qualified with un that satisfy $\mathtt{out}(\cdot)$, hence we add an additional predicate:

- $\mathsf{uno}(T)$ if and only if $T = \mathsf{un}\, p$ and $\mathtt{out}(p)$.

The type system of $\pi^{\lightning}_{\mathsf{OR}}$ is then obtained by using all the rules in Fig. 2.3 extended with the rules in Fig. 3.1. In the figure, Rule (T:WkNil) is used to allow the weakening of unrestricted output-like types whenever the inactive process is being typed. To understand why the new rule is necessary, we use a simple example. Below, we unfold the notation for recursive types: $*!T = \mu \mathsf{a}.!T.\mathsf{a}$.

Example 3.4. Consider the recursive type $T = \mu \mathsf{a}.\, \mathsf{un}!\mathtt{bool}.\mathsf{a}$, and the processes $P_1 = x\langle \mathtt{tt} \rangle.x\langle \mathtt{ff} \rangle.\mathbf{0}$, and $P_2 = x\langle \mathtt{tt} \rangle.\mathbf{0} \mid x\langle \mathtt{ff} \rangle.\mathbf{0}$. With the new type system, we would like to use environment $x : T$ to type P_1. Moreover, we would like the same environment to rule out P_2 (notice that P_2 would be typable in π with the same environment). Let us then analyze the typing derivation trees. For P_1 we have:

$$\text{(T:Out)}\ \frac{\text{(T:Bool)}\ \dfrac{\mathsf{un}(\emptyset)}{\emptyset \vdash \mathtt{tt}} \quad \text{(T:Var)}\ \dfrac{\mathsf{un}(\emptyset)}{x : T \vdash x : q!\mathtt{bool}.U} \quad \dfrac{D}{\emptyset + x : U \vdash x\langle \mathtt{ff} \rangle.\mathbf{0}}}{\emptyset \circ x : \mu \mathsf{a}.\, \mathsf{un}!\mathtt{bool}.\mathsf{a} \circ \emptyset \vdash x\langle \mathtt{tt} \rangle.x\langle \mathtt{ff} \rangle.\mathbf{0}}$$

The first important observation for the typing derivation above is that $\mathsf{un}(T)$ is *false* and $\mathtt{out}(T)$ is true. This means that when splitting the environment (i.e., $\Gamma_1 = \emptyset$, $\Gamma_2 = \mu \mathsf{a}.\, \mathsf{un}!\mathtt{bool}.\mathsf{a}$, and $\Gamma_3 = \emptyset$), as required by Rule (T:Out), only one of the branches will contain $x : T$—in this case, the middle branch. Next, we must obtain U; since we use equi-recursive types, we have that $\mu \mathsf{a}.\, \mathsf{un}!\mathtt{bool}.\mathsf{a} =!\mathtt{bool}.\mu \mathsf{a}.\, \mathsf{un}!\mathtt{bool}.\mathsf{a}$. Therefore, $U = T$, which means that the rightmost branch becomes $x : T \vdash x\langle \mathtt{ff} \rangle.\mathbf{0}$. Below we give the derivation sub-tree D:

$$\text{(T:Out)}\ \frac{\text{(T:Bool)}\ \dfrac{\mathsf{un}(\emptyset)}{\emptyset \vdash \mathtt{ff}} \quad \text{(T:Var)}\ \dfrac{\mathsf{un}(\emptyset)}{x : T \vdash x : q!\mathtt{bool}.T} \quad \text{(T:WkNil)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(\emptyset)}{\emptyset \vdash \mathbf{0}}}{\emptyset + x : T \vdash \mathbf{0}}}{\emptyset \circ x : \mu \mathsf{a}.\, \mathsf{un}!\mathtt{bool}.\mathsf{a} \circ \emptyset \vdash x\langle \mathtt{ff} \rangle.\mathbf{0}}$$

The sub-tree above is obtained by applying Rule (T:Out). Notice that before concluding the rightmost branch, we are left with judgment $x : T \vdash \mathbf{0}$. Then, since $\mathsf{un}(T)$ is false, it is necessary to apply Rule (T:WkNil), followed by (T:Nil) to finish the derivation.

Intuitively, the need for Rule (T:WkNil) follows from the fact that some typing judgments may require the weakening of output-like unrestricted types before concluding the derivation.

We now show the typing derivation for P_2, and precisely detail how the redefined $\mathsf{un}(\cdot)$ predicate makes rules out the undesired output races:

$$\text{(T:Par)}\ \frac{\Gamma_1 \vdash x\langle \texttt{tt}\rangle.\mathbf{0} \quad \Gamma_2 \vdash x\langle \texttt{ff}\rangle.\mathbf{0}}{x : \mu \mathsf{a}.\,\mathsf{un}!\texttt{bool}.\mathsf{a} \vdash x\langle\texttt{tt}\rangle.\mathbf{0} \mid x\langle\texttt{ff}\rangle.\mathbf{0}}$$

Notice that since $\mathsf{un}(T)$ is not true, the splitting operation cannot have a copy of $x : T$ in both Γ_i, $i \in \{1, 2\}$. Therefore, one of the $\Gamma_i = \emptyset$, which implies that one of the branches will not be able to type. △

We further exhibit the intuitions behind the $\pi^{\lightning}_{\mathsf{OR}}$ type system. First, we recall P_4 from Ex. 2.25 and show that it is not well-typed in the new type system as it contains output races. Second, we recall P_3 from Ex. 2.25 and show its derivation tree. In the examples below we omit the application of the rules for boolean values and variables, focusing only on processes:

Example 3.5. Consider process P_4 below (cf. Ex. 2.25):

$$P_4 = (\boldsymbol{\nu} xy)(x\langle\texttt{tt}\rangle.\mathbf{0} \mid x\langle\texttt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle\texttt{tt}\rangle.\mathbf{0})$$

P_4 is well-typed (cf. Ex. 2.25), but only if we are able to share the type of x among two different environments for typing two different threads. This type is $T = *!\texttt{bool} = \mathsf{un}\,\mu\mathsf{a}.!\texttt{bool}.\mathsf{a}$. By following the predicates defined above, it can be seen that $\mathsf{un}(T)$ does not hold, as $\mathtt{out}(T)$ is true. Hence, the splitting rule does not allow to share T among environments, and thus, some of the branches of the typing derivation will not finish:

$$\text{(T:Res)}\ \cfrac{\text{(T:Par)}\ \cfrac{\text{Some of the branches in here fail as } x : T \text{ cannot be shared.}}{x : *!\texttt{bool}, y : *?\texttt{bool}.t \vdash x\langle\texttt{tt}\rangle.\mathbf{0} \mid x\langle\texttt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle\texttt{tt}\rangle.\mathbf{0}}}{\vdash (\boldsymbol{\nu} xy)(x\langle\texttt{tt}\rangle.\mathbf{0} \mid x\langle\texttt{tt}\rangle.y(u).\mathbf{0} \mid y(u').x\langle\texttt{tt}\rangle.\mathbf{0})}$$

△

Example 3.6. Let us consider process $P_3 = (\boldsymbol{\nu} wz)(\boldsymbol{\nu} xy)(x\langle z\rangle.w(u').\mathbf{0} \mid *\,y(u).u\langle\texttt{tt}\rangle.\mathbf{0})$ (cf. Ex. 2.25), whose typing derivation tree is given below:

$$\text{(T:Res)}{\times}2\ \cfrac{\text{(T:Par)}\ \cfrac{D \qquad \text{(T:RIn)}\ \cfrac{\text{(T:Out)}\ \cfrac{\text{(T:Nil)}\ \cfrac{\mathsf{un}(y : *?(!\texttt{bool}), u : \mathsf{end})}{y : *?(!\texttt{bool}), u : \mathsf{end} \vdash \mathbf{0}}}{y : *?(!\texttt{bool}), u : !\texttt{bool} \vdash u\langle\texttt{tt}\rangle.\mathbf{0}}}{y : *?(!\texttt{bool}) \vdash *\,y(u).u\langle\texttt{tt}\rangle.\mathbf{0}}}{\begin{array}{l} x : *!(!\texttt{bool}), y : *?(!\texttt{bool}), \\ w : ?\texttt{bool}, z : !\texttt{bool} \end{array} \vdash x\langle z\rangle.w(u').\mathbf{0} \mid *\,y(u).u\langle\texttt{tt}\rangle.\mathbf{0}}}{\vdash (\boldsymbol{\nu} wz)(\boldsymbol{\nu} xy)(x\langle z\rangle.w(u').\mathbf{0} \mid *\,y(u).u\langle\texttt{tt}\rangle.\mathbf{0})}$$

where the derivation sub-tree D corresponds to:

$$\text{(T:Out)}\ \dfrac{\text{(T:In)}\ \dfrac{\text{(T:WkNil)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(y : *?(!\mathsf{bool}), u' : \mathsf{bool}, w : \mathsf{end})}{y : *?(!\mathsf{bool}), u' : \mathsf{bool}, w : \mathsf{end} \vdash \mathbf{0}}}{x : *!(!\mathsf{bool}), y : *?(!\mathsf{bool}), u' : \mathsf{bool}, w : \mathsf{end} \vdash \mathbf{0}}}{y : *?(!\mathsf{bool}), w : ?\mathsf{bool} + x : *!(!\mathsf{bool}) \vdash w(u').\mathbf{0}}}{x : *!(!\mathsf{bool}), y : *?(!\mathsf{bool}), w : ?T, z : !\mathsf{bool} \vdash x\langle z\rangle.w(u').\mathbf{0}}$$

△

Properties of Typing

Due to the above modifications, it becomes necessary to prove modified statements that ensure typing correctness. We start by proving some auxiliary results. First, we show some basic properties of the typing predicates defined above. The most salient one is that $\mathsf{un}(T)$ does not necessarily implies $\mathsf{un}(\overline{T})$. This occurs because although input-like types can satisfy $\mathsf{un}(\cdot)$, their dual types necessarily satisfy $\mathtt{out}(\cdot)$, and therefore, $\mathsf{un}(\overline{T})$ does not hold. Below, the requirement "$\overline{T}$ is defined" allows us to rule out unnecessary types such as $\mathtt{bool}$ or $\mu\mathsf{a}.\mathtt{bool}$, whose duality is undefined.

Lemma 3.7 (Basic Properties for Types). *Let T be a $\pi_{\mathsf{OR}}^{\sharp}$ type such that $\overline{T}$ is defined. Then, all of the following holds:*

1. *If $\mathsf{un}(T)$ then one of the following holds:*
   (*a*) *If $T = \mathsf{end}$ or $T = \mathsf{a}$ then $\neg\mathtt{out}(\overline{T})$ holds.*
   (*b*) *If $T = \mu\mathsf{a}.T$ or $T = qp$ then $\mathtt{out}(\overline{T})$ holds.*
2. *If $\mathtt{lin}(T)$ then $\mathtt{lin}(\overline{T})$ holds.*

Proof. By induction on the structure of T. For details see App. A.1. □

Next, we show some properties regarding typing environments. Let $\mathcal{U}(\Gamma)$ denote the typing environment containing exactly the entries $x : T \in \Gamma$ such that $\mathsf{un}(T)$. Note that Lem. 3.8(2) below has been modified with respect to the lemma presented in [Vas12] to account for output-like unrestricted types.

Lemma 3.8 (Properties of Typing Environments). *Let $\Gamma = \Gamma_1 \circ \Gamma_2$. Then all of the following hold:*

1. *$\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$.*
2. *Suppose that $x : qp \in \Gamma \wedge (q = \mathtt{lin} \vee (q = \mathsf{un} \wedge \mathtt{out}(p) = \mathtt{tt}))$. Then, either $x : qp \in \Gamma_1$ and $x \notin dom(\Gamma_2)$ or $x : qp \in \Gamma_2$ and $x \notin dom(\Gamma_1)$.*
3. *$\Gamma = \Gamma_2 \circ \Gamma_1$.*
4. *If $\Gamma_1 = \Delta_1 \circ \Delta_2$ then $\Delta = \Delta_2 \circ \Gamma_2$ and $\Gamma = \Delta_1 \circ \Delta$.*

Proof. Every item is proven by induction on the structure of Γ and by using the definition of splitting and predicates $\mathsf{un}(\cdot)$ and $\mathtt{lin}(\cdot)$. For details see App. A.1 □

The *unrestricted weakening* lemma proven below allows us to introduce new variables paired with unrestricted types in a typing environment. This result is useful whenever variables that do not appear free in the process being typed must be added in the environment.

Lemma 3.9 (Unrestricted Weakening). *If* $\Gamma \vdash P$ *and* $\mathsf{un}(T)$ *then* $\Gamma, x : T \vdash P$.

Proof. By induction on the derivation $\Gamma \vdash P$. There are ten cases. The base case is given by Rule (T:Nil), which follows from inversion on the rule, the definition of $\mathsf{un}(\cdot)$, and by applying Lem. 3.8(5). For the inductive step it is first necessary to prove a similar result for the two rules dealing with variables (Rules (T:Bool) and (T:Var)). This follows by a simple case analysis and inversion on the corresponding rule (while applying Lem. 3.8(5)). Using the result for variables, the inductive step for the statement above follows by inversion, applying the IH to the hypotheses obtained, and by reapplying the necessary rule. □

Strengthening allows us to remove linear variables that do not appear free in P from Γ. The statement below also ensures that variables with types that satisfy $\mathtt{lin}(\cdot)$ are consumed. Similarly to Lem. 3.8, Lem. 3.10(2) below has been adapted from [Vas12] to account for output-like unrestricted types.

Lemma 3.10 (Strengthening). *Let* $\Gamma \vdash P$ *and* $x \notin \mathsf{fv}_\pi(P)$. *Then the following holds:*

1. *If* $x : qp \wedge (q = \mathtt{lin} \vee (q = \mathtt{un} \wedge \mathtt{out}(p) = \mathtt{tt}))$ *then* $x : qp \notin \Gamma$.
2. *If* $\Gamma = \Gamma', x : T$ *and* $\mathsf{un}(T)$ *then* $\Gamma' \vdash P$.

Proof. Each item proceeds by induction on the typing derivation $\Gamma \vdash P$ and there are ten cases. First, we must establish a similar result for values, which follows by a case analysis on the applied rule. For Item (1) notice that the Rule (T:Nil) follows immediately, because predicate $\mathsf{un}(\cdot)$ rules out the possibility of $x : qp \in \Gamma$; the inductive cases proceed by applying the IH. In Item 2; the base case proceeds by considering that $x \notin \mathsf{fv}_\pi(\mathbf{0})$, and $\mathsf{un}(\Gamma')$ still holds. All the inductive cases proceed by applying the IH. □

We now state our subject congruence property. Notice that although this statement remains unchanged regarding to the one in Lem. 2.17, the required proof will differ noticeably. In particular, we find differences in the cases where Rule (T:Par) is applied, as it is necessary to split the environment.

Lemma 3.11 (Subject Congruence). *If* $\Gamma \vdash P$ *and* $P \equiv_S Q$ *then* $\Gamma \vdash Q$.

Proof. By a case analysis on the typing derivation for each member of each axiom for $\equiv_S$. For details see App. A.1.

□

The *substitution lemma* below ensures that typing is preserved by substitutions. This lemma will be important for proving *subject reduction* (cf. Thm. 3.13).

Lemma 3.12 (Substitution). *If* $\Gamma_1 \vdash v : T$ *and* $\Gamma_2, x : T \vdash P$ *then* $\Gamma \vdash P\{^v\!/x\}$, *with* $\Gamma = \Gamma_1 \circ \Gamma_2$.

Proof. By induction on the structure of P. For details see App. A.1. □

Using the substitution lemma above we can prove subject reduction to ensure that typing is preserved by the reduction relation given in Fig. 2.1.

Theorem 3.13 (Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \longrightarrow^* Q$ *then* $\Gamma \vdash Q$.

Proof. By induction on the reduction $P \longrightarrow^* Q$ with a case analysis on the last applied rule. See App. A.1 for details.

□

Finally, we prove type safety. Notice that disallowing output races entails a revised class of *well-formed* processes, different from the one in Def. 2.20:

Definition 3.14 (Well-Formed Process). A $\pi^{\lightning}_{\mathsf{OR}}$ process P_0 is *well-formed* if for each of its structural congruent processes $P_0 \equiv_{\mathsf{S}} (\boldsymbol{\nu} x_1 y_1) \dots (\boldsymbol{\nu} x_n y_n)(P \mid Q \mid R)$, with $n \geq 0$, the following conditions hold:

1. If $P \equiv_{\mathsf{S}} v?\,(P'){:}(P'')$ then $v = \mathtt{tt}$ or $v = \mathtt{ff}$.
2. If P and Q are prefixed at the same variable, then they are of the same input-like nature (inputs, replicated inputs, or branchings).
3. If P is prefixed at x_i and Q is prefixed at y_i, $1 \leq i \leq n$, then $P \mid Q$ is a redex.

Two important remarks about the above definition follow. First, since the syntax of $\pi^{\lightning}_{\mathsf{OR}}$ is the same as the one of π, it is not necessary to redefine redexes and pre-redexes, as they remain as in Def. 2.19. Second, the main difference of this definition with respect to Def. 2.20 is in the second item: while Def. 2.20 allows processes prefixed on the same variable to be of the "same nature" (input-like or output-like), our definition only allows these variables to be of input-like nature. Below we state type safety; we lift the notation for programs (cf. Not. 2.21) to $\pi^{\lightning}_{\mathsf{OR}}$ as it is.

Theorem 3.15 (Type Safety). *If* $\vdash P$ *then* P *is well-formed.*

Proof. By contradiction. For details see App. A.1.

□

The following corollary shows that well-formedness is preserved under reduction. It follows from Thm. 3.13 and Thm. 3.15.

Corollary 3.16. *If* $\vdash P$ *and* $P \longrightarrow^* Q$ *then* Q *is well-formed with respect to Def. 3.14.*

3.1.2 A Session π-Calculus without Races ($\pi^{\lightning}_{\mathsf{R}}$)

We introduce a variant of π (cf. § 2.2) that disallows races (i.e., whenever a variable x is shared among two or more threads). This calculus, called $\pi^{\lightning}_{\mathsf{R}}$, can be understood as a further specialization of $\pi^{\lightning}_{\mathsf{OR}}$ (cf. § 3.1.1). We first motivate the type system required for $\pi^{\lightning}_{\mathsf{R}}$. Next, we present the type system for $\pi^{\lightning}_{\mathsf{R}}$. Then, we show the properties ensured by typing in $\pi^{\lightning}_{\mathsf{R}}$. Finally, we introduce a big-step semantics for $\pi^{\lightning}_{\mathsf{R}}$, which will be useful for proving the encoding in Ch. 7 valid (cf. Def. 2.3) and proof its semantic correspondence (cf. Def. 2.9).

Motivation

There are some differences between the expressive power of well-typed π programs and our target languages which need to be reconciled to ensure encoding correctness. The calculus $\pi_{\mathsf{R}}^{\lightning}$ is the source language for the encoding presented in Ch. 7 and ReactiveML, the target. The main difference between these two languages arise from their semantics: in ReactiveML signals are emitted and detected by processes that react to them; this behavior makes signals ideal to simulate channel endpoints in π. However, signals and channels behave in very different ways: while ReactiveML signals are asynchronous and they are broadcast to every process, messages sent across channels in π are synchronous and point-to-point. Indeed, signals can: (1) be emitted even if there is no process to detect them and (2) be simultaneously detected by multiple processes in the same time instant. In contrast, channels in π: (1) cannot send messages to multiple receptors at the same time and (2) can only send a message whenever a receptor is ready to receive it.

Therefore, it is convenient to focus on a sub-class of well-typed π processes with a particular type of infinite behavior. This sub-class corresponds to processes without shared variables among parallel subprocesses. As we mentioned before, well-typed processes bring a flexibility that goes beyond our purposes, as we want ReactiveML to capture the essence of session types: linear behavior. Nonetheless, it is important to mention that our translation allows for forms of infinite behavior. In particular, we want to allow unrestricted behavior that is akin to servers establishing a linear session with its corresponding client, as the example below:

$$P_6 = (\boldsymbol{\nu} xy)((\boldsymbol{\nu} wz)(x\langle w\rangle.z(y_1).Q_1) \mid *\, y(y_2).y_2\langle v\rangle.Q_2) \tag{3.3}$$

Process P_6 represents a client that interacts with a replicated server: during the first synchronization a new session is established. It is then expected that the new session proceeds linearly. Notice that the replicated server remains available to further clients after reduction. This behavior is reminiscent of the replicated servers in [CP10]. As it will be shown later, we can also model processes like P_5' in (3.2), provided that $y \notin \mathsf{fv}_\pi(Q)$. We now present two processes we would like to rule out, as an example:

$$\begin{aligned} P_7 &= (\boldsymbol{\nu} xy)(x\langle v\rangle.Q_1 \mid *\, y(z).Q_2 \mid x\langle v'\rangle.Q_3) \\ P_8 &= (\boldsymbol{\nu} xy)(x\langle v\rangle.Q_1 \mid *\, y(z).Q_2 \mid *\, y(z').Q_3) \end{aligned} \tag{3.4}$$

Above, processes P_7 and P_8 present two forms of unrestricted behavior allowed in typed π [Vas12]. In P_7, a pair of clients are able to interact with a replicated server. In P_8, a client can interact with one of the two replicated servers.

Below we explain the difficulties that arise by using ReactiveML to encode the behavior of P_7 and P_8, due to the semantics of ReactiveML:

Scenario for P_7: The translations of two clients run in parallel. Given the asynchronous nature of the ReactiveML semantics, the signal representing x, say s_x, is emitted twice in the same instant. Two issues arise: (1) unless the order of the received values is fixed, the determinism required in SRP languages breaks and (2) one of the messages would be lost, as s_x can only be detected once in a time instant.

$$\text{(T:WkNil)}\ \frac{\Gamma \vdash \mathbf{0}}{\Gamma, x : \mathsf{un}\, p \vdash \mathbf{0}} \qquad \text{(T:RIn)}\ \frac{\mathsf{un}(\Gamma) \quad x : T \vdash x : \mathsf{un}?T'.U \quad \Gamma, y : T \vdash P}{\Gamma, x : T \vdash * x(y).P}$$

Figure 3.2: Additional weakening rule and new replication rule for $\pi^{\natural}_{\mathsf{R}}$.

Scenario for P_8: The translations of two servers interact with the translation of a client. Thus, due to the broadcast nature of signals in ReactiveML, the emission of s_x is detected by the translation of both servers at the same time, thus activating the translations of Q_2 and Q_3 at the same time: a clearly incorrect behavior.

Remark 3.17. As with $\pi^{\natural}_{\mathsf{OR}}$, the syntax and semantics of $\pi^{\natural}_{\mathsf{R}}$ remain unchanged. Thus, the syntax for $\pi^{\natural}_{\mathsf{R}}$ is given by Def. 2.10, and its semantics by Fig. 2.1.

Type System

We modify the $\mathsf{un}(\cdot)$ predicate (cf. Def. 2.14) to disallow every pre-type qualified with un, and add a rule for dealing with these types when finishing the typing derivation. By modifying $\mathsf{un}(\cdot)$ (cf. Def. 2.14) we disallow the sharing of variables by changing the splitting operation in Def. 2.16. Below, we present the refined predicates for $\pi^{\natural}_{\mathsf{R}}$:

Definition 3.18 (Refined Predicates for $\pi^{\natural}_{\mathsf{R}}$). Let T be a session type (cf. Fig. 2.2). We define $q(T)$ and $q(\Gamma)$ for each qualifier $q \in \{\mathsf{un}, \mathtt{lin}\}$:

- $\mathsf{un}(T)$ if and only if $T = \mathsf{bool}$ or $T = \mathsf{end}$.
- $\mathtt{lin}(T)$ if and only if true.
- $q(\Gamma)$ if and only if $x : T \in \Gamma$ implies $q(T)$.

The only difference with respect to Def. 2.14 is the fact that now pre-types qualified with un do not satisfy $\mathsf{un}(\cdot)$. This redefinition has an effect on the splitting operation. In particular, the splitting operation now forbids to copy variables assigned pre-types qualified with un between split environments.

In tandem with the previous changes, we modify the rules in Fig. 2.3 by adding Rule (T:WkNil) and replacing Rule (T:RIn) with the rule in Fig. 3.2. Intuitively, the former allows us to remove unnecessary recursive types that are product of the changes in predicate $\mathsf{un}(\cdot)$. Then, the latter is changed to disallow any form of unrestricted behavior in the continuation P—observe that the only types that satisfy $\mathsf{un}(\cdot)$ are end and bool.

To understand the changes in the typing rules, we use a series of examples that show how typing derivations work in $\pi^{\natural}_{\mathsf{R}}$:

Example 3.19. In $\pi^{\natural}_{\mathsf{R}}$ we would likes to typecheck process $P_1 = * y(z_1).z_1(z_2).\mathbf{0}$, while discarding both $P_2 = * y(z_1).\mathbf{0} \mid * y(z_1).\mathbf{0}$ and $P_3 = * y(z_1).y(z_2).\mathbf{0}$ with recursive types. We first show how we can type P_1. Assume that $T = \mu\mathsf{a}.\, \mathsf{un}?(?\mathtt{bool}.\mathsf{end}).\mathsf{a}$:

$$\text{(T:RIn)}\ \dfrac{\mathsf{un}(\emptyset) \qquad \text{(T:Var)}\ \dfrac{\mathsf{un}(\emptyset)}{y : T \vdash y : \mathsf{un}\,.?(?\mathtt{bool}.\mathsf{end}).T} \qquad \dfrac{D}{z_1 :?\mathtt{bool}.\mathsf{end} \vdash z_1(z_2).\mathbf{0}}}{y : \mu\mathsf{a}.\, \mathsf{un}?(?\mathtt{bool}.\mathsf{end}).\mathsf{a} \vdash *\,y(z_1).z_1(z_2).\mathbf{0}}$$

The derivation above can be broken down as follows. First notice that Rule (T:RIn) (cf. Fig. 3.2) requires the environment (without $\Delta(y)$) to satisfy $\mathsf{un}(\cdot)$. Next, the rule requires that y is of an appropriate type; in this derivation sub-tree we can apply Rule (T:Var), due to our equi-recursive view of types. Finally, the rule requires that using z_1, we can type the continuation $z_1(z_2).\mathbf{0}$. We do not expand sub-tree D, which can be finished by applying Rule (T:In) and (T:Nil).

Notice that process P_2 does not typecheck because variable y cannot be shared among environments when applying Rule $\lfloor$T:Par$\rfloor$; therefore, one of the branches will not type-check.

Finally, we analyze process P_3. This process should not be typable because it generates races after reduction:

$$(\boldsymbol{\nu}xy)(x\langle \mathtt{tt}\rangle.\mathbf{0} \mid *\,y(z_1).y(z_2).\mathbf{0}) \longrightarrow (\boldsymbol{\nu}xy)(\mathbf{0} \mid y(z_2).\mathbf{0} \mid *\,y(z_1).y(z_2)) = P_3'$$

where P_3' has a race on y. The typing derivation fails because for a reason similar to P_2: it is not possible to share y and thus, the continuation cannot be typed. △

We conclude this section by showing how to type a process that delegates a linear channel, using replication and recursive types.

Example 3.20. Consider process P_3 below (cf. Ex. 2.25):

$$P_3 = (\boldsymbol{\nu}wz)(\boldsymbol{\nu}xy)(x\langle z\rangle.w(u').\mathbf{0} \mid *\,y(u).u\langle \mathtt{tt}\rangle.\mathbf{0})$$

whose typing derivation tree is given below. We only show the rule applications for processes, omitting the ones for variables and constants:

$$\text{(T:Res)}\times 2\ \dfrac{\text{(T:Par)}\ \dfrac{D \qquad \text{(T:RIn)}\ \dfrac{\text{(T:Out)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(u : \mathsf{end})}{u : \mathsf{end} \vdash \mathbf{0}}}{u :!\mathtt{bool} \vdash u\langle \mathtt{tt}\rangle.\mathbf{0}}}{y : *\,?(!\mathtt{bool}) \vdash *\,y(u).u\langle \mathtt{tt}\rangle.\mathbf{0}}}{\begin{array}{l} x : *\,!(!\mathtt{bool}), y : *\,?(!\mathtt{bool}), \\ w :?\mathtt{bool}, z :!\mathtt{bool} \end{array} \vdash x\langle z\rangle.w(u').\mathbf{0} \mid *\,y(u).u\langle \mathtt{tt}\rangle.\mathbf{0}}}{\vdash (\boldsymbol{\nu}wz)(\boldsymbol{\nu}xy)(x\langle z\rangle.w(u').\mathbf{0} \mid *\,y(u).u\langle \mathtt{tt}\rangle.\mathbf{0})}$$

where the derivation sub-tree D corresponds to:

$$\text{(T:Out)}\ \dfrac{\text{(T:In)}\ \dfrac{\text{(T:WkNil)}\ \dfrac{\text{(T:Nil)}\ \dfrac{\mathsf{un}(u' : \mathsf{bool}, w : \mathsf{end})}{u' : \mathsf{bool}, w : \mathsf{end} \vdash \mathbf{0}}}{x : *\,!(!\mathtt{bool}), u' : \mathsf{bool}, w : \mathsf{end} \vdash \mathbf{0}}}{w :?\mathtt{bool} + x : *\,!(!\mathtt{bool}) \vdash w(u').\mathbf{0}}}{x : *\,!(!\mathtt{bool}), w :?\mathtt{bool}, z :!\mathtt{bool} \vdash x\langle z\rangle.w(u').\mathbf{0}}$$

△

Typing Properties

We now prove some useful properties for the type system of $\pi_{\mathsf{R}}^{\lightning}$. As with $\pi_{\mathsf{OR}}^{\lightning}$, we first show some properties regarding typing environments. We shall reuse $\mathcal{U}(\Gamma)$ to denote the typing environment containing exactly the entries $x : T \in \Gamma$ such that $\mathsf{un}(T)$. Note that Lem. 3.21(2) below accounts for the changes of predicate $\mathsf{un}(\cdot)$ in Def. 3.18.

Lemma 3.21 (Properties of Typing Environments). *Let $\Gamma = \Gamma_1 \circ \Gamma_2$. Then all of the following hold:*

1. $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$.
2. *Suppose that $x : qp \in \Gamma$. Then, either $x : qp \in \Gamma_1$ and $x \notin dom(\Gamma_2)$ or $x : qp \in \Gamma_2$ and $x \notin dom(\Gamma_1)$.*
3. $\Gamma = \Gamma_2 \circ \Gamma_1$.
4. *If $\Gamma_1 = \Delta_1 \circ \Delta_2$ then $\Delta = \Delta_2 \circ \Gamma_2$ and $\Gamma = \Delta_1 \circ \Delta$.*

Proof. Every item is proven by induction on the structure of Γ and by using the definition of splitting and predicates $\mathsf{un}(\cdot)$ and $\mathtt{lin}(\cdot)$. Details in App. A.2. ☐

We now prove unrestricted weakening and strengthening for $\pi_{\mathsf{R}}^{\lightning}$. The proofs are similar to the ones presented in the previous section. In the case of strengthening, Item (2) has been modified to account for the modifications in Def. 3.18. Most details are similar to the ones presented in § 3.1.1; thus, we only point out the most salient changes.

Lemma 3.22 (Unrestricted Weakening). *If $\Gamma \vdash P$ and $\mathsf{un}(T)$ then $\Gamma, x : T \vdash P$.*

Proof. The proof is immediate by induction on the derivation $\Gamma \vdash P$. It is similar to the one for Lem. 3.22. There are ten cases. ☐

Lemma 3.23 (Strengthening). *Let $\Gamma \vdash P$ and $x \notin \mathsf{fv}_\pi(P)$. Then the following holds:*

1. *If $x : \mathtt{lin}\, p$ then $x : \mathtt{lin}\, p \notin \Gamma$.*
2. *If $\Gamma = \Gamma', x : T$ and $\mathsf{un}(T)$ then $\Gamma' \vdash P$.*

Proof. Immediate by induction on the derivation $\Gamma \vdash P$. It is similar to the one for Lem. 3.10. There are ten cases. ☐

We now prove subject reduction. For this we must first prove subject congruence and a substitution lemma. The main difference in the proof with respect to the one shown in [Vas12] appear for the cases that involve Rule (T:Par).

Lemma 3.24 (Subject Congruence). *If $\Gamma \vdash P$ and $P \equiv_{\mathsf{S}} Q$ then $\Gamma \vdash Q$.*

Proof (sketch). By a case analysis on each axiom for $\equiv_{\mathsf{S}}$. In each case we prove both directions of the structural congruence. Each direction proceeds by induction on the typing derivation on P or Q (depending on the direction being proven). The proof proceeds similarly as in Lem. 3.11. ☐

Lemma 3.25 (Substitution). *If* $\Gamma_1 \vdash v : T$ *and* $\Gamma_2, x : T \vdash P$ *then* $\Gamma \vdash P\{v/x\}$*, with* $\Gamma = \Gamma_1 \circ \Gamma_2$.

Proof. By induction on the structure of P. Details in App. A.2. The proof is similar to that of Lem. 3.12. □

Theorem 3.26 (Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \longrightarrow^* Q$ *then* $\Gamma \vdash Q$.

Proof. By induction on the reduction $P \longrightarrow^* Q$ with a case analysis on the last applied rule. Details in App. A.2. □

We finish this section by collecting results that concern the process structure and are useful for proving the correctness of our translations. We first define well-formed processes:

Definition 3.27 (Well-Formed Process). An $\pi^{\lightning}_{\mathsf{R}}$ process P_0 is *well-formed* if for each of its structural congruent processes $P_0 \equiv_{\mathsf{S}} (\boldsymbol{\nu} x_1 y_1) \dots (\boldsymbol{\nu} x_n y_n)(P \mid Q \mid R)$, with $n \geq 0$, the following conditions hold:

1. If $P \equiv_{\mathsf{S}} v?\,(P'){:}(P'')$ then $v = \mathtt{tt}$ or $v = \mathtt{ff}$.
2. There does not exist processes P and Q such that they are prefixed on the same variable.
3. If P is prefixed at x_i and Q is prefixed at y_i, $1 \leq i \leq n$, then $P \mid Q$ is a redex.

By disallowing the sharing of qualified pre-types, the class of well-formed processes changes in $\pi^{\lightning}_{\mathsf{R}}$, as with $\pi^{\lightning}_{\mathsf{OR}}$ (cf. Def. 2.20). In this case, the second item of the definition disallows any pair of threads to be prefixed on the same variable. This contrasts with the definition of well-formed processes for $\pi^{\lightning}_{\mathsf{OR}}$ (cf. Def. 3.14), which allows to share variables with input-like types, and the definition of well-formed processes for π (cf. Def. 2.20), which allows both variables with output- and input-like behaviors to be shared. We now prove the type safety of well-formed programs. Notice that Not. 2.21 is lifted as it is for π.

Theorem 3.28 (Type Safety). *If* $\vdash P$ *then* P *is well-formed.*

Proof. By contradiction, for details see App. A.2.

□

We finish by stating a corollary that shows that well-formedness is preserved under reduction. It follows from Thm. 3.26 and Thm. 3.28.

Corollary 3.29. *If* $\vdash P$ *and* $P \longrightarrow^* Q$ *then* Q *is well-formed.*

A Big-Step Semantics for $\pi^{\lightning}_{\mathsf{R}}$

As hinted in the motivation for this section, there are some differences between the semantics of $\pi^{\lightning}_{\mathsf{R}}$ and the semantics of ReactiveML which we must reconcile. The races introduced in π are addressed by the type system presented above. The second one refers to the fact that in ReactiveML, a big-step reduction does not correspond to a single synchronization in $\pi^{\lightning}_{\mathsf{R}}$. Rather, as hinted in Ex. 2.43, several reactions can occur in a single RML big-step reduction. To illustrate this point, we use the following example:

Example 3.30. Consider the following ReactiveML expression:

$$e_4 = \texttt{signal}\ x, x'\ \texttt{in}\ (\texttt{emit}\ x\ 42 \parallel \texttt{await}\ x(y)\ \texttt{in}\ y \parallel \texttt{emit}\ x'\ 12 \parallel \texttt{await}\ x'(z)\ \texttt{in}\ z)$$

Using the semantic rules in Fig. 2.8 and Fig. 2.9, e_4 reduces into $42 \parallel 12$. Hence, two synchronizations have taken place in a single RML big-step reduction. △

The fact that several reactions occur in a single big-step reduction contrasts with the semantics of $\pi^{\sharp}_{\mathsf{R}}$, which only allows a single synchronization in an execution step. The reason for this disparity is rooted in the fact that in ReactiveML a big-step reduction corresponds to all the reactions in a single time instant. Therefore, all the outermost threads that can react will do so.

We tackle this discrepancy by introducing a big-step semantics for $\pi^{\sharp}_{\mathsf{R}}$, and proving that both the big-step semantics and the semantics given by Fig. 2.1 are semantically corresponding (cf. § 2.1.3).

The intuition behind the big-step semantics for $\pi^{\sharp}_{\mathsf{R}}$ must be in the fact that all possible synchronizations between threads in parallel must synchronize in a single step. We capture this idea with the following definition:

Definition 3.31 (Big-Step Semantics for $\pi^{\sharp}_{\mathsf{R}}$). We define the reduction $\hookrightarrow\!\!\!\!\!\rightarrow$ of $\pi^{\sharp}_{\mathsf{R}}$ by introducing the following rule, where $\longrightarrow$ is defined as in Fig. 2.1:

$\lfloor\textsc{Big-Step}\rfloor$

$$\frac{\begin{array}{c}\forall i,j \in \{n+1,\ldots,m\}.((\boldsymbol{\nu}\widetilde{x}\widetilde{y})P_i \not\longrightarrow \wedge (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_i \,|\, P_j) \not\longrightarrow) \\ \forall i\in\{1,\ldots,n\}.\exists j\in\{1,\ldots,n\}.((\boldsymbol{\nu}\widetilde{x}\widetilde{y})P_i \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})P'_i \vee (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_i \,|\, P_j) \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_i \,|\, P'_j))\end{array}}{(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_n \mid P_{n+1} \mid \ldots \mid P_m) \hookrightarrow\!\!\!\!\!\rightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid P'_n \mid P_{n+1} \mid \ldots \mid P_m)}$$

Rule $\lfloor\textsc{Big-Step}\rfloor$ executes all the *outermost reductions* of a $\pi^{\sharp}_{\mathsf{R}}$ process. Hence, a single step of $\hookrightarrow\!\!\!\!\!\rightarrow$ corresponds to a sequence of reductions using the rules in Fig. 2.1 that only reduces the outermost prefix or conditional operator of every thread.

Example 3.32. Consider the following $\pi^{\sharp}_{\mathsf{R}}$ process:

$$P_9 = (\boldsymbol{\nu}xy)(x\langle v\rangle.P_1 \mid x\langle v'\rangle.P_2 \mid y(z).P_3 \mid \texttt{tt}?\,(P_4)\!:\!(P_5)) \tag{3.5}$$

A possible sequence of outermost reductions would be:

$$P_9 \longrightarrow^2 (\boldsymbol{\nu}xy)(P_1 \mid x\langle v'\rangle.P_2 \mid P_3\{v/z\} \mid P_4)$$

Using Rule $\lfloor\textsc{Big-Step}\rfloor$, we would then have:

$$P_9 \hookrightarrow\!\!\!\!\!\rightarrow (\boldsymbol{\nu}xy)(P_1 \mid x\langle v'\rangle.P_2 \mid P_3\{v/z\} \mid P_4)$$

Notice that the big-step reduction $\hookrightarrow\!\!\!\!\!\rightarrow$ captures the nondeterministic behavior of untyped $\pi^{\sharp}_{\mathsf{R}}$ processes, induced by $\longrightarrow$. In particular, considering P_9 in (3.5) there are two possible reductions:

$$P_9 \hookrightarrow\!\!\!\!\!\rightarrow (\boldsymbol{\nu}xy)(P_1 \mid x\langle v'\rangle.P_2 \mid P_3\{v/z\} \mid P_4)$$
$$P_9 \hookrightarrow\!\!\!\!\!\rightarrow (\boldsymbol{\nu}xy)(x\langle v\rangle.P_1 \mid P_2 \mid P_3\{v'/z\} \mid P_4)$$

Another interesting case is replication, which introduces a new pre-redex in the process. Consider process:

$$P_{10} = (\boldsymbol{\nu} xy)(x\langle v\rangle.P_1 \mid x\langle v'\rangle.P_2 \mid *\,y(z).P_3 \mid \mathtt{tt}?\,(P_4)\!:\!(P_5)) \tag{3.6}$$

Notice that Rule $\lfloor \text{Rep} \rfloor$ in Fig. 2.1 would introduce a copy of $*\,y(z).P_3$ after a single synchronization. However, we do not want both $x\langle v\rangle.P_1$ and $x\langle v'\rangle.P_2$ to interact with $*\,y(z).P_3$ in a single $\hookrightarrow\!\!\!\!\rightarrow$ big-step reduction. Rather, we would like to impose an order in the possible reductions: only one output can interact with a replicated input in each step. Therefore, the other output must wait until the next step to interact with the replicated copy of the input. We then have that P_{10} in (3.6) can execute the following big-step reductions:

$$\begin{aligned} P_{10} &\hookrightarrow\!\!\!\!\rightarrow (\boldsymbol{\nu} xy)(P_1 \mid x\langle v'\rangle.P_2 \mid P_3\{^{v}\!/z\} \mid *\,y(z).P_3 \mid P_4) \\ P_{10} &\hookrightarrow\!\!\!\!\rightarrow (\boldsymbol{\nu} xy)(x\langle v\rangle.P_1 \mid P_2 \mid P_3\{^{v'}\!/z\} \mid *\,y(z).P_3 \mid P_4) \end{aligned}$$

△

Remark 3.33. Notice that the big-step semantics for $\pi^{\lightning}_{\mathsf{R}}$ allow the reduction $P \hookrightarrow\!\!\!\!\rightarrow P$ whenever all the parallel sub-processes of P are blocked.

Semantic Correspondence

We now prove the semantic correspondence (cf. Def. 2.9) between the big-step semantics of $\pi^{\lightning}_{\mathsf{R}}$ (i.e., $\hookrightarrow\!\!\!\!\rightarrow$) and the reduction semantics of $\pi^{\lightning}_{\mathsf{R}}$ (i.e., $\longrightarrow$). This result boils down to proving the two following statements: (1) a $\hookrightarrow\!\!\!\!\rightarrow$ big-step reduction can be simulated by several π reduction steps, and (2) for every single reduction step $\longrightarrow$, there exists some big-step reduction that contains it.

Before proving the semantic correspondence between $\hookrightarrow\!\!\!\!\rightarrow$ and $\longrightarrow$, we characterize the syntactic structure of well-typed π programs with the following corollary that follows from Thm. 3.26 and Thm. 3.28:

Corollary 3.34. *For every well-typed π program P (cf. Not. 2.21), it holds that:*

$$P \equiv_{\mathsf{S}} (\boldsymbol{\nu} \widetilde{x}\widetilde{y})(P_1 \mid P_2 \mid \ \ldots \ \mid P_n)$$

with $n \geq 1$ and every P_i, $1 \leq i \leq n$ is either a pre-redex (cf. Def. 2.19) or $P_i = v?\,(Q_1)\!:\!(Q_2)$.

Intuitively, well-typed programs can be written as the restriction of several threads which can be pre-redexes that synchronize or conditional constructs to be evaluated. The semantic correspondence statement follows:

Lemma 3.35 (Semantic Correspondence). *For every well-typed $\pi^{\lightning}_{\mathsf{R}}$ program P the following holds:*

1. *If $P \hookrightarrow\!\!\!\!\rightarrow Q$ then $P \longrightarrow^* Q$.*
2. *$P \longrightarrow^* Q$ then there exists Q' such that $P \hookrightarrow\!\!\!\!\rightarrow^* Q'$ and $Q \longrightarrow^* Q'$.*

Proof. Item 1 proceeds by induction on the number of parallel sub-processes in P. Item 2 proceeds by induction on the reduction $P \longrightarrow^* Q$. The IH guarantees that whenever $P \longrightarrow^* Q_0 \longrightarrow Q$ with $P \longrightarrow^* Q_0$ in $k \leq k_0$ steps, there exists Q_0' such that $P \hookrightarrow\!\!\!\!\rightarrow^* Q_0'$ and $Q_0 \longrightarrow^* Q_0'$. Then, to distinguish the reduction $Q_0 \longrightarrow Q$, we introduce a marked $\overset{\bullet}{\longrightarrow}$ as notation. Hence, $Q_0 \overset{\bullet}{\longrightarrow} Q$ denotes reduction $Q_0 \longrightarrow Q$. We also write $P \overset{\bullet}{\longrightarrow}{}^* P'$ to denote sequence $P \longrightarrow^* \overset{\bullet}{\longrightarrow} \longrightarrow^* P'$. We know, by IH, that $Q_0 \longrightarrow^* Q_0'$. Next, we shall apply a case analysis depending on whether reduction $P \overset{\bullet}{\longrightarrow} Q$ is included in the sequence $Q_0 \longrightarrow^* Q_0'$ (i.e., $Q_0 \overset{\bullet}{\longrightarrow}{}^* Q_0'$) or not. The former case is immediate from the IH.

The latter case follows by showing that if reduction $Q_0 \overset{\bullet}{\longrightarrow} Q$ is not contained in sequence $Q_0 \longrightarrow^* Q_0'$ then $Q_0' \overset{\bullet}{\longrightarrow} Q_0''$, for some Q_0''. To do this, we apply a case analysis on the nature of reduction $Q_0 \overset{\bullet}{\longrightarrow} Q$. There are two cases depending on whether the reduction comes from either a conditional (i.e., Rules $\lfloor \text{IfT} \rfloor$ or $\lfloor \text{IfF} \rfloor$) or a synchronization (i.e., Rules $\lfloor \text{Com} \rfloor$, $\lfloor \text{Sel} \rfloor$ or $\lfloor \text{Rep} \rfloor$). Both cases proceed similarly: using Def. 3.31 we analyze the shape of Q_0' and show the existence of reduction $Q_0' \overset{\bullet}{\longrightarrow} Q_0''$. Furthermore, the existence of $Q_0' \overset{\bullet}{\longrightarrow} Q_0''$ implies that $\overset{\bullet}{\longrightarrow}$ is an outermost reduction in Q_0', and therefore, there exists Q' such that $Q_0' \hookrightarrow\!\!\!\!\rightarrow Q'$. Then, by Lem. 3.35(1), $Q_0' \overset{\bullet}{\longrightarrow}{}^* Q'$ and then by composing the reduction sequences above, $Q_0 \longrightarrow^* Q_0' \overset{\bullet}{\longrightarrow}{}^* Q'$. Since $Q_0 \overset{\bullet}{\longrightarrow} Q$ and $Q_0' \overset{\bullet}{\longrightarrow} Q_0''$, we can rearrange the reductions as $Q_0 \overset{\bullet}{\longrightarrow} Q \longrightarrow^* Q'$. Finally, by the reasoning above and since $P \longrightarrow^* Q_0$, we obtain that $P \longrightarrow^* Q_0 \longrightarrow^* Q_0' \overset{\bullet}{\longrightarrow}{}^* Q'$ and thus, $P \longrightarrow^* Q'$, finishing the proof. For a more detailed proof see App. A.2 □

3.1.3 A Session π-Calculus with Session Establishment (π_{E})

In the section we introduce π_{E}, a conservative extension of $\pi^{\natural}_{\mathsf{OR}}$ with explicit *localities* and *session establishment*. We will use π_{E} as the source language for an `lcc` translation which allows us to isolate and analyze the security protocols required to implement session establishment. We first motivate π_{E} with some examples. Next, we introduce its syntax and semantics. Finally, we present the type system for π_{E} as a conservative extension of the type system of $\pi^{\natural}_{\mathsf{OR}}$ (cf. § 3.1.1).

Motivation

The calculus π_{E} is concerned with two important characteristics of communicating systems: session establishment and localities. Session establishment, as its name implies, means creating a connection between two channel endpoints. This connection will then be used for the session to proceed as intended. Notice that although an encoding of session establishment is presented in [Vas12], π does not have a dedicated construct for this purpose. Intuitively, π_{E} has two new constructs: *request* and *accept*, whose semantics embody the idea of session creation. Moreover, these new constructs are also enriched with the notion of *locality*, an idea reminiscent of the distributed π-calculus [Hen07]. Intuitively, localities allow us to represent *distributed services* that interact with each other while residing in different locations. This is a common feature of web services, as both physical and logical localities are used to delimit systems with different communication and security conditions. As an example, consider the distinct regions and zones in platforms such as Amazon Web Services (AWS) [Ama]. We have designed our request and accept construct in such a way that distributed services can be part of *private* localities whose services can only be requested by clients residing in *authorized locations*.

The request and accept constructs in π_{E} declare replicated services and clients, respectively:

- $\left[* \, a^{\rho}(y).P\right]^{m}$ is the declaration of a persistent service called a, with implementation given by process P, which resides in location m and only accepts requests from the locations contained in the set ρ.
- $\left[\overline{a}^{m}\langle x\rangle.Q\right]^{n}$ is a request of service a, from some client located in n with implementation Q.

Our intention is that persistent services and requests interact in order to establish sessions as long as they are authorized by their respective locations. This is formalized by the following reduction rule:

$$\left[\overline{a}^{m}\langle x\rangle.P\right]^{n} \mid \left[* \, a^{\rho}(y).Q\right]^{m} \longrightarrow_{\mathsf{N}} (\boldsymbol{\nu} xy)(P \mid Q) \mid \left[* \, a^{\rho}(y).Q\right]^{m}$$

which is enabled only if $n \in \rho$, i.e., if n is in ρ. Observe that once the session has been established (i.e., the rightmost process), the new process becomes essentially a $\pi_{\mathsf{OR}}^{\lightning}$ process composed with one or more service declarations in parallel. For π_{E} we shall follow *Barendregt's convention*, whereby all channel names in binding occurrences in any mathematical context are pairwise distinct and also distinct from the free names.

Below, we put the previous intuitions in context by describing a simple example. Suppose that there are three locations in our system: i_1, i_2, and i_3. We have an online store residing in i_1 that due to government regulations can only sell items to people residing in locations i_1 and i_2. We model the store below, with only two items for simplicity.

$$P_s = \left[* \, a^{\{i_1,i_2\}}(y).y \triangleright \{\mathit{item}_1\colon y\langle p_1\rangle.y(z).y\langle e_1\rangle, \mathit{item}_2\colon y\langle p_2\rangle.y(z).y\langle e_2\rangle\}\right]^{i_1} \tag{3.7}$$

Above, the store is modeled by a persistent service that can establish a connection with a client. Once the session has been established, the store waits for the client to select an item from its menu. The store then sends the price of the item (i.e., p_i, $i \in \{1,2\}$), waits for a confirmation from the client and, finally, sends the estimated arrival time (i.e., e_i, $i \in \{1,2\}$).

The clients that interact with P_s can be written by using the dual constructs. In the two processes below we leave the continuations undefined, as we are only interested in the session establishment phase:

$$P_c = \left[\overline{a}^{i_1}\langle x\rangle.Q_1\right]^{i_1}$$
$$P_c' = \left[\overline{a}^{i_1}\langle x\rangle.Q_2\right]^{i_3}$$

There are two clients above: P_c resides in i_1, whereas P_c' resides in i_3. Then, the π_{E} system will correspond to what we call a *network*:

$$N = P_s \mid P_c \mid P_c'$$

Notice that the network above only has one possible reduction, as client P_c' resides in i_3, and is therefore unauthorized to interact with P_s. This corresponds to the desired behavior as we only want client P_c, residing in i_1, to establish a session with service P_s.

Syntax and Semantics

The syntax of π_{E} conservatively extends that of $\pi^{\lightning}_{\mathsf{OR}}$ by adding a new syntactic category of *networks*, which range over $N, M, \ldots$. Networks represent concurrent services which reside in distinguished *locations* and seek to establish sessions. Formally, we reuse the set $\mathcal{V}_s$ as the set of variables for π_{E}, which ranges over $x, y, \ldots$. Also, let $\mathcal{S}_\pi$ be a set of *service names*, ranged over by $a, b, \ldots$, which will be used to establish sessions. We will use $u, v, \ldots$ to denote elements in $\mathcal{V}_s \cup \mathcal{S}_\pi$. Also, let Ω_π be a set of locations that ranges over $m, n, \ldots, i_1, i_2, \ldots$. We will use $\rho, \rho', \ldots$ to denote sets of locations.

Recall that since π_{E} is an extension of $\pi^{\lightning}_{\mathsf{OR}}$, the syntax of π processes builds upon that of π (cf. Def. 2.10, § 3.1.1). We present it below:

Definition 3.36 (Syntax). An π_{E} network is defined by the following syntax:

$$N, M ::= \big[* \, a^{\rho}(x).P\big]^{m} \mid \big[\overline{a}^{m}\langle x\rangle.P\big]^{n} \mid (\boldsymbol{\nu} xy)P \mid \mathbf{0} \mid M \mid N$$

$$P, Q ::= x\langle v\rangle.P \mid x(y).P \mid x \triangleleft l.P \mid x \triangleright \{l_i : P_i\}_{i\in I} \mid *\, x(y).P \mid v?\,(P){:}(Q) \mid P \mid Q \mid \mathbf{0}$$

Networks in π_{E} represent an additional layer of interaction atop $\pi^{\lightning}_{\mathsf{OR}}$ processes. As hinted above, there are two additional constructs to represent session establishment: $\big[* \, a^{\rho}(x).P\big]^{m}$ for service accept and $\big[\overline{a}^{m}\langle y\rangle.P\big]^{n}$ for service request. Formally, $\big[* \, a^{\rho}(x).P\big]^{m}$ defines a service identified by $a \in \mathcal{S}_\pi$ whose behavior P resides in location m. Variable x denotes a variable, bound in P. This service may only establish sessions with requests from locations included in ρ. The service on a will persist after successful interactions with requests. Dually, $\big[\overline{a}^{m}\langle x\rangle.P\big]^{n}$ denotes a request of a service named $a \in \mathcal{S}_\pi$ and located at m. This service request itself resides at n, and has continuation P. Variable x is bound in P.

For simplicity of presentation, we disallow nested sessions, i.e., the establishment of new sessions inside a session process. The syntax of networks also includes parallel composition and inaction, as well as the construct $(\boldsymbol{\nu} xy)P$, which can be seen as a run-time construct, not occurring in "source" processes. As we will see, a network N will be *starting* if it does not contain occurrences of $(\boldsymbol{\nu} xy)P$, and it will be *runtime* otherwise. That is, a starting network will correspond to the composition of services and their requests.

In π_{E}, the set of free variables of a network ($\mathsf{fv}_\pi(\cdot)$) corresponds to the set of free variables of the π processes that compose it. This occurs because we have that $\mathsf{fv}_\pi(\big[* \, a^{\rho}(x).P\big]^{m}) = \mathsf{fv}_\pi(\big[\overline{a}^{m}\langle x\rangle.P\big]^{n}) = \emptyset$.

The operational semantics for π_{E} is given by a reduction relation, denoted $\longrightarrow_{\mathsf{N}}$. It is defined via the rules in Fig. 3.3, building upon the reduction relation for $\pi^{\lightning}_{\mathsf{OR}}$ processes given in Fig. 2.1. In particular, Rule $\lfloor \text{SRed} \rfloor$, appeals to the semantics of $\pi^{\lightning}_{\mathsf{OR}}$, even if in π_{E} $(\boldsymbol{\nu} xy)P$ is a network rather than a process.

As usual, the reflexive, transitive closure of $\longrightarrow_{\mathsf{N}}$ is written $\longrightarrow^{*}_{\mathsf{N}}$. Informally speaking, our operational semantics enables a starting network to evolve into a runtime network, composed of $\pi^{\lightning}_{\mathsf{OR}}$ processes and persistent service definitions. In the following, we shall refer to those $\pi^{\lightning}_{\mathsf{OR}}$ processes simply as processes.

In a slight abuse of notation, we will write $N \equiv_{\mathsf{S}} M$ when two π_{E} networks are structurally congruent. The same goes for $P \equiv_{\mathsf{S}} Q$, in the case of π_{E} processes. The rules of structural congruence for networks are presented below. We then briefly

$$\lfloor\text{SEstR}\rfloor\ \left[\overline{a}^m\langle x\rangle.P\right]^n \mid \left[* \, a^\rho(y).Q\right]^m \longrightarrow_{\mathsf{N}} (\boldsymbol{\nu}xy)(P \mid Q) \mid \left[* \, a^\rho(y).Q\right]^m \quad (n \in \rho)$$

$$\begin{array}{ccc} \lfloor\text{Par}\rfloor & \lfloor\text{SRed}\rfloor & \lfloor\text{NStr}\rfloor \\ \dfrac{N \longrightarrow_{\mathsf{N}} N'}{N \mid M \longrightarrow_{\mathsf{N}} N' \mid M} & \dfrac{(\boldsymbol{\nu}xy)P \longrightarrow (\boldsymbol{\nu}xy)P'}{(\boldsymbol{\nu}xy)P \longrightarrow_{\mathsf{N}} (\boldsymbol{\nu}xy)P'} & \dfrac{N \equiv_{\mathsf{S}} N', N' \longrightarrow M', M' \equiv_{\mathsf{S}} M}{N \longrightarrow_{\mathsf{N}} M} \end{array}$$

Figure 3.3: Reduction rules for networks in π_{E} (extends Fig. 2.1).

discuss the changes in this definition with respect to Def. 2.11. In particular, the differences that appear in the scope extrusion rule:

$$N \mid \mathbf{0} \equiv_{\mathsf{S}} N \quad N \mid M \equiv_{\mathsf{S}} M \mid N \quad N \equiv_{\mathsf{S}} M \text{ if } M \equiv_\alpha N$$
$$(N \mid M) \mid L \equiv_{\mathsf{S}} N \mid (M \mid L) \quad (\boldsymbol{\nu}xy)(\boldsymbol{\nu}wz)P \equiv_{\mathsf{S}} (\boldsymbol{\nu}wz)(\boldsymbol{\nu}xy)P \quad (\boldsymbol{\nu}xy)\mathbf{0} \equiv_{\mathsf{S}} \mathbf{0}$$
$$(\boldsymbol{\nu}xy)P \mid (\boldsymbol{\nu}wz)Q \equiv_{\mathsf{S}} (\boldsymbol{\nu}xy)(\boldsymbol{\nu}wz)(P \mid Q) \quad \text{if } x, y \notin \mathsf{fv}_\pi(Q) \text{ and } w, z \notin \mathsf{fv}_\pi(P)$$

As hinted above, the scope extrusion rule is stated in accordance to the syntax of π_{E}. Namely, since $(\boldsymbol{\nu}xy)P$ is a network, scope extrusion is only present at a network level in π_{E}. It is for this reason that the scope extrusion rule requires two processes with restriction operators.

Type System

We consider a simple type system for π_{E} networks, defined on top of the $\pi_{\mathsf{OR}}^{\lightning}$ type system (cf. § 3.1.1). Typing judgments are of the form $\Phi \vdash_{\mathsf{N}} N$, where Φ is a typing context containing assignments of the form $a : \langle T\rangle$ where T is a $\pi_{\mathsf{OR}}^{\lightning}$ session type type (cf. § 3.1.1) and a is a service name. We extend Fig. 2.2 as follows:

$$\Phi ::= \emptyset \;\big|\; \Phi, a : \langle T\rangle$$

Typing rules for networks are in Fig. 3.4; some intuitions follow. Rules (T:NPar) and (T:NNil) are analogous to their counterparts for π processes (cf. Rules (T:Par) and (T:Nil) in Fig. 2.3); the only difference is that Φ behaves as an unrestricted environment in π. Rule $\lfloor$T:Req$\rfloor$ checks if the type for service a in Φ is dual to the type of its implementation. Rule $\lfloor$T:RAcc$\rfloor$ is similar. Finally, Rule $\lfloor$T:Sess$\rfloor$ closes endpoints x, y, checks for duality, and requires P to be well-typed using type judgments for π processes.

We now prove the type preservation property for the type system for networks. To this end, we rely on auxiliary results from the type system for $\pi_{\mathsf{OR}}^{\lightning}$ presented in § 3.1.1. We start by proving our subject congruence result.

Lemma 3.37 (Subject Congruence for π_{E}). *If $\Phi \vdash_{\mathsf{N}} N$ and $N \equiv_{\mathsf{S}} M$ then $\Phi \vdash_{\mathsf{N}} M$.*

Proof. Using a case analysis on all the rules for $\equiv_{\mathsf{S}}$. For details see App. A.3. □

We then proceed to prove subject reduction for the π_{E} type system. We do not need more auxiliary results, as they follow from the results in § 3.1.1.

$$\lfloor\text{T:NNIL}\rfloor\ \Phi \vdash_{\mathsf{N}} \mathbf{0}$$

$$(\text{T:REQ})\ \frac{\Phi \vdash_{\mathsf{N}} a : \langle T\rangle \quad x : \overline{T} \vdash P}{\Phi \vdash_{\mathsf{N}} \left[\overline{a}^{m}\langle x\rangle.P\right]^{n}} \qquad (\text{T:RACC})\ \frac{\Phi \vdash_{\mathsf{N}} a : \langle T\rangle \quad x : T \vdash P}{\Phi \vdash_{\mathsf{N}} \left[* \, a^{\rho}(x).P\right]^{m}}$$

$$(\text{T:NPAR})\ \frac{\Phi \vdash_{\mathsf{N}} N_1 \quad \Phi \vdash_{\mathsf{N}} N_2}{\Phi \vdash_{\mathsf{N}} N_1 \mid N_2} \qquad (\text{T:SESS})\ \frac{x : T, y : \overline{T} \vdash P}{\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)P}$$

Figure 3.4: Typing rules for networks in π_{E}.

Theorem 3.38 (Subject Reduction for π_{E}). *If $\Phi, \Gamma \vdash_{\mathsf{N}} N$ and $N \longrightarrow_{\mathsf{N}}^{*} N'$ then $\Phi, \Gamma \vdash_{\mathsf{N}} N'$.*

Proof. By induction on k, the length of the reduction, followed by a case analysis on the last applied rule. For details see App. A.3. □

We also extend the notions of pre-redex and redex from $\pi_{\mathsf{OR}}^{\lightning}$ (cf. Def. 2.19). They are built on top of Def. 2.19. We distinguish between process pre-redexes and network pre-redexes, according to syntax in Def. 3.36.

Definition 3.39 (Pre-redexes and Redexes in π_{E}). We say $x\langle v\rangle.P$, $x(y).P$, $x \triangleleft l.P$, $x \triangleright \{l_i : P_i\}_{i\in I}$, and $*\, x(y).P$ are *process pre-redexes* (*at variable x*). A *process redex* is a process R such that $(\boldsymbol{\nu} xy)R \longrightarrow$ and:

1. $R = v?\,(P){:}(Q)$ with $v \in \{\mathtt{tt}, \mathtt{ff}\}$ (or)
2. $R = x\langle v\rangle.P \mid y(z).Q$ (or)
3. $R = x\langle v\rangle.P \mid *\, y(z).Q$ (or)
4. $R = x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$, with $j \in I$.

Similarly, networks $\left[\overline{a}^{m}\langle x\rangle.P\right]^{n}$ and $\left[*\, a^{\rho}(x).P\right]^{n}$ are both *network pre-redexes* at service name a and that $R = \left[\overline{a}^{m}\langle x\rangle.P_1\right]^{n} \mid \left[*\, a^{\rho}(y).P_2\right]^{m}$, with $n \in \rho$, is a *network redex*. We will say that any process redex or network redex R is *communicating* whenever $R \neq v?\,(P){:}(Q)$.

Whenever the kind of pre-redex is unimportant we will call them simply pre-redexes. We will re-use notation Not. 2.24 for pre-redexes $y(z).Q$ and $*\, y(z).Q$. We now define (*typable*) *closed networks*, as the analogues of well-typed programs in π (cf. Not. 2.21):

Notation 3.40 ((*Typable*) *Closed Networks*). A network N such that $\mathsf{fv}_{\pi}(N) = \emptyset$ is called a *closed network*. A closed network is typable if $\Phi \vdash_{\mathsf{N}} N$ holds for some environment Φ.

We now prove type safety for runtime networks. This theorem relies on Thm. 3.15. This property will allow us to preserve the well-formedness of the $\pi_{\mathsf{OR}}^{\lightning}$ processes that conform the network (cf. Def. 3.14).

Lemma 3.41 (Type Safety for Runtime Networks). *If $N \equiv_{\mathsf{S}} (\boldsymbol{\nu} xy)P \mid M$ and $\Phi \vdash_{\mathsf{N}} N$ then $(\boldsymbol{\nu} xy)P$ is well-formed (cf. Def. 3.14).*

Proof. Straightforward from the previous results. For details see App. A.3. □

3.2 An Asynchronous Session π-Calculus ($\mathsf{a}\pi$)

In this section we introduce $\mathsf{a}\pi$, a session π-calculus with asynchronous (queue-based) semantics. The design of this calculus follows the typed framework of Kouzapas et al. [KYHH16].

In § 3.2.1 we motivate $\mathsf{a}\pi$ by using some examples and providing informal intuitions about its design. Next, in § 3.2.2, we formally introduce the syntax and (asynchronous) semantics of $\mathsf{a}\pi$. The type system is introduced in § 3.2.3, § 3.2.4, § 3.2.5, and § 3.2.6. Finally, as with $\pi_{\mathsf{R}}^{\lightning}$, we present a big-step semantics for $\mathsf{a}\pi$ and prove it semantically corresponds with the reduction semantics (cf. § 3.2.7 and § 3.2.8).

3.2.1 *Motivation*

As mentioned in § 2.4, one interesting aspect of ReactiveML is the fact that signals can be emitted at any moment during execution, independently of whether they are detected or not (they are nonblocking). Hence, synchronization in ReactiveML can be seen as being *asynchronous*. The natural question is then whether an asynchronous session π-calculus would be a better fit to RML than a synchronous calculus like $\pi_{\mathsf{R}}^{\lightning}$, in terms of the properties satisfied by the translation. To answer this question we use $\mathsf{a}\pi$. In this calculus, communication occurs in two *phases*: (1) processes interact with *local queues*, which contain messages being received or sent, and (2) queues in charge of dual endpoints synchronize *remotely*.

Intuitively, the *buffered* semantics of $\mathsf{a}\pi$ aims at modeling the low-level mechanism of communication that can be found in network transport protocols such as TCP (Transmission Control Protocol). Moreover, the semantics of $\mathsf{a}\pi$ provides a *fine-grained* view of interactions by mediating synchronizations with FIFO (first in, first out) buffers, which help to model the non-blocking property of asynchrony. In general, implementations of TCP protocols should be able to correctly preserve the order of the sent messages, as the program must be able to reconstruct the overall message using the packets sent. In $\mathsf{a}\pi$, this order-preserving property is ensured by a session type system equipped with *subtyping* [GH05].

In $\mathsf{a}\pi$, every channel endpoint is assigned two local queues: an *input queue*, which contains all the messages that have been received, and an *output queue*, containing all the messages just before they are sent. Then, communication occur between queues, rather than processes: a process writes messages to the output queue corresponding to some channel endpoint x, which then transmits to the endpoint queue of its complementary endpoint $\overline{x}$. The transmitted messages can be read from this input queue by the receptor process.

Syntactically, $\mathsf{a}\pi$ is very similar to π, $\pi_{\mathsf{OR}}^{\lightning}$, and $\pi_{\mathsf{R}}^{\lightning}$. The main differences is that we use a restriction operator that only binds a single variable (i.e., binding x also binds its complementary endpoint $\overline{x}$), following [KYHH16]. To illustrate this, consider the following $\mathsf{a}\pi$ process:

$$\begin{aligned} P_{11} = (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(&x\langle \texttt{ff}\rangle.\mathbf{0} \mid \overline{x}(z).(y\langle \texttt{tt}\rangle.\mathbf{0} \mid \overline{y}(z').\mathbf{0}) \mid \\ &x[i:\epsilon, o:\epsilon] \mid \overline{x}[i:\epsilon, o:\epsilon] \mid y[i:\epsilon, o:\epsilon] \mid \overline{y}[i:\epsilon, o:\epsilon]) \end{aligned} \tag{3.8}$$

Process P_{11} can be seen as having two parts: the first one, in the first line, is similar to π: we have a process that is sending ff over endpoint x, to be received by endpoint $\overline{x}$, which then will activate a similar communication on endpoints y and $\overline{y}$. The main difference is in the second line: processes $x[i:\epsilon, o:\epsilon]$, $\overline{x}[i:\epsilon, o:\epsilon]$, $y[i:\epsilon, o:\epsilon]$, and $\overline{y}[i:\epsilon, o:\epsilon]$ denote the queues of endpoints x and y, as well as their dual endpoints $\overline{x}$ and $\overline{y}$. In P_{11}, the process implementing x writes ff in the queue $x[i:\epsilon, o:\epsilon]$, which then synchronizes with the queue $\overline{x}[i:\epsilon, o:\epsilon]$. Then, the process using $\overline{x}$ reads this value. At this point, the interactions between the processes implementing endpoints y and $\overline{y}$ can be executed.

Remark 3.42. With respect to the constructs introduced in [KYHH16], we do not consider session establishment, nor we allow shared variables and configurations. The absence of session establishment is used to keep the symmetry with π (which does not consider explicit session establishment), while the absence of shared variables is needed for reasons similar to the ones presented in § 3.1.2.

3.2.2 *Syntax and Semantics*

The syntax of $\mathsf{a}\pi$ includes variables $x, y, \ldots$ and *co-variables*, denoted $\overline{x}, \overline{y}$. Intuitively, x and $\overline{x}$ denote the two endpoints of a session, with $\overline{\overline{x}} = x$. We write $\mathcal{V}_a$ to denote the set of variables and co-variables; $k, k', \ldots$ will be used to range over $\mathcal{V}_a$. We use $\widetilde{m}, \widetilde{m'}, \ldots$ to range over sequence of values. As in π, the set of values includes booleans and variables, and will be denoted $\mathcal{U}_a$. We also assume a countably infinite set of labels $\mathcal{B}_\pi$. The syntax of processes is as follows:

Definition 3.43 ($\mathsf{a}\pi$ **and** $\mathsf{a}\pi^\star$)**.** The set $\mathsf{a}\pi$ of asynchronous session processes is defined as:

$$\begin{aligned} P, Q \quad ::= \quad & k\langle v\rangle.P \mid k(y).P \mid k \triangleleft l.P \mid k \triangleright \{l_i : P_i\}_{i\in I} \mid v?\,(P)\!:\!(Q) \mid P \mid Q \mid \mathbf{0} \\ \mid \quad & (\boldsymbol{\nu} x)P \mid \mu X.P \mid X \mid k[i:\widetilde{m}; o:\widetilde{m}] \end{aligned}$$

We write $\mathsf{a}\pi^\star$ to denote the sub-language of $\mathsf{a}\pi$ without queues.

Differences with respect to Def. 2.10 appear in the second line above. The usual (single) restriction $(\boldsymbol{\nu} x)P$ is convenient in a queue-based setting; it binds both x and $\overline{x}$ in P. We consider recursion $\mu X.P$ rather than input-guarded replication. Communication in $\mathsf{a}\pi$ is mediated by queues of messages m, which can be values v or labels l; we use ϵ to denote the empty queue. Given an endpoint k, the process $k[i:\widetilde{m}; o:\widetilde{m}]$ explicitly represents the output and input parts of the queue; it appears shaded, as we consider this a runtime process, similarly to queues in MPST (§ 2.5).

Synchronization between processes proceeds similarly to MPST, with an added step: the sending endpoint first enqueues the message m in its own output queue; then, m is moved to the input queue of the receiving endpoint; finally, the receiving endpoint retrieves m from its input queue.

$$\lfloor\textsc{Send}\rfloor\ \ x\langle v\rangle.P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathsf{A}} P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}\cdot v]$$
$$\lfloor\textsc{Sel}\rfloor\ \ x \triangleleft l.P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathsf{A}} P \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}\cdot l]$$
$$\lfloor\textsc{Com}\rfloor\ \ x[i:\widetilde{m_1}, o:m\cdot\widetilde{m_2}] \mid \overline{x}[i:\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathsf{A}} x[i:\widetilde{m_1}, o:\widetilde{m_2}] \mid \overline{x}[i:\widetilde{m_1}\cdot m, o:\widetilde{m_2}]$$
$$\lfloor\textsc{Recv}\rfloor\ \ x(y).P \mid x[i:v\cdot\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathsf{A}} P\{v/y\} \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}]$$
$$\lfloor\textsc{Bra}\rfloor\ \ x \triangleright \{l_i : P_i\}_{i\in I} \mid x[i:l_j\cdot\widetilde{m_1}, o:\widetilde{m_2}] \longrightarrow_{\mathsf{A}} P_j \mid x[i:\widetilde{m_1}, o:\widetilde{m_2}] \quad (j\in I)$$
$$\lfloor\textsc{IfT}\rfloor\ \ \texttt{tt?}\,(P)\,\texttt{:}\,(Q) \longrightarrow_{\mathsf{A}} P \qquad \lfloor\textsc{IfF}\rfloor\ \ \texttt{ff?}\,(P)\,\texttt{:}\,(Q) \longrightarrow_{\mathsf{A}} Q$$
$$\lfloor\textsc{Rec}\rfloor\ \ \mu X.P \longrightarrow_{\mathsf{A}} P\{\mu X.P/X\}$$
$$\lfloor\textsc{Res}\rfloor\ \frac{P \longrightarrow_{\mathsf{A}} P'}{(\boldsymbol{\nu}x)P \longrightarrow_{\mathsf{A}} (\boldsymbol{\nu}x)P'} \qquad \lfloor\textsc{Par}\rfloor\ \frac{P \longrightarrow_{\mathsf{A}} P'}{P \mid R \longrightarrow_{\mathsf{A}} P' \mid R} \qquad \lfloor\textsc{Str}\rfloor\ \frac{P \equiv_{\mathsf{A}} P',\ P' \longrightarrow_{\mathsf{A}} Q',\ Q' \equiv_{\mathsf{A}} Q}{P \longrightarrow_{\mathsf{A}} Q}$$

Figure 3.5: Reduction relation for aπ processes.

We assume the expected notions of free/bound variables. In particular, we use $\mathsf{fpv}(P)$ to denote the free process variables in P and $\mathsf{fv}_\pi(P)$ to denote the free variables of process P. The operational semantics of aπ can be then defined as a reduction relation coupled with a structural congruence relation $\equiv_{\mathsf{A}}$ (given below). Reduction is defined by the rules in Fig. 3.5, which are either exactly as those for π or follow the above intuitions for queue-based message passing.

Definition 3.44. Structural congruence is defined as the smallest congruence on processes that satisfies the following axioms:

$$\text{(SC}_{\mathsf{a}}\pi\text{:1)}\ (\boldsymbol{\nu}x)(\boldsymbol{\nu}y)P \equiv_{\mathsf{A}} (\boldsymbol{\nu}y)(\boldsymbol{\nu}x)P \qquad \text{(SC}_{\mathsf{a}}\pi\text{:2)}\ (\boldsymbol{\nu}x)\mathbf{0} \equiv_{\mathsf{A}} \mathbf{0} \qquad \text{(SC}_{\mathsf{a}}\pi\text{:3)}\ (\boldsymbol{\nu}x)(x[i:\epsilon; o:\epsilon] \mid \overline{x}[i:\epsilon; o:\epsilon]) \equiv_{\mathsf{A}} \mathbf{0}$$
$$\text{(SC}_{\mathsf{a}}\pi\text{:4)}\ P \mid \mathbf{0} \equiv_{\mathsf{A}} P \qquad \text{(SC}_{\mathsf{a}}\pi\text{:5)}\ P \mid Q \equiv_{\mathsf{A}} Q \mid P \qquad \text{(SC}_{\mathsf{a}}\pi\text{:6)}\ (P \mid Q) \mid R \equiv_{\mathsf{A}} P \mid (Q \mid R)$$
$$\text{(SC}_{\mathsf{a}}\pi\text{:7)}\ \frac{P \equiv_\alpha Q}{P \equiv_{\mathsf{A}} Q} \qquad \text{(SC}_{\mathsf{a}}\pi\text{:8)}\ \frac{x \notin \mathsf{fv}_\pi(Q)}{(\boldsymbol{\nu}x)P \mid Q \equiv_{\mathsf{A}} (\boldsymbol{\nu}x)(P \mid Q)}$$

We now define (evaluation) contexts for aπ. For this calculus we shall consider two types of contexts: unary and binary. We do this distinction because aπ processes are considered to have two parts: non-queue processes (i.e., a$\pi^\star$ processes) and queue processes.

Definition 3.45 (Contexts for aπ). The syntax of (unary) contexts in aπ is given by the following grammar:

$$E ::= [-] \mid E \mid P \mid P \mid E \mid (\boldsymbol{\nu}x)E$$

where P is an aπ process and '$[\cdot]$' represents a 'hole'. Moreover, we write $C[-_1, -_2]$ to denote binary contexts $(\boldsymbol{\nu}\widetilde{x})([-_1] \mid [-_2])$. We shall also write $E[P]$ (resp. $C[P,Q]$) to denote the aπ process obtained by filling the hole in $E[-]$ (resp. $C[-_1, -_2]$) with P (resp. P and Q).

Due to the fact that both π and aπ abstract from an explicit phase of session initiation (cf. Rem. 3.42), we find it useful to identify aπ processes which are *properly* *initialized* (*PI*): intuitively, these PI processes contain all the queues required to reduce.

Definition 3.46 (Properly Initialized Processes). Let $P = C[P_1, P_2]$ with $\mathsf{fv}_\pi(P) = \emptyset$ be an $\mathsf{a}\pi$ process such that P_1 is in $\mathsf{a}\pi^\star$ (i.e., it does not include queues) and $\mathsf{fv}_\pi(P_1) = \{k_1, \ldots, k_n\}$. We say P is *properly initialized* (*PI*) if P_2 contains (empty) queues for each session declared in P_1, i.e., if $P_2 = k_1[i : \epsilon, o : \epsilon] \mid \cdots \mid k_n[i : \epsilon, o : \epsilon]$.

3.2.3 Type System

The type system we develop for $\mathsf{a}\pi$ follows closely the one in [KYHH16]. As mentioned above, we do not consider explicit session initiation nor shared types. First, we assume a single *ground type*, the boolean `bool`. We also consider *session types*, ranged over $T, T', \ldots$, which describe communication behavior along endpoints. We assume that $U, U', \ldots$ range over ground types and session types.

Definition 3.47 (Asynchronous Session Types). The syntax of types is given by the grammar below:

$$\begin{array}{lcl} U, U' & ::= & \mathsf{bool} \mid T \\ T, T' & ::= & !U.T \mid ?U.T \mid \&\{l_i : T_i\}_{i\in I} \mid \oplus\{l_i : T_i\}_{i\in I} \mid \mathbf{t} \mid \mu\mathbf{t}.T \mid \mathsf{end} \end{array}$$

The types presented above correspond to the types in Fig. 2.2 without qualifiers. Therefore, recall that type `end` represents a terminated session and `bool` represents booleans. Similarly, recall that type $!U.T$ is assigned to an endpoint that sends a value of type U and then continues according to type T. Dually, type $?U.T$ is assigned to an endpoint that receives a value of type U and then proceeds according to type T. Types $\oplus\{l_i : T_i\}_{i\in I}$ and $\&\{l_i : T_i\}_{i\in I}$ are assigned to endpoints that implement selection (internal choice) and branching (external choice), respectively. Type $\mu\mathbf{t}.T$ allows us to express recursive protocols. As in π, we take an equi-recursive view of types, which means that a recursive type is assumed to be equal to its unfolding as they represent the same regular infinite trees. We also assume $\mu\mathbf{t}.T$ is assumed to be contractive (i.e., containing no subexpression of the form $\mu\mathbf{t}_1.\ldots\mu\mathbf{t}_n.\mathbf{t}_1$).

3.2.4 Subtyping and Duality

Subtyping in session types provides flexibility when typing processes. It is characterized by the notion of *composability*, which determines which behaviors are composable and the ways in which they can be composed. We illustrate the convenience of subtyping using a simplified version of a classical example, taken from [GH05].

Example 3.48. Suppose a mathematical server that can compute the boolean conjunction of two boolean values and the negation. The type of such server T_s can be defined as follows:

$$\begin{array}{l} T_s = \&\{\mathit{con}\colon ?\mathsf{bool}.?\mathsf{bool}.!\mathsf{bool}.\mathsf{end} \\ \qquad\qquad \mathit{not}\colon ?\mathsf{bool}.!\mathsf{bool}.\mathsf{end}\} \end{array}$$

As mentioned above, type $\&\{\ldots\}$ denotes an offer made from the server to the client who must then select one of the above options. The client's type would then be the complementary of T_s, written $\overline{T_s}$:

$$\begin{aligned}\overline{T_s} = \oplus\{&con\colon \texttt{!bool.!bool.?bool.end}\\ &not\colon \texttt{!bool.?bool.end}\}\end{aligned}$$

Observe that the communication structure remains the same, but $\overline{T_s}$ describes the complementary operations to T_s (i.e., instead of a receive, the client must execute a send and vice-versa). In other words, during an execution, the client can "select" a label and execute behavior complementary to the one exhibited by the server.

Suppose then that the server is updated to allow now the disjunction of boolean operators. The update type T'_s would now be:

$$\begin{aligned}T'_s = \&\{&con\colon \texttt{?bool.?bool.!bool.end}\\ &not\colon \texttt{?bool.!bool.end}\\ &dis\colon \texttt{?bool.?bool.!bool.end}\}\end{aligned}$$

It is possible to observe that a server implementing T'_s should be able to safely interact with a client implementing $\overline{T_s}$, since the complementary interactions for labels *con* and *not* are present in T'_s. This means, in a sense, that T'_s is *composable* with more peers than T_s, as T'_s implement the same or more behaviors than T_s. This means that T_s is a *subtype* of T'_s. $\triangle$

Formally, due to the presence of recursive types, our subtyping relation must be defined coinductively. To give such definition, we follow the approach in [Pie02]. In our definitions we assume $\mathcal{P}(A)$ denotes the powerset of set A.

Definition 3.49 (Subtyping Relation for Session Types). Let $\mathcal{T}$ be the closed set of contractive session types. We say that a type T_1 is a subtype of T_2, written $T_1 \lesssim T_2$, if the pair (T_1, T_2) is in the largest fixed point of the monotone function $\mathcal{F} : \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$ defined by:

$$\begin{aligned}
\mathcal{F}(\mathcal{R}) \overset{\mathsf{def}}{=}\ & \{(\mathsf{end}, \mathsf{end}), (\mathsf{bool}, \mathsf{bool})\} && (1)\\
& \cup \{(!U_1.T_2, !U'_1.T'_2) \mid (U'_1, U_1), (T_2, T'_2) \in \mathcal{R}\} && (2)\\
& \cup \{(?U_1.T_2, ?U'_1.T'_2) \mid (U_1, U'_1), (T_2, T'_2) \in \mathcal{R}\} && (3)\\
& \cup \{(\oplus\{l_i : T_i\}_{i\in I}, \oplus\{l_j : T'_j\}_{j\in J}) \mid I \subseteq J \wedge \forall i \in I.(T_i, T'_i) \in \mathcal{R}\} && (4)\\
& \cup \{(\&\{l_i : T_i\}_{i\in I}, \&\{l_j : T'_j\}_{j\in J}) \mid J \subseteq I \wedge \forall j \in J.(T_j, T'_j) \in \mathcal{R}\} && (5)\\
& \cup \{(\mu\mathbf{t}.T, T') \mid (T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\}, T') \in \mathcal{R}\} \cup \{(T, \mu\mathbf{t}.T') \mid (T, T'\{^{\mu\mathbf{t}.T'}/_{\mathbf{t}}\}) \in \mathcal{R}\} && (6)
\end{aligned}$$

We comment on the definition above. First, line (1) is standard, as we require `end` and `bool` to be subtypes of themselves. Similarly, lines (2) and (3) are as expected: output is contravariant on the message type and input is covariant on the message type. Next, lines (4) and (5) deal with selection and branching: in line (4), a selection is covariant in the number of labels whereas in line (5), branching is contravariant. Finally, line (6) deals with recursion, and embodies the idea of equi-recursivity: we require that the subtype of a recursive type must be a subtype of its unfolding.

Next, we introduce duality; due to our addition of subtyping, duality must also be defined coinductively. The definition follows the coinductive definition presented in [BDGK14] by Bernardi et al.

Definition 3.50 (Duality). Let $\mathcal{T}$ be the closed set of contractive session types. We say that a type T_1 is a dual of T_2, written $T_1 \perp T_2$, if the pair (T_1, T_2) is in the largest fixed point of the monotone function $\mathcal{D} : \mathcal{P}(\mathcal{T} \times \mathcal{T}) \to \mathcal{P}(\mathcal{T} \times \mathcal{T})$ defined by:

$$\begin{aligned}\mathcal{D}(\mathcal{R}) \overset{\mathsf{def}}{=} & \{(\mathsf{end}, \mathsf{end})\} \cup \{(!U_1.T_2, ?U_1'.T_2') \mid U_1 \sim U_1' \wedge (T_2, T_2') \in \mathcal{R}\} \\ & \cup \{(?U_1.T_2, !U_1'.T_2') \mid U_1 \sim U_1' \wedge (T_2, T_2') \in \mathcal{R}\} \\ & \cup \{(\oplus\{l_i : T_i\}_{i\in I}, \&\{l_i : T_i'\}_{i\in I}) \mid \forall i \in I.(T_i, T_i') \in \mathcal{R}\} \\ & \cup \{(\&\{l_i : T_i\}_{i\in I}, \oplus\{l_i : T_i'\}_{i\in I}) \mid \forall i \in I.(T_i, T_i') \in \mathcal{R}\} \\ & \cup \{(\mu\mathbf{t}.T, T') \mid (T\{^{\mu\mathbf{t}.T}/\mathbf{t}\}, T') \in \mathcal{R}\} \cup \{(T, \mu\mathbf{t}.T') \mid (T, T'\{^{\mu\mathbf{t}.T'}/\mathbf{t}\}) \in \mathcal{R}\}\end{aligned}$$

where $\sim$ denotes equivalence up-to equality of trees (see [Pie02, Vas12]).

The duality relation pairs session types with complementary behaviors. Duality shows which pairs of session types can safely interact. Hence, an input is the dual of output (and vice versa), and branching is the dual of selection (and vice versa). Using duality, we then can state the following property of subtyping:

Lemma 3.51. *Let T_1, T_2, T_1', and T_2' be session types such that $T_1 \perp T_1'$ and $T_2 \perp T_2'$. Then, it holds that $T_1 \lesssim T_2$ if and only if $T_2' \lesssim T_1'$.*

Proof. We prove both directions by coinduction:

$\Rightarrow$) By assumption, $T_1 \lesssim T_2$, $T_1 \perp T_1'$ and $T_2 \perp T_2'$ for some T_1' and some T_2'. Then, by definition of $\lesssim$, there exists a relation $\mathcal{S}$ such that $(T_1, T_2) \in \mathcal{S}$, called a *subtyping relation*. We need to prove that there exists a relation $\mathcal{S}'$ such that $(T_2', T_1') \in \mathcal{S}'$. Let $\mathcal{S}' = \{(T_2', T_1') \mid T_1 \lesssim T_2\}$. It can be shown that $\mathcal{S}'$ is a subtyping relation by induction on the structure of T_1, finishing the proof.

$\Leftarrow$) This follows a similar strategy as above. The subtyping relation we show in this case is $\mathcal{S}' = \{(T_1, T_2) \mid T_2' \lesssim T_1'\}$.

□

Next, we define the set of *composable behaviors* of a type T, this is, the set of session types that can be composed with a given type T, up to subtyping:

Definition 3.52 (Composable Types). The set of *composable* types of a session type T with a dual T' (i.e., $T \perp T'$), written $\mathtt{comp}(T)$, is defined as:

$$\mathtt{comp}(T) \overset{\mathsf{def}}{=} \{T'' \mid T'' \lesssim T'\}$$

We now state some properties of subtyping, taken from [KYHH16]. Informally, we need to ensure that the subtyping relation is a pre-order and that the set of composable types of a session types is bigger or equal than its subtypes.

Lemma 3.53 (Properties of Subtyping [KYHH16]).

(1) *The subtyping relation $\lesssim$ is a pre-order.*

(2) *For every pair of session types S_1 and S_2, it holds that $S_1 \lesssim S_2$ if and only if it holds that $\mathtt{comp}(S_2) \subseteq \mathtt{comp}(S_1)$.*

To clarify the intuitions behind subtyping, we review Ex. 3.48, considering the previously presented definitions. Below, we show how to prove that a type is a subtype of another and we analyze the relation between duality and subtyping.

Example 3.54. Let us consider the following types:

$$\begin{aligned}
T_s &= \&\{con\colon ?\mathtt{bool}.?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}\\
&\qquad\quad not\colon ?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}\}\\
\overline{T_s} &= \oplus\{con\colon !\mathtt{bool}.!\mathtt{bool}.?\mathtt{bool}.\mathtt{end}\\
&\qquad\quad not\colon !\mathtt{bool}.?\mathtt{bool}.\mathtt{end}\}\\
T_s' &= \&\{con\colon ?\mathtt{bool}.?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}\\
&\qquad\quad not\colon ?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}\\
&\qquad\quad dis\colon ?\mathtt{bool}.?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}\}
\end{aligned}$$

The first thing to notice is that using our notion of duality (cf. Def. 3.50), we can establish that $T_s \perp \overline{T_s}$, formalizing our notion of complementarity between session types.

Next, we will show that $T_s \lesssim T_s'$. To do this, we exhibit a subtyping relation $\mathcal{R}$ that is a subset of $\mathcal{F}(\mathcal{R})$ (cf. Def. 3.49). Consider:

$$\begin{aligned}
\mathcal{R} = \{&(T_s, T_s'), (?\mathtt{bool}.?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}, ?\mathtt{bool}.?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}),\\
&(?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}, ?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}), (!\mathtt{bool}.\mathtt{end}, !\mathtt{bool}.\mathtt{end})\\
&(?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}, ?\mathtt{bool}.!\mathtt{bool}.\mathtt{end}), (!\mathtt{bool}.\mathtt{end}, !\mathtt{bool}.\mathtt{end}),\\
&(\mathtt{end}, \mathtt{end}), (\mathtt{bool}, \mathtt{bool})\}
\end{aligned}$$

It can be shown that $\mathcal{R}$ is a subset of $\mathcal{F}(\mathcal{R})$ and therefore $T_s \lesssim T_s'$. Furthermore, by Lem. 3.51, we have that for some type $\overline{T_s'}$, if it holds that $T_s' \perp \overline{T_s'}$, then $\overline{T_s'} \lesssim \overline{T_s}$, which can also be proven by showing an appropriate relation.

Summarizing, this example shows that, assuming $n \geq 1$ and $m \geq 1$, a branching offering n can be replaced by a branching type offering $n + m$ options, whereas a selection with n options can be replaced by a selection type containing $n - m$ options. △

3.2.5 *Typing Processes and Queues*

In defining typing for asynchronous calculi, it is common to distinguish between *pure processes* (processes without queues) and *runtime processes* (processes with queues) [HYC08, KYHH16]. In our case, this means distinguishing between $\mathsf{a}\pi^\star$ and the full syntax given by $\mathsf{a}\pi$ (cf. Def. 3.43). To define a type system for $\mathsf{a}\pi$, we first extend the syntax in Def. 3.47 by adding *message types*, which we use to type queues $k[i : \widetilde{m}; o : \widetilde{m}]$:

Definition 3.55 (Message Types). We extend the syntax of session types in Def. 3.47 with *message types* M as follows:

$$\begin{array}{llll}
\text{(General)} & G ::= T \mid M & \text{(Input Messages)} & M_i ::= \emptyset \mid ?U.M_i \mid \&\, l.M_i\\
\text{(Messages)} & M ::= M_i \mid M_o & \text{(Output Messages)} & M_o ::= \emptyset \mid !U.M_o \mid \oplus\, l.M_o
\end{array}$$

Our type system uses two judgments, one for variables and constants and another one for (runtime) processes:

$$\Gamma \vdash v : U \qquad \Gamma \vdash_{\Sigma} P \triangleright \Delta$$

where *classical* and *linear* environments Γ and Δ are defined as follows:

$$\Gamma, \Gamma' ::= \emptyset \mid \Gamma, x : U \mid X : \Delta \qquad \Delta, \Delta' = \emptyset \mid \Delta, k : G$$

A linear environment Δ collects information regarding session types and queues. Notice that $\Delta, k : G$ is only well-defined if $k \notin dom(\Delta)$. The classical environment Γ collects all the types of variables of ground types and process variables, which are assigned *linear* environments Δ. In the judgment for (runtime) processes $\Gamma \vdash_{\Sigma} P \triangleright \Delta$, we use Σ to denote a set that contains all the free session names for the queues implemented in P. Whenever $P \in \mathsf{a}\pi^{\star}$, we write $\Gamma \vdash P \triangleright \Delta$ instead of $\Gamma \vdash_{\emptyset} P \triangleright \Delta$. Before introducing our typing rules we introduce some auxiliary definitions:

Definition 3.56 (Auxiliary Definitions for Typing Rules). Let Δ and Δ' such that $dom(\Delta) = dom(\Delta')$ be linear environments. Then:

1. We say that $\Delta \lesssim \Delta'$ whenever for every $k \in dom(\Delta)$ it holds that $\Delta(k) \lesssim \Delta'(k)$.
2. The predicate $\mathsf{end}(\Delta)$ is true whenever for every $k \in \Delta$, $\Delta(k) = \mathsf{end}$ and false otherwise.

Fig. 3.6 presents the typing rules. Before commenting on them, we introduce an auxiliary *merging operator* on types. Due to the asynchronous communication of $\mathsf{a}\pi$, it is possible that the full behavior of the endpoint is split between the process and its corresponding queues. Therefore, it is necessary to define a way to reconstruct the overall session type that captures the behavior of the session.

Definition 3.57 (Merging Session Types and Message Types). Let T be a session type and M be a message type. The merging operation $T \asymp M$ is defined as follows:

$$\begin{aligned} T \asymp \emptyset &\overset{\mathsf{def}}{=} T \\ T \asymp\, !U.M_o &\overset{\mathsf{def}}{=}\, !U.(T \asymp M_o) && (1) \\ T_k \asymp \oplus l_k.M_o &\overset{\mathsf{def}}{=} \oplus\{l_i : T_i \asymp M_o\}_{i \in I}\ (k \in I) && (2) \\ ?U.T \asymp\, ?U.M_i &\overset{\mathsf{def}}{=} T \asymp M_i && (3) \\ \&\{l_i : T_i\}_{i \in I} \asymp \& l_k.M_i &\overset{\mathsf{def}}{=} l_k \asymp M_i\ (k \in I) && (4) \end{aligned}$$

and undefined otherwise. Given linear environments Δ_1 and Δ_2, the definition is extended as follows:

$$(\Delta_1 \asymp \Delta_2)(k) \overset{\mathsf{def}}{=} \begin{cases} T & \textit{if}\ \exists i \in \{1,2\}.(k : T \in \Delta_i \wedge k \notin dom(\Delta_1) \cap dom(\Delta_2)) \\ T \asymp M & \textit{if}\ k : T \in \Delta_i \wedge k : M \in \Delta_j\ \textit{with}\ i \neq j \wedge i, j \in \{1,2\} \end{cases}$$

In Def. 3.57, numerals (1)–(4) represent the inductive cases. The base case, which is not numbered, corresponds to merging any session type T with the type of an empty queue (i.e., $\emptyset$).

In (1), we merge a session type T with the type of an output queue that contains a message of type U to be sent. The merging appends an output type sending a value of type U at the beginning of T before inductively calling itself on T and M_o.

In (2), the merging of a session type T_k with the message type of an output queue $\oplus l_k.M_o$, creates a type T' such that $\oplus\{l_k : T_k\} \lesssim T'$. For example, suppose the merging $!\texttt{bool}.\texttt{end} \asymp \oplus l.\emptyset$. Notice that, up-to subtyping, this merging can yield either $\oplus\{l : !\texttt{bool}.\texttt{end}\}$ or $\oplus\{l : !\texttt{bool}.\texttt{end}, l' : T\}$, depending on what is needed for typing.

In (3) and (4), merging a session type with an input message type M_i assumes that the message has arrived already to their final destination. Thus, merging "consumes" the outermost action of the type and continue inductively, merging the continuations. Specifically, in (3), when merging an input type $?U.T$ with a message type $?U.M_i$, we consume prefix $?U$ and require the merge of T and M_i. Similarly, in (4), the merging picks the corresponding label l_k and is called inductively on the continuations.

The definition is extended to linear environments in a way that guarantees that whenever the same session endpoint k is found in both Δ_1 and Δ_2, one of them corresponds to a message type M and the other to a session type T. Thus, ensuring that it is not possible for Δ_1 and Δ_2 to share a session endpoint with the same session type or message type.

The rules in Fig. 3.6 are grouped in three parts, depending on the shape of Σ. The first group at the top of the figure contains the rules that deal with $\mathsf{a}\pi^\star$ processes (i.e., $\Sigma = \emptyset$, hence omitted). The second group, in the middle, contains rules for typing queues; finally, the third group, at the bottom, contains rules dealing with inaction, weakening of Σ, restriction, parallel composition and subtyping. We briefly comment on all the rules below:

- Rules (T:Bool), (T:Var) and (T:RVar) are standard. They deal with booleans, variables and recursive variables. Notice that the recursive variable is assigned a Δ environment to allow for interleaved sessions inside a recursion.

- Rule (T:Send) types an output process by verifying the message is of the correct type and that the continuation types with T.

- Analogously, Rule (T:Rcv) types an input process by adding variable x into the classical environment Γ and typing the continuation P with the corresponding type T.

- Rule (T:Sel) types a selection operator by ensuring that the selected label j belongs to the set I and that the continuation P types with the corresponding T_j.

- Rule (T:Bra) requires that all the possible continuations from a branching construct are typable with their corresponding types.

- Rules (T:Del) and (R:DRcv) are similar to Rules (T:Send) and (T:Rcv), dealing with delegation.

- Rules (T:If) and (T:Rec) are standard. The former asks for both branches of the conditional to be typable with the same environments. The latter adds the

$$\text{(T:Bool)}\ \Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathsf{bool} \qquad \text{(T:Var)}\ \Gamma, x : U \vdash x : u \qquad \text{(T:RVar)}\ \Gamma, X : \Delta \vdash X \triangleright \Delta$$

$$\text{(T:Send)}\ \frac{\Gamma \vdash v : U \quad \Gamma \vdash P \triangleright \Delta, k : T}{\Gamma \vdash k\langle v \rangle.P \triangleright \Delta, k : !U.T} \qquad \text{(T:Rcv)}\ \frac{\Gamma, x : U \vdash P \triangleright \Delta, k : T}{\Gamma \vdash k(x).P \triangleright \Delta, x : ?U.T}$$

$$\text{(T:Sel)}\ \frac{\Gamma \vdash P \triangleright \Delta, k : T_j \quad j \in I}{\Gamma \vdash k \triangleleft l_j.P \triangleright \Delta, k : \oplus\{l_i : T_i\}_{i \in I}}$$

$$\text{(T:bra)}\ \frac{\forall i \in I. \Gamma \vdash P_i \triangleright \Delta, k : T_i}{\Gamma \vdash k \triangleright \{l_i : P_i\}_{i \in I}.P \triangleright \Delta, k : \&\{l_i : T_i\}_{i \in I}}$$

$$\text{(T:Del)}\ \frac{\Gamma \vdash P \triangleright \Delta, k : T_2}{\Gamma \vdash k\langle k' \rangle.P \triangleright \Delta, k : !T_1.T_2, k' : T_1} \qquad \text{(T:DRcv)}\ \frac{\Gamma \vdash P \triangleright \Delta, x : T_1, k : T_2}{\Gamma \vdash k(x).P \triangleright \Delta, x : ?T_1.T_2}$$

$$\text{(T:If)}\ \frac{\Gamma \vdash v : \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash v?\,(P)\,\mathbf{:}\,(Q) \triangleright \Delta} \qquad \text{(T:Rec)}\ \frac{\Gamma, X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X.P \triangleright \Delta}$$

$$\text{(T:IQEnd)}\ \frac{\mathsf{end}(\Delta)}{\Gamma \vdash_k k[i : \epsilon, o : \widetilde{m_2}] \triangleright \Delta, k : \emptyset} \qquad \text{(T:OQEnd)}\ \frac{\mathsf{end}(\Delta)}{\Gamma \vdash_k k[i : \widetilde{m_1}, o : \epsilon] \triangleright \Delta, k : \emptyset}$$

$$\text{(T:QOut)}\ \frac{\Gamma \vdash v : U \quad \Gamma \vdash_k k[i : \widetilde{m_1}, o : \widetilde{m_2}] \triangleright \Delta, k : M_o}{\Gamma \vdash_k k[i : \widetilde{m_1}, o : v \cdot \widetilde{m_2}] \triangleright \Delta, k : !U.M_o}$$

$$\text{(T:QIn)}\ \frac{\Gamma \vdash v : U \quad \Gamma \vdash_k k[i : \widetilde{m_1}, o : \widetilde{m_2}] \triangleright \Delta, k : M_i}{\Gamma \vdash_k k[i : v \cdot \widetilde{m_1}, o : \widetilde{m_2}] \triangleright \Delta, k : ?U.M_i}$$

$$\text{(T:QSel)}\ \frac{\Gamma \vdash_k k[i : \widetilde{m_1}, o : \widetilde{m_2}] \triangleright \Delta, k : M_o}{\Gamma \vdash_k k[i : \widetilde{m_1}, o : l \cdot \widetilde{m_2}] \triangleright \Delta, k : \oplus l.M_o} \qquad \text{(T:QBra)}\ \frac{\Gamma \vdash_k k[i : \widetilde{m_1}, o : \widetilde{m_2}] \triangleright \Delta, k : M_i}{\Gamma \vdash_k k[i : l \cdot \widetilde{m_1}, o : \widetilde{m_2}] \triangleright \Delta, k : \& l.M_i}$$

$$\text{(T:Nil)}\ \frac{\mathsf{end}(\Delta)}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \text{(T:WkS)}\ \frac{k \notin \mathsf{fv}_\pi(P) \quad \Gamma \vdash_\Sigma P \triangleright \Delta}{\Gamma \vdash_{\Sigma,k} P \triangleright \Delta} \qquad \text{(T:Res)}\ \frac{T \perp T' \quad \Gamma \vdash_{\Sigma,x,\overline{x}} P \triangleright \Delta, x : T, \overline{x} : T'}{\Gamma \vdash_\Sigma (\boldsymbol{\nu} x)P \triangleright \Delta}$$

$$\text{(T:QConc)}\ \frac{\Gamma \vdash_{\Sigma_1} P \triangleright \Delta_1 \quad \Gamma \vdash_{\Sigma_2} Q \triangleright \Delta_2 \quad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Gamma \vdash_{\Sigma_1 \cup \Sigma_2} P \mid Q \triangleright \Delta_1 \asymp \Delta_2} \qquad \text{(T:Sub)}\ \frac{\Gamma \vdash_\Sigma P \triangleright \Delta \quad \Delta \lesssim \Delta'}{\Gamma \vdash_\Sigma P \triangleright \Delta'}$$

Figure 3.6: Typing rules for aπ.

recursive variable X assigned to environment Δ to Γ and types the continuation P.

- Rules (T:IQEnd) and (T:OQEnd) type empty input and output queues, respectively. In both cases, the rule requires that the session name of the queue k is the only element in Σ and that the type is the empty message type $\emptyset$.

- Rules (T:QOUT) and (T:QIN) type queues that contain messages in the output and input queue, respectively. In both cases Σ is required to contain the appropriate session name for each queue. Also, the types of the values inside the queue are tested with respect to the message types.
- Rules (T:SELQ) and (T:BRAQ) are similar to $\lfloor$T:QOUT$\rfloor$ and $\lfloor$T:QIN$\rfloor$, but deal with labels in the queues.
- Rule (T:NIL) types an inactive process with a *terminated* linear environment. The condition $\mathsf{end}(\Delta)$ means that Δ only contains end types.
- Rule (T:WKS) allows you to add a session name k to Σ provided that $k \notin \mathsf{fv}_\pi(P)$.
- Rule (T:RES) requires process P to be typed with the appropriate dual session endpoints and with $\Sigma, k, \overline{k}$, ensuring that both queues for k and $\overline{k}$ are present in P.
- Rule (T:QCONC) types two processes in parallel by using the merge operator $\asymp$ on the linear environments Δ_1, Δ_2.
- Finally, Rule (T:SUB) deals with subtyping and is defined as expected.

3.2.6 *Typing Properties*

In this section we present all the guarantees ensured by the type system of aπ. First, we present some important preliminary notions; we start by defining well-configured linear environments. Intuitively, these environments capture the idea of "correct" environments. The assumption is that for every entry $k : T$ in Δ, there exists an entry corresponding to $\overline{k} : T'$, with T' being the complementary of T—notice that this is similar to the notion of consistent environments in MPST (cf. § 2.5).

Definition 3.58 (Well-Configured Linear Environment). A linear environment Δ is *well-configured* if, for every $k \in dom(\Delta)$, $\Delta(k) = T$ implies $\Delta(\overline{k}) = T'$ and $T \perp T'$.

Next, we introduce a notion of reduction for our linear environments. Informally, this definition is used to precisely characterize the evolution of session types due to process reductions.

Definition 3.59 (Linear Environment Reduction). Let Δ and Δ' be linear environments. Then, we define the follow reduction rules:

$$\lfloor\text{E:COM}\rfloor\ \{k :!U.T, \overline{k} :?U.T'\} \rightharpoonup \{k : T, \overline{k} : T'\}$$

$$\lfloor\text{E:SEL}\rfloor\ \{k : \oplus\{l_i : T_i\}_{i\in I}, \overline{k} : \&\{l_i : T_i'\}_{i\in I}\} \rightharpoonup \{k : T_j, \overline{k} : T_j'\} \quad (j \in I)$$

$$\lfloor\text{E:COMP}\rfloor\ \frac{\Delta_1 \rightharpoonup \Delta_1'}{\Delta_1, \Delta_2 \rightharpoonup \Delta_1', \Delta_2}$$

Remark 3.60. It is perhaps useful to recall that π does not have an analogue to Def. 3.59 above (cf. § 2.2). This difference arises because the semantics of π (cf. Fig. 2.1) require the synchronizing endpoints to be always bound by a restriction.

We now prove weakening, strengthening and substitution lemmas:

Lemma 3.61 (Weakening Lemmas). *Let* $\Gamma \vdash_{\Sigma} P \triangleright \Delta$*:*

1. *If* $X \notin dom(\Gamma)$ *then* $\Gamma, X : \Delta' \vdash_{\Sigma} P \triangleright \Delta$*, for any* Δ'*.*
2. *If* $k \notin dom(\Delta)$ *then* $\Gamma \vdash_{\Sigma} P \triangleright \Delta, k : \mathsf{end}$*.*

Proof. We prove each numeral:

1. By induction on the structure of the aπ process P. The base cases are whenever $P = \mathbf{0}$ and $P = k[i : \widetilde{h_1}, o : \widetilde{h_2}]$. Both are immediate by inversion and reapplying their corresponding typing rule. All the inductive cases, except for $P = \mu X'.P'$ conclude by applying inversion, applying the IH and reapplying the corresponding typing rule. For the case $P = \mu X'.P'$ we also consider the fact that $X \notin dom(\Gamma)$ to apply the IH and conclude by reapplying Rule (T:Rec).
2. By induction on the structure of process P. The base cases are whenever $P = \mathbf{0}$ and $P = k[i : \widetilde{h_1}, o : \widetilde{h_2}]$. As above, they are straightforward by using inversion and applying the corresponding rule. Notice that adding $k : \mathsf{end}$ in Δ means that the environment is still terminated and therefore, the condition $\mathsf{end}(\Delta)$ still holds. The inductive cases proceed by using inversion, the IH and reapplying the corresponding typing rule.

□

Lemma 3.62 (Strengthening Lemmas). *Let* $\Gamma \vdash_{\Sigma} P \triangleright \Delta$*:*

1. *If* $X \notin \mathsf{fpv}(P)$ *then* $\Gamma, X : \Delta' \vdash_{\Sigma} P \triangleright \Delta$ *implies* $\Gamma \vdash_{\Sigma} P \triangleright \Delta$*.*
2. *If* $k \notin \mathsf{fv}_{\pi}(P)$ *then* $\Gamma \vdash_{\Sigma} P \triangleright \Delta, k : T$ *implies* $\Gamma \vdash_{\Sigma} P \triangleright \Delta$*.*

Proof. Both numerals are proven by induction on the structure of P. We only detail 1 as 2 is similar.

Base Cases: There are two base cases: (1) $P = \mathbf{0}$ and (2) $P = k[i : \widetilde{h_1}, o : \widetilde{h_2}]$. Both cases follow immediately from inversion and reapplying the corresponding typing rule.

Inductive Step: For the inductive step, we only detail the case $P = \mu X'.P'$ as all the other cases are similar. By assumption, $\Gamma, X : \Delta' \vdash_{\Sigma} \mu X'.P' \triangleright \Delta$ and $X \notin \mathsf{fpv}(\mu X.P')$. We can then deduce that $\Sigma = \emptyset$. Furthermore, by inversion, $\Gamma, X : \Delta' \vdash \mu X'.P' \triangleright \Delta$. Then, by applying inversion once more, $\Gamma, X : \Delta', X' : \Delta \vdash P' \triangleright \Delta$ and $X \notin \mathsf{fpv}(P')$. Thus, by IH, $\Gamma, X' : \Delta \vdash_{\Sigma} P' \triangleright \Delta$ and by reapplying Rule (T:Rec), we obtain $\Gamma \vdash_{\Sigma} \mu X.P' \triangleright \Delta$, concluding the case.

□

Lemma 3.63 (Substitution Lemmas).

1. *If* $\Gamma, x : U \vdash_{\Sigma} P \triangleright \Delta$ *and* $\Gamma \vdash v : U$ *then* $\Gamma, v : U \vdash_{\Sigma} P\{^{v}/_{x}\} \triangleright \Delta$*.*
2. *If* $\Gamma \vdash_{\Sigma} P \triangleright \Delta, k : T$ *then* $\Gamma \vdash_{\Sigma} P\{^{k'}/_{k}\} \triangleright \Delta, k' : T$*.*

Proof. We prove both items by induction on the structure of P.

1. By induction on the structure of P and a case analysis on the free variables of the process.

   Base Case: There are two base cases: (1) $P = \mathbf{0}$ and (2) $P = k[i : \widetilde{h_1}, o : \widetilde{h_2}]$. We detail (2) as the other cases are immediate. By assumption, $P = k[i : \widetilde{h_1}, o : \widetilde{h_2}]$. Then, we distinguish four cases (i) $\widetilde{h_1} = \epsilon, \widetilde{h_2} = \epsilon$, (ii) $\widetilde{h_1} \neq \epsilon, \widetilde{h_2} = \epsilon$, (iii) $\widetilde{h_1} = \epsilon, \widetilde{h_2} \neq \epsilon$, (iv) $\widetilde{h_1} \neq \epsilon, \widetilde{h_2} \neq \epsilon$. Case (i) is immediate, as no substitutions can be made. Notice that our type system does not allow for delegation and hence, substitution of channels cannot occur. Below we detail Case (iv).

   By assumption, $\Gamma, x : U \vdash_\Sigma k[i : \widetilde{h_1}, o : \widetilde{h_2}] \triangleright \Delta$. We can then deduce that $\Sigma = k$ and $\Delta = \Delta', k : M$. We then distinguish cases depending on whether M is M_i or M_o. We only detail whenever $M = M_o$. Then, without loss of generality, assume that $\widetilde{h_2} = v_1 \cdot \ldots \cdot v_m \cdot x \cdot \ldots \cdot v_n, n, m \geq 1, m \leq n$. By applying inversion m times we have that $\Gamma, x : U \vdash_\Sigma k[i : \widetilde{h_1}, o : x \cdot \ldots \cdot v_n] \triangleright \Delta', !U.M'_o$. Then, by inversion and assumption ($\Gamma \vdash v : U$), we can apply Rule (T:QOut) to conclude that $\Gamma, v : U \vdash_\Sigma k[i : \widetilde{h_1}, o : v \cdot \ldots \cdot v_n] \triangleright \Delta', !U.M'_o$. We finish the proof by reapplying Rule (T:QOut) m times.

   Inductive Steps: All the inductive cases proceed similarly as the base case detailed above. The only difference is that the IH is used to deal with the continuations of the processes.

2. The proof is similar to the one above. The main difference is that for this statement whenever $P = k\langle v \rangle.Q$ and $P = k(x).Q$, we assume that P is typed using Rule (T:Del) and (T:DRcv), respectively. Both cases finish by applying the IH.

□

We finally present the main results for our the type system: subject reduction and type safety. The latter ensures that our processes do not reduce to *errors* as understood in [KYHH16]. Before giving the formal definition of error processes, we introduce the idea behind them with an example:

Example 3.64. Below, we assume that every process Q_i, $i \in \{1, 2, 3\}$ contains the necessary queues for making the process reduce.

$$\begin{aligned}
P_{12} &= (\boldsymbol{\nu} x)(x\langle \mathtt{tt} \rangle.\mathbf{0} \mid \overline{x}\langle \mathtt{tt} \rangle.\mathbf{0} \mid Q_1) \\
P_{13} &= (\boldsymbol{\nu} x)(x\langle \mathtt{tt} \rangle.x(y).\mathbf{0} \mid \overline{x}(z).\overline{x}\langle \mathtt{ff} \rangle.\mathbf{0} \mid Q_2) \\
P_{14} &= (\boldsymbol{\nu} x)(x\langle \mathtt{tt} \rangle.x(y).\mathbf{0} \mid x\langle \mathtt{tt} \rangle.\mathbf{0} \mid \overline{x}(z).\overline{x}\langle \mathtt{ff} \rangle.\mathbf{0} \mid Q_3)
\end{aligned}$$

We consider processes P_{12} and P_{14} errors, and process P_{13} as a non-error process. △

Formally, errors are defined in terms of k-processes and k-redexes. Intuitively, a k-process is any process whose subject is k and does not have a parallel composition operator at top level; similarly, a k-redex is a redex formed only by k-processes.

Definition 3.65 (k-Redex). A k-redex is one of the following compositions of two k-processes:

1. $k\langle v\rangle.P \mid k[i:\widetilde{h_1}, o:\widetilde{h_2}]$.
2. $k(x).P \mid k[i: v\cdot\widetilde{h_1}, o:\widetilde{h_2}]$.
3. $k \triangleleft l.P \mid k[i:\widetilde{h_1}, o:\widetilde{h_2}]$.
4. $k \triangleright \{l_i : P_i\}_{i\in I} \mid k[i: l_j\cdot\widetilde{h_1}, o:\widetilde{h_2}]$, with $j\in I$.
5. $k[i:\widetilde{h_1}, o: v\cdot\widetilde{h_2}] \mid \overline{k}[i:\widetilde{h_1}, o:\widetilde{h_2}]$.

Definition 3.66 (Errors). We say that an $\mathsf{a}\pi$ process P is an error whenever $P \equiv_{\mathsf{A}} (\boldsymbol{\nu}\widetilde{x})(Q_1 \mid Q_2 \mid R)$ for any pair of processes Q_1 and Q_2 which are both k-processes but do not form a k-redex.

Before stating our subject reduction theorem, we prove subject congruence to ensure that typing is preserved by the structural congruence in Def. 3.44.

Theorem 3.67 (Subject Congruence for $\mathsf{a}\pi$). *If $\Gamma \vdash_\Sigma P \triangleright \Delta$ and $P \equiv_{\mathsf{A}} Q$ then $\Gamma \vdash_\Sigma Q \triangleright \Delta$.*

Proof. By a case analysis on the rules of $\equiv_{\mathsf{A}}$ (cf. Def. 3.44). For details see App. A.4. □

Next, we present an auxiliary result that clarifies the semantics of well-typed processes. In [KYHH16], the authors demonstrated that for any given endpoint k, the reduction of a well-typed process under a well-configured linear environment implies that the queue of k, corresponds to a process $Q_k = k[i:\widetilde{h_1}, o:\widetilde{h_2}]$ with $\widetilde{h_1} = \epsilon$ or $\widetilde{h_2} = \epsilon$. This statement is useful for proving subject reduction, because in the cases involving queues, the result implies that only one set of rules (i.e., rules for input queues or output queues) is applicable. Below we show the original statement as the authors presented it. The proof can be found in the cited paper.

Lemma 3.68 ([KYHH16]). *Let $P = P_1 \mid k[i:\epsilon, o:\epsilon]$ and $\Gamma \vdash_\Sigma P \triangleright \Delta$ with Δ well-configured (cf. Def. 3.58). Then if $P \longrightarrow_{\mathsf{A}} P_1' \mid k[i:\widetilde{h_1}, o:\widetilde{h_2}]$ then $\widetilde{h_1} = \epsilon$ or $\widetilde{h_2} = \epsilon$.*

Theorem 3.69 (Subject Reduction for $\mathsf{a}\pi$). *If $\Gamma \vdash_\Sigma P \triangleright \Delta$ with Δ well-configured (cf. Def. 3.58) and $P \longrightarrow_{\mathsf{A}}^* Q$ then $\Gamma \vdash_\Sigma Q \triangleright \Delta'$ with $\Delta \rightharpoonup^* \Delta'$ and Δ' is well-configured.*

Proof. By induction on the length of reduction $P \longrightarrow_{\mathsf{A}}^* Q$. The base case is immediate. For details see App. A.4. □

We now state type safety for the $\mathsf{a}\pi$ type system. This result ensures that communication executes correctly for well-typed $\mathsf{a}\pi$ programs (cf. Def. 2.21). In other words, type safety ensures that well-typed programs never reduce to an error.

Theorem 3.70 (Type Safety for $\mathsf{a}\pi$). *If P is a well-typed program, then $\Gamma \vdash_\emptyset P \triangleright \emptyset$ and P never reduces to an error.*

Proof. The proof is a direct consequence of subject reduction (Thm. 3.69). The proof goes by contradiction, by assuming that a typable program reduces to an error and then proving that said error is not typable, thus leading to a contradiction.

More in detail, assume that $P \longrightarrow_{\mathsf{A}}^{*} P'$ and that $\Gamma \vdash_{\emptyset} P \triangleright \Delta$. Then, assume that P' is an error (i.e., P' contains, up to structural congruence, a term Q that is the parallel composition of two k-processes that do not form a k-redex). Notice that the definition of $\asymp$ and Rule (T:QConc) prevents are important for the proof, as they will prevent the typing of errors. We proceed by a case analysis in Q. We only detail the case whenever (1) $Q = x\langle v\rangle.Q_1 \mid x\langle v'\rangle.Q_2$. All the other cases are similar.

If $Q = x\langle v\rangle.Q_1 \mid x\langle v'\rangle.Q_2$, then the linear environment Δ would need to contain $k :!U.T$, which can be split by Rule (T:QConc), however, the merging operation $\asymp$ is only defined for a session type and a message type, not two session types, thus reaching a contradiction. □

Next, we identify the set of $\mathsf{a}\pi$ programs induced by PI processes (cf. Def. 3.46) and typing. First, we have that as a consequence from Thm. 3.70, the following corollary holds:

Corollary 3.71. *If a PI process P is well-typed then for every Q such that $P \longrightarrow_{\mathsf{A}}^{*} Q$, Q is not an error.*

The previous corollary then allow us to define what we call *well-formed* programs:

Definition 3.72 (Well-Formed $\mathsf{a}\pi$ Programs). We say that a $\mathsf{a}\pi$ process P is well-formed if there exists a PI process P_0 such that P_0 is well-typed and $P_0 \longrightarrow_{\mathsf{A}}^{*} P$.

These well-formed $\mathsf{a}\pi$ programs will be focus of the translation in Ch. 8. We require well-formed programs to originate from a PI process (cf. Def. 3.46), ensuring that all the necessary queues are present. Furthermore, the well-typedness condition on the PI processes ensures typing of P by Thm. 3.70. The last property we prove in this section is a basic fact about well-typed programs. Intuitively, we say that it is possible to use the structural congruence in Def. 3.44 to write the $\mathsf{a}\pi$ program as a context $C[P, Q]$, where P is in $\mathsf{a}\pi^{\star}$ and Q only contains queues.

Lemma 3.73. *For every well-typed $\mathsf{a}\pi$ program P, it holds that:*

$$P \equiv_{\mathsf{A}} C[P', Q]$$

with $C[\cdot,\cdot] = (\boldsymbol{\nu}\widetilde{k})([\cdot]_1 \mid [\cdot]_2)$ (cf. Def. 3.45), $P' \in \mathsf{a}\pi^{\star}$, and Q containing only queues.

Proof. By Def. 3.72, there exists a PI process P_0 (cf. Def. 3.46) such that P_0 is well-typed and $P_0 \longrightarrow_{\mathsf{A}}^{*} P$. By Thm. 3.69, P is well-typed and by Thm. 3.70, it is not an error (Def. 3.66). Thus, we can apply the structural congruence ($\equiv_{\mathsf{A}}$) rules for scope extrusion and commutativity of the parallel composition operator to show that $P \equiv_{\mathsf{A}} C[P', Q]$, with $P' \in \mathsf{a}\pi^{\star}$ and Q only containing queues. □

Before introducing examples, we present an useful notation for compactly writing PI processes. This notation will be used throughout the dissertation.

Notation 3.74. Let $P \equiv_{\mathsf{A}} (\boldsymbol{\nu}\widetilde{x})(\prod_{i\in\{1,\ldots,n\}} Q_i \mid \prod_{k_j\in\widetilde{k}} k_j[i:\epsilon, o:\epsilon])$ be PI process (cf. Def. 3.46) with variables $\widetilde{k} = \widetilde{x}\widetilde{\overline{x}}$. We write P as $C[\prod_{i\in\{1,\ldots,n\}} Q_i, \mathcal{K}(\widetilde{k})]$, where $\mathcal{K}(\widetilde{k}) = \prod_{k_j\in\widetilde{k}} k_j[i:\epsilon, o:\epsilon]$.

We conclude this section by presenting examples that provide more intuitions about the type system and the semantics of aπ. For this example, we omit trailing inaction processes and types (i.e., $\mathbf{0}$, end), and only write them when necessary.

Example 3.75. Let us recall the process P_{11} in (3.8), rewritten using Not. 3.74 and the considerations above:

$$P_{11} = (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(x\langle \mathtt{ff}\rangle \mid \overline{x}(z).(y\langle \mathtt{tt}\rangle \mid \overline{y}(z')) \mid \mathcal{K}(\widetilde{w}))$$

with $\widetilde{w} = x\overline{x}y\overline{y}$. The typing derivation is given below. Notice that, as with the examples for $\pi^{\natural}_{\mathsf{OR}}$ and $\pi^{\natural}_{\mathsf{R}}$, we only show the derivation sub-trees for processes, and omit the ones for variables:

$$\text{(T:Res)}\times 2\;\dfrac{\text{(T:QConc)}\;\dfrac{\text{(T:QConc)}\;\dfrac{\dfrac{D_1}{\emptyset \vdash x\langle \mathtt{ff}\rangle \triangleright x:!\mathsf{bool}} \qquad \dfrac{D_2}{\emptyset \vdash R \triangleright \Delta}}{\emptyset \vdash x\langle \mathtt{ff}\rangle \mid R \triangleright x:!\mathsf{bool}, \Delta} \qquad D_3}{\emptyset \vdash_{\widetilde{w}} x\langle \mathtt{ff}\rangle.\mathbf{0} \mid R \mid \mathcal{K}(\widetilde{w}) \triangleright x:!\mathsf{bool}, \underbrace{\overline{x}:?\mathsf{bool}, y:!\mathsf{bool}, \overline{y}:?\mathsf{bool}}_{\Delta}}}{\emptyset \vdash (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(x\langle \mathtt{ff}\rangle \mid \underbrace{\overline{x}(z).(y\langle \mathtt{tt}\rangle \mid \overline{y}(z'))}_{R} \mid \mathcal{K}(\widetilde{w})) \triangleright \emptyset}$$

where D_1, D_2, and D_3 are given below (in that order):

$$\text{(T:Send)}\;\dfrac{\text{(T:Nil)}\;\dfrac{\mathsf{end}(x:\mathsf{end})}{\emptyset \vdash \mathbf{0} \triangleright x:\mathsf{end}}}{\emptyset \vdash x\langle \mathtt{ff}\rangle \triangleright x:!\mathsf{bool}}$$

$$\text{(T:Rcv)}\;\dfrac{\text{(T:QConc)}\;\dfrac{\dfrac{\text{(T:Send)}, \text{(T:Nil)}}{z:\mathsf{bool} \vdash y\langle \mathtt{tt}\rangle.\mathbf{0} \triangleright \Delta'} \qquad \dfrac{\text{(T:Rcv)}, \text{(T:Nil)}}{z:\mathsf{bool} \vdash \overline{y}(z') \triangleright \overline{y}:?\mathsf{bool}}}{z:\mathsf{bool} \vdash y\langle \mathtt{tt}\rangle \mid \overline{y}(z') \triangleright \underbrace{\overline{x}:\mathsf{end}, y:!\mathsf{bool}}_{\Delta'}, \overline{y}:?\mathsf{bool}}}{\emptyset \vdash \overline{x}(z).(y\langle \mathtt{tt}\rangle \mid \overline{y}(z')) \triangleright x:!\mathsf{bool}, \Delta}$$

$$\text{(Q:Conc)}\;\dfrac{\begin{array}{c}\text{(T:IQEnd)}\\ \dfrac{\mathsf{end}(\emptyset)}{\emptyset \vdash_x \mathcal{K}(x) \triangleright x:\emptyset}\end{array} \quad \begin{array}{c}\text{(T:IQEnd)}\\ \dfrac{\mathsf{end}(\emptyset)}{\emptyset \vdash_{\overline{x}} \mathcal{K}(\overline{x}) \triangleright \overline{x}:\emptyset}\end{array} \quad \begin{array}{c}\text{(T:IQEnd)}\\ \dfrac{\mathsf{end}(\emptyset)}{\emptyset \vdash_y \mathcal{K}(y) \triangleright y:\emptyset}\end{array} \quad \begin{array}{c}\text{(T:IQEnd)}\\ \dfrac{\mathsf{end}(\emptyset)}{\emptyset \vdash_{\overline{y}} \mathcal{K}(\overline{y}) \triangleright \overline{y}:\emptyset}\end{array}}{\emptyset \vdash_{\widetilde{w}} \mathcal{K}(\widetilde{w}) \triangleright x:\emptyset, \overline{x}:\emptyset, y:\emptyset, \overline{y}:\emptyset}$$

The first thing we notice is that the asynchronous nature of aπ makes these derivation trees longer than the ones presented in § 3.1.1 and § 3.1.2. Moreover, we can notice that a general typing strategy consists in grouping queues and processes separately to apply the corresponding rules. An interesting part of the derivation is the subtree D_3, which shows how queues are typed and how Σ (in this case $\widetilde{w}$) is used to ensure that queues appear only once. Intuitively, the hypothesis $\Sigma_1 \cap \Sigma_2 = \emptyset$ in Rule (T:QConc) enable us to split the queue names in $\widetilde{w}$ evenly, assigning each element to a single queue, which cannot appear repeated in the process. $\triangle$

3.2.7 *A Big-Step Semantics for* aπ

We present a big-step semantics that will be useful for proving the correctness of the translation in Ch. 8. Before presenting the big-step semantics for aπ, we will provide some informal intuitions behind the translation presented in Ch. 8. These intuitions do not aim to provide the full picture of the translation; further explanations will be given in its corresponding chapter.

Remark 3.76. Notice that the target language for the translation in Ch. 8 will be introduced in § 3.4 and is called RMLq (a queue-based variant of ReactiveML).

Remarkably, in aπ queues are *local* to processes that write or read from them. Therefore, the translation in Ch. 8, assumes that writing and reading operations are instantaneous. Hence, in a single big-step reduction, the target RMLq expression executes several reading and writing operations. Moreover, we follow the design decision presented in § 3.1.2, and only allow for outermost synchronizations in the big-step semantics for aπ.

We start by defining some auxiliary terminology. In the sequel, we assume that a value v can be *marked* to differentiate it from other values.

Notation 3.77. Let v be an aπ value. We shall write $\widehat{v}$ to denote that v is *marked*. Also, we write $\neg\widehat{v}$ when referring explicitly to unmarked values. Moreover, for any process P which contain queues with marked values, we will say that P is a marked process.

Marked values denote a "break point" in the input queue of a given endpoint, to denote the number of writing and reading operations that can be done in a single step. This goes in accordance with our decision of considering writing and reading operations on queues as instantaneous. In this regard, markings signal the end of an "instant" in aπ. Informally, we achieve this by marking the first value available in the next instant. For example, in a queue $x[i : \widetilde{m_1} \cdot \widehat{v} \cdot \widetilde{m_2}, o : \widetilde{m_3}]$, values $\widehat{v} \cdot \widetilde{m_2}$ will only be available for the process in the next big-step reduction, not in the current one.

The big-step semantics of aπ is given by two "sub-semantics": (1) a semantics that deals with synchronization between queues and (2) a semantics focused solely on the interaction between queues and processes. Our goal is to build a reduction relation that simulates in a single step all the executions that an RMLq big-step reduction can execute.

Definition 3.78 (Big-Step Queue Reduction for aπ). Let $\rightharpoondown$ denote the semantics obtained by replacing Rule $\lfloor \text{Com} \rfloor$ in Fig. 3.5 with Rule:

$$\lfloor \text{ComM} \rfloor$$
$$x[i : \widetilde{m_1}, o : v \cdot \widetilde{m_2}] \mid \overline{x}[i : \widetilde{m_1}, o : \widetilde{m_2}] \rightharpoondown x[i : \widetilde{m_1}, o : \widetilde{m_2}] \mid \overline{x}[i : \widetilde{m_1} \cdot \widehat{v}, o : \widetilde{m_2}]$$

Then, let

$$P_0 = C[P, Q_1 \mid \ldots \mid Q_n \mid Q_{n+1} \mid \ldots \mid Q_m]$$

with $1 \leq n \leq m$ be a well-typed program in aπ. The big-step queue reduction of process P_0, denoted $P_0 \rightarrowtail Q$ is given by:

$$\lfloor \text{QBStep} \rfloor \quad \frac{\forall i \in \{1, \ldots n\} \exists j \in \{1, \ldots n\}.(Q_i \mid Q_j \rightharpoondown Q_i' \mid Q_j') \;\; \forall i, j \in \{n+1, \ldots m\}.(Q_i \mid Q_j \not\rightharpoondown)}{C[P, Q_1 \mid \ldots \mid Q_n \mid Q_{n+1} \mid \ldots \mid Q_m] \rightarrowtail C[P, Q_1' \mid \ldots \mid Q_n' \mid Q_{n+1} \mid \ldots \mid Q_m]}$$

Above, Rule $\lfloor\text{ComM}\rfloor$ marks the values after synchronization from queue used for outputs and an input queue. In accordance to our translation, Rule $\lfloor\text{QBStep}\rfloor$ only allows for a single synchronization (with Rule $\lfloor\text{ComM}\rfloor$) between the queues corresponding to complementary endpoints. Notice that whenever a synchronization is not possible, queues are unchanged. Below, we tweak the aπ semantics in Fig. 3.5 to account for marks.

Definition 3.79 (Big-Step Process Semantics). Given a well-formed aπ program P, we write $P \rightharpoonup Q$, to denote the semantics generated by the rules in Fig. 3.5, without Rule $\lfloor\text{Com}\rfloor$ and replacing Rules $\lfloor\text{Recv}\rfloor$ and $\lfloor\text{Bra}\rfloor$ with:

$$\lfloor\text{RecvM}\rfloor\ x(y).P \mid x[i : \neg\widehat{v} \cdot \widetilde{m_1}, o : \widetilde{m_2}] \rightharpoonup P\{^{v}\!/xy\} \mid x[i : \widetilde{m_1}, o : \widetilde{m_2}]$$
$$\lfloor\text{BraM}\rfloor\ x \triangleright \{l_i : P_i\}_{i\in I} \mid x[i : \neg\widehat{l_j} \cdot \widetilde{m_1}, o : \widetilde{m_2}] \rightharpoonup P_j \mid x[i : \widetilde{m_1}, o : \widetilde{m_2}] \quad (j \in I)$$

We write $\rightharpoonup^*$ to denote the reflexive transitive closure of $\rightharpoonup$.

The second layer of the big-step semantics for aπ deals with the interaction between a$\pi^\star$ processes and queues. Rule $\lfloor\text{Com}\rfloor$ is excluded from the definition of $\ggg\!\!\rightarrow$, as no further queue synchronizations can occur. The big-step semantics for aπ is then presented as the composition of $\rightarrowtail$ and $\rightharpoonup$. First, we introduce a function for removing the marks in queues.

Definition 3.80 (Removing Marks). Let P be a aπ process. The function $\mathtt{unm}(Q)$ is defined inductively as:

$$\mathtt{unm}(k_i[i : \widetilde{m_1} \cdot \widehat{v} \cdot \widetilde{m_2}, o : \widetilde{m_3}]) \stackrel{\text{def}}{=} k_i[i : \widetilde{m_1} \cdot v \cdot \widetilde{m_2}, o : \widetilde{m_3}]$$
$$\mathtt{unm}(P_1 \mid P_2) \stackrel{\text{def}}{=} \mathtt{unm}(P_1) \mid \mathtt{unm}(P_2) \quad \mathtt{unm}((\boldsymbol{\nu} x)P) \stackrel{\text{def}}{=} (\boldsymbol{\nu} x)\mathtt{unm}(P)$$

and the identity for every other case of P.

Definition 3.81 (Big-Step Semantics for aπ). Let $P = C[P_1 \mid \ldots \mid P_n, Q_1 \mid \ldots \mid Q_m]$, with $n, m \geq 1$ be a well-formed aπ program. The big-step semantics for aπ is generated by the following rule:

$$\lfloor\text{BStep}\rfloor\ \frac{\begin{array}{c} C[P_1 \mid \ldots \mid P_n, Q_1 \mid \ldots \mid Q_m] \rightarrowtail C[P_1 \mid \ldots \mid P_n, Q'_1 \mid \ldots \mid Q'_m] \\ C[P_1 \mid \ldots \mid P_n, Q'_1 \mid \ldots \mid Q'_n] \rightharpoonup^* C[P'_1 \mid \ldots \mid P'_n, Q''_1 \mid \ldots \mid Q''_m] \not\rightharpoonup \end{array}}{C[P_1 \mid \ldots \mid P_n, Q_1 \mid \ldots \mid Q_m] \ggg\!\!\rightarrow C[P'_1 \mid \ldots \mid P'_n, \mathtt{unm}(Q''_1 \mid \ldots \mid Q''_m)]}$$

An important observation from Rule $\lfloor\text{BStep}\rfloor$ is that the marks are removed at the end of every big-step reduction. This effectively means that there is only one marked value in each synchronizing queue at every big-step reduction. This fact accounts for the correctness of the $\mathtt{unm}(\cdot)$ function. To clarify the semantics we use the following example.

Example 3.82. Consider the following aπ process

$$\begin{array}{c} C[Q, \mathcal{K}(x\overline{x})] = (\boldsymbol{\nu} x)(x\langle v_1\rangle.x\langle v_2\rangle.x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid \\ x[i : \epsilon, o : \epsilon] \mid \overline{x}[i : \epsilon, o : \epsilon]) \end{array} \tag{3.2}$$

We analyze the first big-step reduction below:

$$\begin{aligned} C[Q, \mathcal{K}(x\overline{x})] \gg\!\!\!\rightarrow\ & (\boldsymbol{\nu} x)(x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid \\ & \qquad x[i:\epsilon, o: v_1\cdot v_2] \mid \overline{x}[i:\epsilon, o:\epsilon]) \\ =\ & Q_1 \end{aligned}$$

Above, observe that $C[P, \mathcal{K}(x\overline{x})] \rightarrowtail C[P, \mathcal{K}(x\overline{x})]$ since all the queues are empty and no synchronization is possible. Moreover, observe that:

$$\begin{aligned} C[Q, \mathcal{K}(x\overline{x})] \rightharpoonup^*\ & (\boldsymbol{\nu} x)(x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid \\ & \qquad x[i:\epsilon, o: v_1\cdot v_2] \mid \overline{x}[i:\epsilon, o:\epsilon]) \\ \not\rightharpoonup =\ & Q_1 \end{aligned}$$

The reduction above occurs by applying Rule $\lfloor\textsc{Send}\rfloor$ twice to obtain Q_1 and $Q_1 \not\rightharpoonup$. The complete sequence of big-step reductions is given below:

$$\begin{aligned} Q_1 &\gg\!\!\!\rightarrow (\boldsymbol{\nu} x)(x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid \mathsf{unm}(x[i:\epsilon, o: v_2] \mid \\ &\qquad \overline{x}[i:\widehat{v_1}, o:\epsilon])) = Q_2 \\ Q_2 &\gg\!\!\!\rightarrow (\boldsymbol{\nu} x)(x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid \mathsf{unm}(x[i:\epsilon, o:\epsilon] \mid \\ &\qquad \overline{x}[i:\widehat{v_2}, o:\epsilon])) = Q_3 \\ Q_3 &\gg\!\!\!\rightarrow (\boldsymbol{\nu} x)(x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_4).\mathbf{0} \mid \mathsf{unm}(x[i:\epsilon, o:\epsilon] \mid \overline{x}[i:\epsilon, o:v_2])) = Q_4 \\ Q_4 &\gg\!\!\!\rightarrow (\boldsymbol{\nu} x)(x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_4).\mathbf{0} \mid \mathsf{unm}(x[i:\widehat{v_2}, o:\epsilon] \mid \overline{x}[i:\epsilon, o:\epsilon])) = Q_5 \\ Q_5 &\gg\!\!\!\rightarrow (\boldsymbol{\nu} x)(\mathbf{0} \mid \overline{x}(y_4).\mathbf{0} \mid \mathsf{unm}(x[i:\epsilon, o:v_3] \mid \overline{x}[i:\epsilon, o:\epsilon])) = Q_6 \\ Q_6 &\gg\!\!\!\rightarrow (\boldsymbol{\nu} x)(\mathbf{0} \mid \overline{x}(y_4).\mathbf{0} \mid \mathsf{unm}(x[i:\epsilon, o:\epsilon] \mid \overline{x}[i:\epsilon, o:\widehat{v_3}])) = Q_7 \gg\!\!\!\rightarrow \mathbf{0} \end{aligned}$$

Above, we have left the marking of values explicit to better observe where new instants begin and where $\rightharpoonup$ reductions cannot continue. In the big-step reduction from Q_1 to Q_2, we apply Rule $\lfloor\textsc{ComM}\rfloor$ once, to communicate v_2 from the output queue of x to the input queue of $\overline{x}$ to obtain Q_2, and the value gets marked. Hence, $Q_2 \not\rightharpoonup$ and no further reductions are possible.

The most interesting big-step reduction is perhaps the one from Q_2 to Q_3. In it, we observe how the two types of reductions ($\rightarrowtail$ and $\rightharpoonup$) are composed. First notice that:

$$\begin{aligned} Q_2 \rightarrowtail\ & (\boldsymbol{\nu} x)(x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid x[i:\epsilon, o:\epsilon] \mid \\ & \qquad \overline{x}[i: v_1\cdot\widehat{v_2}, o:\epsilon]) = Q_2' \end{aligned}$$

Above, we can observe that the new marked value is v_2. This means that in the current instant the only value available is v_1. Hence,

$$Q_2' \rightharpoonup^* (\boldsymbol{\nu} x)(x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid x[i:\epsilon, o:\epsilon] \mid \overline{x}[i:\widehat{v_2}, o:\epsilon]) \not\rightharpoonup$$

All the other steps continues as expected. The most important thing to notice is that the big-step reduction sequence presented above coincides with the following aπ reduction sequence:

$$C[Q, \mathcal{K}(x\overline{x})] \longrightarrow_{\mathsf{A}}^2 C[Q_1, x[i:\epsilon, o: v_1\cdot v_2] \mid \overline{x}[i:\epsilon, o:\epsilon]]$$

$$\begin{aligned}
&\longrightarrow_{\mathsf{A}} C[Q_2, x[i:\epsilon, o:v_2] \mid \overline{x}[i:v_1, o:\epsilon]] \\
&\longrightarrow_{\mathsf{A}}^2 C[Q_3, x[i:\epsilon, o:\epsilon] \mid \overline{x}[i:v_2, o:\epsilon]] \\
&\longrightarrow_{\mathsf{A}}^2 C[Q_4, x[i:\epsilon, o:\epsilon] \mid \overline{x}[i:\epsilon, o:v_2]] \\
&\longrightarrow_{\mathsf{A}} C[Q_5, x[i:v_2, o:\epsilon] \mid \overline{x}[i:\epsilon, o:\epsilon]] \\
&\longrightarrow_{\mathsf{A}}^2 C[Q_6, x[i:\epsilon, o:v_3] \mid \overline{x}[i:\epsilon, o:\epsilon]] \\
&\longrightarrow_{\mathsf{A}} C[Q_7, x[i:\epsilon, o:\epsilon] \mid \overline{x}[i:v_3, o:\epsilon]] \\
&\longrightarrow_{\mathsf{A}} C[\mathbf{0}, x[i:\epsilon, o:v_3] \mid \overline{x}[i:\epsilon, o:\epsilon]]
\end{aligned}$$

where every process Q_i $(i \in \{1,2,3,4,5,6,7\})$ is given below:

$$\begin{aligned}
&Q_1 = Q_2 = x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \\
&Q_3 = x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \\
&Q_4 = Q_5 = x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_4).\mathbf{0} \\
&Q_6 = Q_7 = \mathbf{0} \mid \overline{x}(y_4).\mathbf{0}
\end{aligned}$$

△

3.2.8 Semantic Correspondence

We now proceed to prove our semantic correspondence result. First, we prove that for every well-typed aπ program, the big-step reductions always yield unmarked processes. Then, we prove that there exists a semantic correspondence between the big-step process semantics (cf. Def. 3.79) and the reduction semantics in Fig. 3.5. Then, we prove a single-step semantic correspondence, which clarifies the relation between $\ggg\!\!\rightarrow$ and $\longrightarrow_{\mathsf{A}}$. This result is then generalized to multiple steps.

Lemma 3.83. *Let P be a well-typed* aπ *program. If $P \ggg\!\!\rightarrow^* Q$ then $Q = \mathtt{unm}(Q')$ for some Q'.*

Proof. This statement follows directly by the definition of the big-step semantics of aπ (cf. Def. 3.81). □

Lemma 3.84. *For every* aπ *process P with marked values, the following holds:*

1. *If $P \rightharpoondown^* Q$ then $\mathtt{unm}(P) \longrightarrow_{\mathsf{A}}^* \mathtt{unm}(Q)$.*

2. *If $P \rightharpoonup^* Q$ then $\mathtt{unm}(P) \longrightarrow_{\mathsf{A}}^* \mathtt{unm}(Q)$.*

Proof. We prove each item:

1. By induction on the reduction $P \rightharpoondown^* Q$ and a case analysis on the last applied rule. Notice that $\rightharpoondown$ and $\longrightarrow_{\mathsf{A}}$ only differ in the rule for communication between queues: Rule $\lfloor\text{ComM}\rfloor$ for $\rightharpoondown$ (cf. Def. 3.78) and Rule $\lfloor\text{Com}\rfloor$ for $\longrightarrow_{\mathsf{A}}$. Observe that the only difference is the marking of the value being added in the input queue. From the previous observation, it follows that whenever $P \rightharpoondown Q$ with Rule $\lfloor\text{ComM}\rfloor$ then $P \longrightarrow_{\mathsf{A}} Q$ with Rule $\lfloor\text{Com}\rfloor$.

2. The proof follows by induction on the length n of reduction $P \rightharpoonup^* Q$ and case analysis on the last applied Rule from Def. 3.79. The base case and inductive step are immediate.

□

Lemma 3.85 (Single-Step Semantic Correspondence). *Let P be a well-formed* aπ *program. Then, the following holds:*

1. *If $P \ggg\!\!\rightarrow Q$ then $P \longrightarrow_{\mathsf{A}}^* Q$.*
2. *If $P \longrightarrow_{\mathsf{A}} Q$ then there exists R such that $P \ggg\!\!\rightarrow R$ and $Q \longrightarrow_{\mathsf{A}}^* R$.*

Proof. We prove both numerals. The first one can be derived from the previous definitions and the second one follows from Thm. 3.69 and Lem. 3.73. For details see App. A.4. □

We conclude this section by proving a multi-step semantic correspondence (cf. Def. 2.9). The statement below is split int two parts: first, we ensure that the big-step semantics of aπ only includes behavior that is also exhibited by its reduction semantics. The second part of the statement ensures that the all the reductions exhibited by P are captured by some big-step reduction.

Lemma 3.86 (Semantic Correspondence). *Let P be a well-formed* aπ *program. Then, the following holds:*

1. *If $P \ggg\!\!\rightarrow^* Q$ then $P \longrightarrow_{\mathsf{A}}^* Q$.*
2. *If $P \longrightarrow_{\mathsf{A}}^* Q$ then there exists R such that $P \ggg\!\!\rightarrow^* R$ and $Q \longrightarrow_{\mathsf{A}}^* R$.*

Proof. We prove both items by induction on the length of the reduction. The proofs are similar to the ones presented in Lem. 3.85. □

3.3 Extending `lcc` with Private Information (`lcc`$^{\mathsf{p}}$)

We introduce `lcc`$^{\mathsf{p}}$, an extension of `lcc` in which abstractions are generalized with *local information*. The idea is to use `lcc`$^{\mathsf{p}}$ to model systems where private information is important (e.g., session establishment protocols with cryptography). These developments are based on the work presented in [HL09] for `utcc` [OV08a].

In § 3.3.1 we motivate this extension via examples. Next, we present the syntax and semantics in § 3.3.2 and the type system in § 3.3.3. We conclude by presenting properties ensured by types in § 3.3.4.

3.3.1 *Motivation*

The encoding of $\pi_{\mathsf{OR}}^{\sharp}$ into `lcc` that will be presented in Ch. 4 relies critically on `lcc` abstractions (i.e., universal quantifier) to represent intra-session communication in π (including scope extrusions) and their associated continuations. However, abstractions in `lcc` turn out to be overly powerful for modeling scope extrusion, in the sense that they can also represent combinations of name-passing and restriction not possible in a π-calculus such as $\pi_{\mathsf{OR}}^{\sharp}$.

In particular, this issue arises because the local construct in `lcc` (i.e., the existential quantifier) allows abstractions to obtain information that should be hidden. To illustrate this, we consider the following example:

Example 3.87. Let Q_2 be the `lcc` process defined below:

$$Q_2 = \exists x.\big(\overline{\varphi(x)} \parallel \forall y\big(\varphi(y) \rightarrow \overline{\mathtt{tt}}\big)\big) \parallel \mathit{Spy} \tag{3.3}$$

where *Spy* is defined as:

$$\mathit{Spy} = \forall z\big(\varphi(z) \rightarrow \overline{\mathtt{ff}}\big)$$

Supposing that we would like the synchronization inside the local construct private (something that can be done in $\pi_{\mathsf{OR}}^{\sharp}$), we can notice that Q above fails in this regard—the semantics of `lcc` in Fig. 2.5 allow *Spy* to access $\varphi(x)$ in the store:

$$Q_2 \longrightarrow_1 \exists x.\big(\forall y\big(\varphi(y) \rightarrow \overline{\mathtt{tt}}\big) \parallel \overline{\mathtt{ff}}\big)$$

To intuitively understand how the privacy of the restriction construct behaves in $\pi_{\mathsf{OR}}^{\sharp}$, compared to the one in `lcc`, we consider the following $\pi_{\mathsf{OR}}^{\sharp}$ process:

$$P_{15} = (\boldsymbol{\nu} xy)(x\langle v\rangle.P_x \mid y(z).Q_y) \mid R \tag{3.4}$$

The first thing to notice is that the restriction operator $(\boldsymbol{\nu} xy)$ ensures that communication between endpoints x and y is private, i.e., they cannot be interfered by some external process. In particular, we have that the semantics of $\pi_{\mathsf{OR}}^{\sharp}$ (cf. Fig. 2.1) does not allow R to get a hold of v in the reduction:

$$P_{15} \longrightarrow (\boldsymbol{\nu} xy)(P_x \mid P_y\{v/z\}) \mid R$$

$\triangle$

The situation illustrated by the example contrasts with a similar scenario for `lcc`, given by process Q_2 (cf. (3.3)), as the semantics of `lcc` does allow an external abstraction to get information from inside the local construct. To address this issue we would like to limit the power of abstractions in `lcc` so we can preserve the nature and essential assumptions of the restriction operator in $\pi_{\mathsf{OR}}^{\sharp}$. Hence, we would like the privacy of session endpoints (inherited from the restriction operator) to be explicitly programmed at the declarative level of $\mathtt{lcc}^{\mathsf{p}}$ processes, relying on some extra mechanism to limit abstractions.

To this end, next we develop a simple *typing discipline* for $\mathtt{lcc}^{\mathsf{p}}$, built upon the approach in [HL09]—where the focus is in `utcc` and session-based concurrency is not addressed. Our type system admits only abstractions whose associated variables adhere to a precisely defined policy. Informally speaking, we shall classify/sort variables as either *unrestricted* (i.e., public) or *restricted* (i.e., privacy-sensitive), and decree that restricted variables are non-abstractable. This is a simple access control mechanism for $\mathtt{lcc}^{\mathsf{p}}$ abstractions. A well-typed $\mathtt{lcc}^{\mathsf{p}}$ process will then only contain abstractions of the form $\forall\widetilde{x}(d\,;\,e \rightarrow P)$ where e is a *secure pattern*, i.e., it respects the classification policy by not involving non-abstractable variables.

The type system is defined in general terms; one application is an encoding of π_{E} into $\texttt{lcc}^{\mathsf{p}}$—see § 3.1.3. In this case, the sorting policy applies to the predicates used to represent synchronizations. This way, we assume a refined signature in which $\varphi(\widetilde{x}; \widetilde{y})$ is a predicate with $\widetilde{x}$ restricted and $\widetilde{y}$ unrestricted. This allows us to distinguish process that try to access private variables, but are not allowed to do so (cf. Ex. 3.89).

3.3.2 Syntax and Semantics

Abstractions in $\texttt{lcc}$ act on global information posted in the constraint store. This may be an issue when specifying processes that use their local information to perform some public (observable) behavior. This is the case of session processes after a session has been established, which rely on (private) endpoints to communicate. Another relevant example of local information are the (private) keys used in protocols for secure communications.

To address this interplay between local information and public behavior, we consider $\texttt{lcc}^{\mathsf{p}}$, a variant of $\texttt{lcc}$ in which abstractions are generalized so as to account for local information. The syntax of $\texttt{lcc}^{\mathsf{p}}$ results from Def. 2.26 by replacing the abstraction operator $\forall\widetilde{x}(e \rightarrow P)$ (in the grammar for guards) with the following one:

$$\forall\widetilde{x}(d\,;\, e \rightarrow P)$$

The new element is constraint d, a piece of local information used jointly with e to trigger P. That is, d is used as additional resource in inferring e from the (global) constraint store; still, d is used locally, for it is not added to the store. This construct is a generalization in the sense that $\forall\widetilde{x}(e \rightarrow P)$ in $\texttt{lcc}$ corresponds to $\forall\widetilde{x}(\texttt{tt}\,;\, e \rightarrow P)$ in $\texttt{lcc}^{\mathsf{p}}$. The operational semantics of $\texttt{lcc}^{\mathsf{p}}$ formalizes these intuitions. It is defined by the LTS in Fig. 2.5, replacing Rule (C:Sync) with the following rule:

$$\lfloor\text{C:SyncLoc}\rfloor\ \frac{\begin{array}{c} c \otimes d \vdash \exists\widetilde{y}.(e\{\widetilde{t}/\widetilde{x}\} \otimes f) \quad \widetilde{y} \cap \mathsf{fv}(c,d,e,P) = \emptyset \\ \mathbf{mgc}\big(c \otimes d, \exists\widetilde{y}.(e\{\widetilde{t}/\widetilde{x}\} \otimes f)\big) \quad c \otimes d \vdash \texttt{ff} \implies c \vdash \texttt{ff} \end{array}}{\overline{c} \parallel \forall\widetilde{x}(d\,;\, e \rightarrow P) \longrightarrow_1 \exists\widetilde{y}.\,(P\{\widetilde{t}/\widetilde{x}\} \parallel \overline{f})}$$

In this new rule, premise $c \otimes d \vdash \texttt{ff} \implies c \vdash \texttt{ff}$ ensures that only local assumptions d which do not conflict with the information in the (global) constraint store c are allowed. All other notions and definitions for $\texttt{lcc}$ processes will carry over to $\texttt{lcc}^{\mathsf{p}}$. Next we define a simple type system for disciplining abstractions with local information in $\texttt{lcc}^{\mathsf{p}}$.

3.3.3 The Type System

The typing rules for secure patterns/processes are defined in Fig. 3.7. For simplicity, we assume that patterns are conjunctions of predicates applied to terms over the function signature. We consider two environments, Δ and Θ, where Δ is the set of variables used as restricted and Θ is the set of variables used as unrestricted. Empty environments are denoted '$\cdot$'.

As hinted at above, the objective of the type system is to identify $\texttt{lcc}^{\mathsf{p}}$ processes whose abstractions contain secure patterns. We consider three kinds of judgments:

$$\text{(L:P{\scriptsize RED})}\ \frac{}{res(\varphi(\widetilde{t_1};\widetilde{t_2})); unr(\widetilde{t_2}\,)\setminus var(\widetilde{t_1}) \vdash_{\bullet} \varphi(\widetilde{t_1};\widetilde{t_2})} \qquad \text{(L:T{\scriptsize RUE})}\ \frac{}{\cdot;\cdot \vdash_{\bullet} \mathtt{tt}} \qquad \text{(L:F{\scriptsize ALSE})}\ \frac{}{\cdot;\cdot \vdash_{\bullet} \mathtt{ff}}$$

$$\text{(L:A{\scriptsize SSOC-L})}\ \frac{\Delta;\Theta \vdash_{\bullet} (c\otimes d)\otimes e}{\Delta;\Theta \vdash_{\bullet} c\otimes (d\otimes e)} \qquad \text{(L:A{\scriptsize SSOC-R})}\ \frac{\Delta;\Theta \vdash_{\bullet} c\otimes (d\otimes e)}{\Delta;\Theta \vdash_{\bullet} (c\otimes d)\otimes e} \qquad \text{(L:C{\scriptsize OMM})}\ \frac{\Delta;\Theta \vdash_{\bullet} c\otimes d}{\Delta;\Theta \vdash_{\bullet} d\otimes c}$$

$$\text{(L:C{\scriptsize OMB})}\ \frac{\Delta_1;\Theta_1 \vdash_{\bullet} c \quad \Delta_2;\Theta_2 \vdash_{\bullet} d \quad (\Delta_1\cap\Theta_2) = (\Delta_2\cap\Theta_1) = \emptyset}{\Delta_1\cup\Delta_2;\Theta_1\cup\Theta_2 \vdash_{\bullet} c\otimes d}$$

$$\text{(L:E{\scriptsize XIST})}\ \frac{\Delta;\Theta \vdash_{\bullet} c}{\Delta;\Theta \vdash_{\bullet} \exists\widetilde{x}.c} \qquad \text{(L:B{\scriptsize ANG})}\ \frac{\Delta;\Theta \vdash_{\bullet} c}{\Delta;\Theta \vdash_{\bullet} !c} \qquad \text{(L:A{\scriptsize BS})}\ \frac{\vdash_{\diamond} P \quad \Delta;\Theta \vdash_{\bullet} c \quad \widetilde{x}\subseteq \Theta\setminus \mathsf{fv}(d)}{\vdash_{\mathbf{A}} \forall\widetilde{x}(d\,;\,c \rightarrow P)}$$

$$\text{(L:S{\scriptsize UM})}\ \frac{\vdash_{\mathbf{A}} G_1 \quad \vdash_{\mathbf{A}} G_2}{\vdash_{\mathbf{A}} G_1 + G_2} \qquad \text{(L:G{\scriptsize UARD})}\ \frac{\vdash_{\mathbf{A}} G}{\vdash_{\diamond} G}$$

$$\text{(L:T{\scriptsize ELL})}\ \frac{c\in\mathcal{C}}{\vdash_{\diamond} \overline{c}} \qquad \text{(L:P{\scriptsize AR})}\ \frac{\vdash_{\diamond} P_1 \quad \vdash_{\diamond} P_2}{\vdash_{\diamond} P_1 \parallel P_2} \qquad \text{(L:R{\scriptsize EPL})}\ \frac{\vdash_{\diamond} P}{\vdash_{\diamond} !P} \qquad \text{(L:L{\scriptsize OCAL})}\ \frac{\vdash_{\diamond} P}{\vdash_{\diamond} \exists\widetilde{x}.\,P}$$

Figure 3.7: Typing rules for $\mathtt{lcc}^{\mathsf{p}}$.

- Judgment $\Delta;\Theta \vdash_{\bullet} c$ concerns patterns: it says that pattern c is well-formed, under restricted variables Δ and unrestricted variables Θ.
- The judgment for guards (abstractions, non-deterministic choice) is denoted $\vdash_{\mathbf{A}} G$
- Finally, a well-typed process P is denoted by $\vdash_{\diamond} P$.

Recall that we write $\widetilde{t}$ to denote a vector of terms with its length given by $|\widetilde{t}|$. We will consider predicates of the form $\varphi(\widetilde{t_1};\widetilde{t_2})$ where, intuitively, $\widetilde{t_1}$ denotes restricted parameters and $\widetilde{t_2}$ denotes unrestricted parameters. We introduce some useful functions on terms:

Definition 3.88. Let x and t denote a variable and an arbitrary term, respectively. Functions $unr(t)$, $res(t)$, and $var(t)$ yield, respectively, the set of variables appearing unrestricted in t according to the sorting; the set of variables appearing restricted in t; and the set of all variables occurring in t. They are defined as:

$$\begin{array}{ll} unr(x) \overset{\mathsf{def}}{=} \{x\} & unr(\varphi(\widetilde{t_1};\widetilde{t_2})) \overset{\mathsf{def}}{=} unr(\widetilde{t_2}) \\ var(x) \overset{\mathsf{def}}{=} \{x\} & var(\varphi(\widetilde{t_1};\widetilde{t_2})) \overset{\mathsf{def}}{=} var(\widetilde{t_1}) \cup var(\widetilde{t_2}) \\ res(x) \overset{\mathsf{def}}{=} \{x\} & res(\varphi(\widetilde{t_1};\widetilde{t_2})) \overset{\mathsf{def}}{=} var(\widetilde{t_1}) \cup (res(\widetilde{t_2}) \setminus unr(\widetilde{t_2})) \end{array}$$

We assume that these functions extend to sequences of terms by letting

$$unr(\widetilde{t}) = \bigcup_{1\leq i\leq |\widetilde{t}|} unr(t_i)$$

and similarly for $res(\cdot)$ and $var(\cdot)$.

Notice that $var(t) = res(t) \cup unr(t)$. Also, $res(t) \cap unr(t)$ may be non-empty; in $\varphi(\widetilde{t_1}; \widetilde{t_2})$, terms in $\widetilde{t_2}$ could contain restricted variables (e.g., in nested predicates, which are not needed in our encodings).

We now comment on some of the rules in Fig. 3.7. Rule (L:Pred) decrees that all variables in $\widetilde{t_1}$ as well as the variables occurring restricted in $\widetilde{t_2}$ are restricted. The remaining variables are unrestricted. Rules (L:Assoc-l), (L:Assoc-R), and (L:Comm) define basic properties of constraint conjunctions. Rule (L:Comb) identifies the restricted and unrestricted variables in the pattern $c \otimes d$. The set of restricted variables for c must be disjoint from the set of unrestricted variables for d, and vice versa. This avoids treating restricted variables in c or d as unrestricted variables in $c \otimes d$. Typing rules for guards and processes are simple. Rule (L:Abs) says that abstraction $\forall \widetilde{x}(d\,;\, c \rightarrow P)$ is secure as long as variables $\widetilde{x}$ are unrestricted in the typing for c, and no variables in d occur in $\widetilde{x}$.

Example 3.89 (An Ill-Typed Process). As a simple illustration of our type discipline, consider the following process:

$$Q_3 = \forall x, y, w\big(\mathtt{tt}\,;\, \varphi(w; y) \otimes \varphi'(w, x; \epsilon) \rightarrow \overline{\varphi''(\epsilon; x, y)}\big) \tag{3.5}$$

Above, we assume that in constraint $\varphi(w; y)$ variable w is restricted, while y is unrestricted. Similarly, suppose that both w and x are restricted in $\varphi'(w, x; \epsilon)$, and that both x, y are unrestricted in $\varphi''(\epsilon; x, y)$. The constant ϵ is used to denote an empty sequence of variables; it is useful to remember which predicates do not have restricted (or unrestricted) variables.

Using Rule (L:Comb), we obtain that pattern $\mathsf{snd}(w, y) \otimes \{w{:}x\}$ has an unrestricted variable (y) and two restricted variables, w and x. We then infer that Q_3 is not typable because a typing derivation would need to perform an insecure abstraction on the restricted variable w. The tree for the failed derivation is as follows:

$$\dfrac{\dfrac{\varphi''(\epsilon; x, y) \in \mathcal{C}}{\vdash_{\diamond} \overline{\varphi''(\epsilon; x, y)}}\,\text{(L:Tell)} \qquad D_1 \qquad \dfrac{\text{The derivation fails here.}}{\{x, w, y\} \subseteq \{y\}}}{\vdash_{\mathbf{A}} \forall x, y, w\big(\mathtt{tt}\,;\, \varphi(w; y) \otimes \varphi'(w, x; \epsilon) \rightarrow \overline{\varphi''(\epsilon; x, y)}\big)}\,\text{(L:Abs)}$$

with D_1:

$$\dfrac{\{w\}; \{y\} \vdash_{\bullet} \varphi(w; y) \qquad \{w, x\}; \emptyset \vdash_{\bullet} \varphi'(w, x; \epsilon)}{\{w, x\}; \{y\} \vdash_{\bullet} \varphi(w; y) \otimes \varphi'(w, x; \epsilon)}\,\text{(L:Comb)}$$

Notice that the derivation fails since predicate $\{x, w, y\} \subseteq \{y\}$ does not hold. △

3.3.4 Typing Properties

Before presenting the main results of the type system above, we must define the notion of *substitution* for $\mathtt{lcc}^{\mathsf{p}}$. We do this below:

Definition 3.90 (Substitution). Given terms $\widetilde{t} = t_1, \ldots, t_n$ and process variables $\widetilde{x} = x_1, \ldots, x_n$, the application of a substitution to a constraint, guard and process,

$$\mathtt{tt}\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} \mathtt{tt} \quad (!c)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} !(c\{\widetilde{t}/\widetilde{x}\}) \quad \mathtt{ff}\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} \mathtt{ff}$$

$$\varphi(\widetilde{u};\widetilde{v})\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} \varphi(\widetilde{u}\{\widetilde{t}/\widetilde{x}\};\widetilde{v}\{\widetilde{t}/\widetilde{x}\})$$

$$(c \otimes d)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} c\{\widetilde{t}/\widetilde{x}\} \otimes d\{\widetilde{t}/\widetilde{x}\} \quad (\exists \widetilde{y}.c)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} \exists \widetilde{y}.c\{\widetilde{t}/\widetilde{x}\} \text{ if } (\widetilde{y} \cap \widetilde{x} = \emptyset)$$

$$\forall \widetilde{y}(d\,;\, c \rightarrow P)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} \forall \widetilde{y}(d\{\widetilde{t}/\widetilde{x}\}\,;\, c\{\widetilde{t}/\widetilde{x}\} \rightarrow P\{\widetilde{t}/\widetilde{x}\}) \text{ if } (\widetilde{y} \cap \widetilde{x} = \emptyset)$$

$$(G_1 + G_2)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} G_1\{\widetilde{t}/\widetilde{x}\} + G_2\{\widetilde{t}/\widetilde{x}\}$$

$$(\overline{c})\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} \overline{c\{\widetilde{t}/\widetilde{x}\}} \quad (P \parallel Q)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} P\{\widetilde{t}/\widetilde{x}\} \parallel Q\{\widetilde{t}/\widetilde{x}\}$$

$$(\exists \widetilde{y}.\, P) \stackrel{\mathsf{def}}{=} \exists \widetilde{y}.\, P\{\widetilde{t}/\widetilde{x}\} \text{ if } (\widetilde{y} \cap \widetilde{x} = \emptyset) \quad (!\, P)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} !\,(P\{\widetilde{t}/\widetilde{x}\}) \quad (G)\{\widetilde{t}/\widetilde{x}\} \stackrel{\mathsf{def}}{=} G\{\widetilde{t}/\widetilde{x}\}$$

Figure 3.8: Substitution in $\mathtt{lcc}^{\mathsf{p}}$.

denoted respectively $c\{\widetilde{t}/\widetilde{x}\}$, $G\{\widetilde{t}/\widetilde{x}\}$, and $P\{\widetilde{t}/\widetilde{x}\}$, is inductively defined on the structure of constraints, guards and process as in Fig. 3.8.

Below, we present the main results regarding the type system: subject congruence, substitution lemma, and subject reduction. As with all the other process calculi we have presented, subject congruence shows that typing is preserved by the structural congruence in Def. 2.28. Similarly, subject reduction ensures that the semantics preserve typing.

Lemma 3.91 (Subject Congruence). *If $P \equiv Q$ and $\vdash_\diamond P$, then $\vdash_\diamond Q$.*

Proof. By a case analysis on $P \equiv Q$ (cf. Def. 2.28). Since congruences are symmetric, we need to prove for both $P \equiv Q$ and $Q \equiv P$. There are eight cases. For details see App. A.5. ☐

Proposition 3.92 (Substitution). *Let P and t be a process and a term, respectively. If $\vdash_\diamond P$ then $\vdash_\diamond P\{t/x\}$.*

Proof. By induction on the structure of P. Interesting cases are when $P = \overline{c}$ and $P = \forall \widetilde{y}(d\,;\, e \rightarrow P')$, for some $\widetilde{y}$, d, e, and P'; other cases are straightforward.

- Case $P = \overline{c}$: By the well-typedness assumption we infer that $c \in \mathcal{C}$, which immediately implies that $c\{t/x\} \in \mathcal{C}$ and so we conclude using (L:Tell).
- Case $P = \forall \widetilde{y}(d\,;\, e \rightarrow P')$: By the well-typedness assumption we infer that P' is well-typed, that $\Delta; \Theta \vdash_\bullet e$, and that $\widetilde{y} \subseteq \Theta \backslash \mathsf{fv}(d)$. Since universal and existential quantifiers are binders, occurrences of variables in $\widetilde{y}$ are not affected by $\{t/x\}$; this in particular rules out the possibility of renaming an unrestricted variable into a restricted one. The substitution thus only affects free variables (not in $\widetilde{y}$) and the thesis follows.

☐

Theorem 3.93 (Subject Reduction). *If $P \xrightarrow{\alpha}_1 Q$ and $\vdash_\diamond P$ then $\vdash_\diamond Q$.*

Proof. By a case analysis on the transition rule applied (cf. Fig. 2.5). There are eight cases. For details see App. A.5. □

3.4 Queue-Based ReactiveML (RMLq)

In this section we present RMLq, a variant of ReactiveML that uses *queues* to store and access values across time units. These queues store values that can be read by expressions at any given time. This extension was considered to bring ReactiveML closer to the idea of asynchrony presented in aπ (§ 3.2).

In § 3.4.1 we motivate the extension with some examples. Then, in § 3.4.2 we formally introduce the syntax and semantics for the calculus and provide some examples. Finally, we conclude by introducing some equivalences for RMLq configurations in § 3.4.3.

3.4.1 Motivation

Intuitively, RMLq was designed to be the target language for a translation of aπ (cf. § 3.2). Considering this, we claim that the key challenge of encoding aπ in a synchronous language such as ReactiveML lies on the fact that signals (and the values they contain) are *ephemeral*, i.e., they are deleted at the end of an instant. This fact contrasts with the queues in aπ, which can be accessed at any time during execution, without fear of losing messages. Thus, one must define a mechanism to make values in queues persistent over time. Before presenting our solution, we discuss on the way emitted values are stored during a ReactiveML big-step reduction.

Intuitively, ReactiveML uses event sets (Def. 2.38), to store the values emitted by a signal x during a single big-step reduction. These events are reset every time a new big-step reduction occurs. Hence, it is not possible to access values emitted during a previous instant. Therefore, it is not possible to use these event sets to store persistent values; a feature needed for modeling the asynchrony present in aπ. It can be argued that ReactiveML is not natively equipped with the necessary features to persistent emissions.

Considering the previous caveats, we have decided to focus on the way values are stored in memory from a more computational perspective. Since ReactiveML is built on top of OCaml, we have decided to make use of the more operational features of the programming language and consider a persistent memory that stores transmitted values in queues. Thus, RMLq becomes a variant of ReactiveML enriched with explicit states. This means, that RMLq processes in parallel can also synchronize by putting and popping values from queues. Formally, programs (i.e., executable expressions) are represented in RMLq by *configurations* $K = \langle e \diamond \Sigma \rangle$. In K, e represents a ReactiveML expression and Σ is a set of queues that can be modified by the execution of e.

Notice that having a explicit state and being able to synchronize values from queues is coherent with the semantics of aπ, where processes can interact with their queues, which are then in charge of synchronization. Then, it can be argued that by using configurations we bring RMLq closer to aπ, which in turn allows for tighter encodability results; this is a sought feature in our work.

To introduce RMLq, consider the following configuration, in which a ReactiveML expression stores a value in a queue to be emitted at a later instant:

$$\begin{aligned} K_1 = \langle &\texttt{put}\ q_1\ \texttt{tt} \parallel \\ &\texttt{pause}; \\ &\texttt{signal}\ x\ \texttt{in let}\ y = \texttt{pop}\ q_1\ \texttt{in emit}\ x\ y; \texttt{pause}; \texttt{emit}\ x\ y \diamond q_1 : \epsilon\rangle \end{aligned} \tag{3.6}$$

In K_1 we have a configuration with a ReactiveML expression which first puts value tt in queue q_1—which is empty at the start of execution. At the next instant, the expression declares a signal x and obtains the value stored in q_1 to be then emitted in the current instant, and also in the next one. Below, we formally introduce RMLq and provide with more explanation on the semantics.

3.4.2 Syntax and Semantics

Besides configurations, states, and expressions, the syntax of RMLq shall consider queues $q : \widetilde{h}$, where the first element is called a *queue identifier*, ranged over by $q, q', \ldots$ and $\widetilde{h}$ represents a (possibly empty) sequence of values. Sequences of values shall be ranged over by $\widetilde{h}, \widetilde{h'}, \ldots$ and configurations are ranged over by $K, K', \ldots$. The syntax of RMLq is given below.

Definition 3.94 (RMLq)**.** Let h denote constants, variables, signal names and pairs (h, h). We obtain the syntax of RMLq by extending the grammar in Def. 2.36 as shown below:

$$\begin{array}{rcl@{\qquad}rcl} e & ::= & \cdots \mid \texttt{pop}\ q \mid \texttt{put}\ q\ h \mid \texttt{isEmpty}\ q & \Sigma, \Sigma' & ::= & \emptyset \mid \Sigma, q : \widetilde{h} \\ \widetilde{h}, \widetilde{h'} & ::= & \epsilon \mid \widetilde{h} \cdot u & K, K' & ::= & \langle e \diamond \Sigma\rangle \end{array}$$

Above, h is used to denote *first-order values*, i.e., values that are not λ-expressions or process declarations. Whenever it is clear from the context, we will only call them values. Intuitively, in the syntax of RMLq the grammatical category of expressions is extended with constructs for dealing with queues. Given a queue identifier q, expression pop q pops the first value from q and returns it. Analogously, put q h puts the first-order value h at the end of q. Expression isEmpty q returns tt or ff, depending on whether q is empty or not. As hinted before, a state is a set of queues, where $\Sigma, q : \widetilde{h}$ denotes the union of Σ with the singleton $\{q : \widetilde{h}\}$. Notice that the union is only well-defined whenever $q \notin dom(\Sigma)$. Sequences associated with queue identifiers, denoted $\widetilde{h}$, are defined as concatenations of first-order values. Finally, configurations are pairs of expressions and states, written $\langle e \diamond \Sigma\rangle$.

The semantics of RMLq is obtained by extending the big-step semantics of ReactiveML, presented in § 2.4.1, with configurations and states. We denote big-step reductions in RMLq by using a similar notation to the one used for ReactiveML:

$$\langle e \diamond \Sigma\rangle \xrightarrow[S]{E,b} \langle e' \diamond \Sigma'\rangle$$

Above, the dashed arrow decrees that configuration $\langle e \diamond \Sigma\rangle$ evaluates to configuration $\langle e' \diamond \Sigma'\rangle$ in a single big-step reduction, where S is a signal environment, b is a termination boolean, and E is an event set. The notions of signal environments, termination booleans, and event sets are as in the semantics of ReactiveML. The big-step semantics is then generated by the rules in Fig. 3.9, Fig. 3.10, and Fig. 3.11.

We focus on presenting the most salient differences between the semantics of RMLq and ReactiveML: (1) extending the semantics to consider configurations and states, and (2) the rules for queue expressions in Fig. 3.11.

Configurations and states pose a challenge in RMLq. In particular, because expressions e_1 and e_2 executing in parallel could potentially try to modify the same queue at the same time (expressions dealt by Rules $\lfloor$L-Par$\rfloor$, $\lfloor$L-Done$\rfloor$, and $\lfloor$Pair$\rfloor$). Ideally, one would like to avoid these *interferences* altogether, but such assumptions are too restrictive. To address this issue we assume that expressions can "lock" the queue to execute a certain action. Thus, we have *reading locks* and *writing locks*. The most problematic cause of interference is whenever two expressions executed in parallel have the same lock on the same queue. Moreover, as we show at a later stage, our translation never induces situations in which two expressions are taking the same lock on the same queue.

To rule out the undesired interference, we define two functions $\mathtt{w}(e)$ and $\mathtt{r}(e)$ that return the set of queue identifiers of the queues from which e has the writing lock and reading lock, respectively. Then, with this information, it is possible to deduce that the kind of expressions we would like to rule out are of the form $\mathtt{let}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e_3$, where the condition $\mathtt{w}(e_1) \cap \mathtt{w}(e_2) = \mathtt{r}(e_1) \cap \mathtt{r}(e_2) = \emptyset$ does not hold.

Definition 3.95. Let e be an RMLq expression:

1. Function $\mathtt{w}(e)$ is defined below:
   - If $e = \mathtt{put}\ q\ v$ then $\mathtt{w}(e) = \{q\}$.
   - If $e = ()$, $e = x$, $e = c$, $e = n$, $e = \mathtt{pause}$, $e = \mathtt{pop}\ q$, or $e = \mathtt{isEmpty}\ q$ then $\mathtt{w}(e) = \emptyset$.
   - Inductively defined for every other case.
2. Function $\mathtt{r}(e)$ is defined below:
   - If $e = \mathtt{pop}\ q$, then $\mathtt{r}(e) = \{q\}$.
   - If $e = ()$, $e = x$, $e = c$, $e = n$, $e = \mathtt{pause}$, $e = \mathtt{put}\ q\ v$, or $e = \mathtt{isEmpty}\ q$ then $\mathtt{r}(e) = \emptyset$.
   - Inductively defined for every other case.

We are now left to handle the kind of interferences that are not captured by the assumption $\mathtt{w}(e_1) \cap \mathtt{w}(e_2) = \mathtt{r}(e_1) \cap \mathtt{r}(e_2) = \emptyset$. They come in two types:

(1) *The sets $\mathtt{w}(e_1)$, $\mathtt{w}(e_2)$, $\mathtt{r}(e_1)$, and $\mathtt{r}(e_2)$ are pairwise disjoint:* This case means that there are either no interferences or no queues are being modified in each expression.

(2) *Either $\mathtt{w}(e_i) \cap \mathtt{r}(e_j) \neq \emptyset$, $\mathtt{w}(e_i) \cap \mathtt{r}(e_j) \neq \emptyset$ holds:* This case means that there are two expressions e_1 and e_2 executed in parallel and that one of them has a writing lock and the other a reading lock.

$$\lfloor\textsc{Pause}\rfloor \quad \frac{}{\langle \mathtt{pause} \diamond \Sigma\rangle \xrightarrow[S]{\emptyset,\mathtt{ff}} \langle () \diamond \Sigma\rangle} \qquad \lfloor\textsc{Val}\rfloor \quad \frac{}{\langle v \diamond \Sigma\rangle \xrightarrow[S]{\emptyset,\mathtt{tt}} \langle v \diamond \Sigma\rangle}$$

$$\lfloor\textsc{Recur}\rfloor \quad \frac{\langle v\{{}^{x}/_{\mathtt{rec}\ x = v}\} \diamond \Sigma\rangle \xrightarrow[S]{E,\mathtt{tt}} \langle v' \diamond \Sigma_1\rangle}{\langle \mathtt{rec}\ x = v \diamond \Sigma\rangle \xrightarrow[S]{E,\mathtt{tt}} \langle v' \diamond \Sigma_1\rangle}$$

$$\lfloor\textsc{Run}\rfloor \quad \frac{\langle e \diamond \Sigma\rangle \xrightarrow[S]{E_1,\mathtt{tt}} \langle \mathtt{process}\ e' \diamond \Sigma_1\rangle \quad \langle e' \diamond \Sigma_1\rangle \xrightarrow[S]{E_2,b} \langle e'' \diamond \Sigma_2\rangle}{\langle \mathtt{run}\ e \diamond \Sigma\rangle \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} \langle e'' \diamond \Sigma_2\rangle}$$

$$\lfloor\textsc{Appl}\rfloor \quad \frac{\begin{array}{c}\langle e_1 \diamond \Sigma\rangle \xrightarrow[S]{E_1,\mathtt{tt}} \langle \lambda x.e_3 \diamond \Sigma_1\rangle \\ \langle e_2 \diamond \Sigma_1\rangle \xrightarrow[S]{E_2,\mathtt{tt}} \langle v' \diamond \Sigma_2\rangle \quad \langle e_3\{{}^{x}/_{v'}\} \diamond \Sigma_2\rangle \xrightarrow[S]{E_3,\mathtt{tt}} \langle v \diamond \Sigma_3\rangle\end{array}}{\langle e_1\ e_2 \diamond \Sigma\rangle \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3,\mathtt{tt}} \langle v \diamond \Sigma_3\rangle}$$

$$\lfloor\textsc{L-Par}\rfloor \quad \frac{\begin{array}{c}\mathsf{w}(e_1) \cap \mathsf{w}(e_2) = \mathsf{r}(e_1) \cap \mathsf{r}(e_2) = \emptyset \quad \Sigma' = \Sigma_1 \uplus_{\Sigma} \Sigma_2 \\ \langle e_1 \diamond \Sigma\rangle \xrightarrow[S]{E_1,b_1} \langle e_1' \diamond \Sigma_1\rangle \quad \langle e_2 \diamond \Sigma\rangle \xrightarrow[S]{E_2,b_2} \langle e_2' \diamond \Sigma_2\rangle \quad b_1 \wedge b_2 = \mathtt{ff}\end{array}}{\langle \mathtt{let}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e_3 \diamond \Sigma\rangle \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}} \langle \mathtt{let}\ x_1 = e_1'\ \mathtt{and}\ x_2 = e_2'\ \mathtt{in}\ e_3 \diamond \Sigma'\rangle}$$

$$\lfloor\textsc{L-Done}\rfloor \quad \frac{\begin{array}{c}\mathsf{w}(e_1) \cap \mathsf{w}(e_2) = \mathsf{r}(e_1) \cap \mathsf{r}(e_2) = \emptyset \quad \langle e_1 \diamond \Sigma\rangle \xrightarrow[S]{E_1,\mathtt{tt}} \langle v_1 \diamond \Sigma_1\rangle \\ \langle e_2 \diamond \Sigma\rangle \xrightarrow[S]{E_2,\mathtt{tt}} \langle v_2 \diamond \Sigma_2\rangle \quad \langle e_3\{{}^{x_1,x_2}/_{v_1,v_2}\} \diamond \Sigma_1 \uplus_{\Sigma} \Sigma_2\rangle \xrightarrow[S]{E_3,b} \langle e_3' \diamond \Sigma_3\rangle\end{array}}{\langle \mathtt{let}\ x_1 = e_1\ \mathtt{and}\ x_2 = e_2\ \mathtt{in}\ e_3 \diamond \Sigma\rangle \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3,b} \langle e_3' \diamond \Sigma_3\rangle}$$

$$\lfloor\textsc{Sig-P}\rfloor \quad \frac{\langle e_1 \diamond \Sigma\rangle \xrightarrow[S]{E_1,\mathtt{tt}} \langle n \diamond \Sigma_1\rangle \quad n \in S \quad \langle e_2 \diamond \Sigma_1\rangle \xrightarrow[S]{E_2,b} \langle e_2' \diamond \Sigma_2\rangle}{\langle \mathtt{present}\ e_1 ?\, e_2 : e_3 \diamond \Sigma\rangle \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2,b} \langle e_2' \diamond \Sigma_2\rangle}$$

$$\lfloor\textsc{Sig-NP}\rfloor \quad \frac{\langle e_1 \diamond \Sigma\rangle \xrightarrow[S]{E,\mathtt{tt}} \langle n \diamond \Sigma_1\rangle \quad n \notin S}{\langle \mathtt{present}\ e_1 ?\, e_2 : e_3 \diamond \Sigma\rangle \xrightarrow[S]{E,\mathtt{ff}} \langle e_3 \diamond \Sigma_1\rangle}$$

Figure 3.9: Big-step semantics for RMLq expressions (Part 1).

Dealing with the previous cases requires a way for modified states to be *merged*. In Fig. 3.9, the merging can be seen in Rules $\lfloor\textsc{L-Par}\rfloor$, $\lfloor\textsc{L-Done}\rfloor$, and $\lfloor\textsc{Pair}\rfloor$, written as

$$\lfloor\textsc{Emit}\rfloor\ \frac{\langle e_1 \diamond \Sigma\rangle \overset{E_1,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle n \diamond \Sigma_1\rangle \quad \langle e_2 \diamond \Sigma_1\rangle \overset{E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v \diamond \Sigma_2\rangle}{\langle \mathtt{emit}\ e_1\ e_2 \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} [\{v\}/n],\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle () \diamond \Sigma_2\rangle}$$

$$\lfloor\textsc{Pair}\rfloor\ \frac{\langle e_1 \diamond \Sigma\rangle \overset{E_1,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v_1 \diamond \Sigma_1\rangle \quad \langle e_2 \diamond \Sigma\rangle \overset{E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v_2 \diamond \Sigma_2\rangle}{\langle (e_1, e_2) \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle (v_1, v_2) \diamond \Sigma_1 \uplus_{\Sigma} \Sigma_2\rangle}$$

$$\lfloor\text{DW-NS}\rfloor\ \frac{\langle e_2 \diamond \Sigma\rangle \overset{E,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle n \diamond \Sigma_1\rangle \quad n \notin S}{\langle \mathtt{do}\ e_1\ \mathtt{when}\ e_2 \diamond \Sigma\rangle \overset{E,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle \mathtt{do}\ e_1\ \mathtt{when}\ n \diamond \Sigma_1\rangle}$$

$$\lfloor\textsc{Lp-Stu}\rfloor\ \frac{\langle e \diamond \Sigma\rangle \overset{E,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle e' \diamond \Sigma_1\rangle}{\langle \mathtt{loop}\ e \diamond \Sigma\rangle \overset{E,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle e'; \mathtt{loop}\ e \diamond \Sigma_1\rangle}$$

$$\lfloor\text{DW-}\textsc{Int}\rfloor\ \frac{\langle e_2 \diamond \Sigma\rangle \overset{E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle n \diamond \Sigma_1\rangle \quad n \in S \quad \langle e_1 \diamond \Sigma_1\rangle \overset{E_1,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle e_1' \diamond \Sigma_2\rangle}{\langle \mathtt{do}\ e_1\ \mathtt{when}\ e_2 \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle \mathtt{do}\ e_1'\ \mathtt{when}\ n \diamond \Sigma_2\rangle}$$

$$\lfloor\text{DW-}\textsc{End}\rfloor\ \frac{\langle e_2 \diamond \Sigma\rangle \overset{E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle n \diamond \Sigma_1\rangle \quad n \in S \quad \langle e_1 \diamond \Sigma_1\rangle \overset{E_1,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v \diamond \Sigma_2\rangle}{\langle \mathtt{do}\ e_1\ \mathtt{when}\ e_2 \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v \diamond \Sigma_1\rangle}$$

$$\lfloor\textsc{Lp-Un}\rfloor\ \frac{\langle e \diamond \Sigma\rangle \overset{E_1,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v \diamond \Sigma_1\rangle \quad \langle \mathtt{loop}\ e \diamond \Sigma_1\rangle \overset{E_2,b}{\underset{S}{\dashrightarrow}} \langle e' \diamond \Sigma_2\rangle}{\langle \mathtt{loop}\ e \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2,b}{\underset{S}{\dashrightarrow}} \langle e' \diamond \Sigma_2\rangle}$$

$$\lfloor\text{DU-}\textsc{End}\rfloor\ \frac{\langle e_2 \diamond \Sigma\rangle \overset{E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle n \diamond \Sigma_1\rangle \quad \langle e_1 \diamond \Sigma_1\rangle \overset{E_1,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v \diamond \Sigma_2\rangle}{\langle \mathtt{do}\ e_1\ \mathtt{until}\ e_2(x) \to e_3 \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle v \diamond \Sigma_2\rangle}$$

$$\lfloor\text{DU-P}\rfloor\ \frac{\langle e_2 \diamond \Sigma\rangle \overset{E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle n \diamond \Sigma_1\rangle \quad n \in S \quad \langle e_1 \diamond \Sigma_1\rangle \overset{E_1,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle e_1' \diamond \Sigma_2\rangle}{\langle \mathtt{do}\ e_1\ \mathtt{until}\ e_2(x) \to e_3 \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle e_3\{{}^{S^v(n)}/x\} \diamond \Sigma_2\rangle}$$

$$\lfloor\text{DU-NP}\rfloor\ \frac{\langle e_2 \diamond \Sigma\rangle \overset{E_2,\mathtt{tt}}{\underset{S}{\dashrightarrow}} \langle n \diamond \Sigma_1\rangle \quad n \notin S \quad \langle e_1 \diamond \Sigma_1\rangle \overset{E_1,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle e_1' \diamond \Sigma_2\rangle}{\langle \mathtt{do}\ e_1\ \mathtt{until}\ e_2(x) \to e_3 \diamond \Sigma\rangle \overset{E_1 \sqcup_{\mathsf{E}} E_2,\mathtt{ff}}{\underset{S}{\dashrightarrow}} \langle \mathtt{do}\ e_1'\ \mathtt{until}\ e_2(x) \to e_3 \diamond \Sigma_2\rangle}$$

Figure 3.10: Big-step semantics for RMLq expressions (Part 2).

$\Sigma_1 \uplus_{\Sigma} \Sigma_2$. Intuitively, merging works on two states Σ_1 and Σ_2 that have been obtained by modifying an initial state Σ. Then, the operation will fuse the modified queues of Σ_1 and Σ_2 into a new state.

$$\lfloor\textsc{Sig-Dec}\rfloor\ \dfrac{\begin{array}{c}\langle e_1 \diamond \Sigma\rangle \xrightarrow[S]{E_1,\mathtt{tt}} \langle v_1 \diamond \Sigma_1\rangle \quad \langle e_2 \diamond \Sigma_1\rangle \xrightarrow[S]{E_2,\mathtt{tt}} \langle v_2 \diamond \Sigma_2\rangle \\ \langle e_3\{^x/_n\} \diamond \Sigma_2\rangle \xrightarrow[S]{E_3,b} \langle e_3' \diamond \Sigma_3\rangle \quad n \text{ fresh} \quad S(n) = (v_1, v_2, m)\end{array}}{\langle \mathtt{signal}_{e_2}\ x : e_1\ \mathtt{in}\ e_3 \diamond \Sigma\rangle \xrightarrow[S]{E_1 \sqcup_{\mathsf{E}} E_2 \sqcup_{\mathsf{E}} E_3, b} \langle e_3' \diamond \Sigma_3\rangle}$$

$$\lfloor\textsc{Case}\rfloor\ \dfrac{\langle e \diamond \Sigma\rangle \xrightarrow[S]{\emptyset,\mathtt{tt}} \langle c_j \diamond \Sigma_1\rangle \quad j \in I \quad \langle e_j \diamond \Sigma_1\rangle \xrightarrow[S]{E,b} \langle e_j' \diamond \Sigma_2\rangle}{\langle \mathtt{match}\ e\ \mathtt{with}\ \{c_i \to e_i\}_{i\in I} \diamond \Sigma\rangle \xrightarrow[S]{E,b} \langle e_j' \diamond \Sigma_2\rangle}$$

$\lfloor\textsc{Put-Q}\rfloor$

$$\langle \mathtt{put}\ q\ v \diamond \Sigma, q : \widetilde{h}\rangle \xrightarrow[S]{\emptyset,\mathtt{tt}} \langle () \diamond \Sigma, q : \widetilde{h}\cdot v\rangle$$

$\lfloor\textsc{Pop-Q}\rfloor$

$$\langle \mathtt{pop}\ q \diamond \Sigma, q : v\cdot\widetilde{h}\rangle \xrightarrow[S]{\emptyset,\mathtt{tt}} \langle v \diamond \Sigma, q : \widetilde{h}\rangle$$

$\lfloor\textsc{NEmpty}\rfloor$

$$\langle \mathtt{isEmpty}\ q \diamond \Sigma, q : \widetilde{h}\rangle \xrightarrow[S]{\emptyset,\mathtt{tt}} \langle () \diamond \Sigma, q : \widetilde{h}\rangle$$

$\lfloor\textsc{Pop-Q}_\epsilon\rfloor$

$$\langle \mathtt{pop}\ q \diamond \Sigma, q : \epsilon\rangle \xrightarrow[S]{\emptyset,\mathtt{ff}} \langle \mathtt{pop}\ q \diamond \Sigma, q : \epsilon\rangle$$

$$\lfloor\textsc{Empty}\rfloor\ \langle \mathtt{isEmpty}\ q \diamond \Sigma, q : \epsilon\rangle \xrightarrow[S]{\emptyset,\mathtt{ff}} \langle \mathtt{isEmpty}\ q \diamond \Sigma, q : \epsilon\rangle$$

Figure 3.11: Big-step semantics for RMLq (Part 3): Queue-related operations.

The merging operation is formalized below. We first introduce some auxiliary definitions:

Definition 3.96 (Subtracting and Comparing Sequences). Let $\widetilde{h_1} = v_1 \cdot \ldots \cdot v_n$ and $\widetilde{h_2} = v_1 \cdot \ldots \cdot v_m$, with $0 \le m \le n$ be sequences of first-order values.

1. The subsequence relation $\widetilde{h_2} \subseteq \widetilde{h_1}$ holds if and only if $|\widetilde{h_1}| \ge |\widetilde{h_2}|$ and $\widetilde{h_1} = v_1 \cdot \ldots \cdot v_m \cdot \ldots \cdot v_n$ (i.e., $\widetilde{h_2}$ is a prefix of $\widetilde{h_1}$).

2. The subtraction operation $\widetilde{h_1} \setminus \widetilde{h_2}$ is defined as follows:

$$\widetilde{h_1} \setminus \widetilde{h_2} \overset{\mathsf{def}}{=} \begin{cases} v_{n-m} \cdot \ldots \cdot v_n & \text{if } n > m \\ \epsilon & \text{otherwise} \end{cases}$$

Definition 3.97 (State Merging). Let Σ, Σ_1 and Σ_2 be states such that $dom(\Sigma) = dom(\Sigma_1) = dom(\Sigma_2)$, and q be a queue such that $\Sigma(q) = \widetilde{h}$, $\Sigma_1(q) = \widetilde{h_1}$, and $\Sigma_2(q) = \widetilde{h_2}$. Then, assuming $i, j \in \{1, 2\}$ with $i \neq j$, and that $|\widetilde{h_i}| \le |\widetilde{h_j}|$, the state merging operation is defined as:

$$\Sigma_1 \uplus_\Sigma \Sigma_2(q) \overset{\mathsf{def}}{=} \widetilde{h_i} \cdot (\widetilde{h_j} \setminus \widetilde{h})$$

Notice that although Rules $\lfloor$L-Par$\rfloor$, $\lfloor$L-Done$\rfloor$, and $\lfloor$Pair$\rfloor$ are the most prominent sources of parallelism in the big-step semantics of RMLq, they are not the only one. Take, for example, Rule $\lfloor$DU-P$\rfloor$: in principle, both e_1 and e_2 are evaluated simultaneously. To control this kind of parallelism, we give the evaluation of these expressions a specific order. In particular, for expressions dealing with signals, we have forced a specific evaluation order, aiming to keep the states consistent throughout the derivation. Similarly, we have decided to give an evaluation order to applications: in Rule $\lfloor$Appl$\rfloor$, we specify the order in which the expressions are executed. Once again, the evaluation order arises from the fact that first it is necessary to recover the λ-expression before applying e_2 to e_1.

We now briefly address the second main difference with respect to the ReactiveML semantics: the queue-related rules in Fig. 3.11. Rule $\lfloor$Put-Q$\rfloor$ pushes an element into a queue at the end of the time instant and terminates instantaneously. Rule $\lfloor$Pop-Q$\rfloor$ takes the first element from the queue (if not empty) and terminates instantaneously. Rule $\lfloor$NEmpty$\rfloor$ enables `isEmpty` to terminate instantaneously if the queue is not empty. Rule $\lfloor$Pop-Q$_\epsilon\rfloor$ keeps the thread execution stuck for at least one instant if the queue is empty; Rule $\lfloor$Empty$\rfloor$ is similar.

We conclude this section by illustrating `RMLq` configurations and their big-step reductions. We also use these examples to clarify the role of the state merging operation.

Example 3.98. Let K_1, K_2, K_3, and K_4 be defined as follows:

$$\begin{aligned}
K_1 = {} & \langle \mathtt{put}\ q_1\ \mathtt{tt}; \mathtt{put}\ q_1\ \mathtt{tt}; \mathtt{put}\ q_1\ \mathtt{ff} \parallel \\
& \quad \mathtt{signal}\ s_1\ \mathtt{in}\ \mathtt{emit}\ s_1\ \mathtt{tt}; \mathtt{pause}\,; \mathtt{put}\ q_2\ s_1 \diamond q_1 : \epsilon, q_2 : \epsilon \rangle \\
K_2 = {} & \langle \mathtt{signal}\ s\ \mathtt{in}\ \mathtt{put}\ q_1\ \mathtt{tt}; \mathtt{pause}\,; \mathtt{emit}\ s\ (\mathtt{pop}\ q_1) \parallel \\
& \quad \mathtt{await}\ s(x)\ \mathtt{in}\ \mathtt{put}\ q_2\ x \diamond q_1 : \epsilon, q_2 : \epsilon \rangle \\
K_3 = {} & \langle \mathtt{put}\ q\ \mathtt{tt}; \mathtt{put}\ q\ \mathtt{tt}; () \parallel \mathtt{let}\ y_1 = \mathtt{pop}\ q\ \mathtt{in} \\
& \qquad\qquad\qquad\qquad \mathtt{let}\ y_2 = \mathtt{pop}\ q\ \mathtt{in}\ \mathtt{let}\ y_3 = \mathtt{pop}\ q\ \mathtt{in}\ () \diamond q : \mathtt{tt} \cdot \mathtt{ff} \rangle \\
K_4 = {} & \langle \mathtt{put}\ q\ \mathtt{tt} \parallel \mathtt{put}\ q\ \mathtt{tt} \parallel \mathtt{let}\ y_1 = \mathtt{pop}\ q\ \mathtt{in}\ \mathtt{let}\ y_2 = \mathtt{pop}\ q\ \mathtt{in}\ () \diamond q : \mathtt{tt} \rangle
\end{aligned}$$

In K_1, we can observe that adding elements to a queue is an instantaneous operation. The execution of the program is as follows:

$$\begin{aligned}
K_1 & \vdash\!\!\dashrightarrow \langle () \parallel \mathtt{put}\ q_2\ s_1 \diamond q_1 : \mathtt{tt} \cdot \mathtt{tt} \cdot \mathtt{ff}, q_2 : \epsilon \rangle \\
& \vdash\!\!\dashrightarrow \langle () \diamond q_1 : \mathtt{tt} \cdot \mathtt{tt} \cdot \mathtt{ff}, q_2 : s_1 \rangle
\end{aligned}$$

We illustrate the merging of states by showing the derivation tree for the first big-step reduction. Let $\Sigma = q_1 : \epsilon, q_2 : \epsilon$, $\Sigma_1 = q_1 : \mathtt{tt} \cdot \mathtt{tt} \cdot \mathtt{ff}, q_2 : \epsilon$, $\Sigma_2 = q_1 : \epsilon, q_2 : \epsilon$, $K_1' = \langle \mathtt{put}\ q_1\ \mathtt{tt}; \mathtt{put}\ \mathtt{tt}\,; \mathtt{put}\ \mathtt{ff}\ \diamond \Sigma \rangle$, and $K_1'' = \langle \mathtt{signal}\, s_1\ \mathtt{in}\ \mathtt{emit}\, s_1\, \mathtt{tt}; \mathtt{pause}\,; \mathtt{put}\ q_2\ s_1 \diamond \Sigma \rangle$:

$$\lfloor \text{L-Par} \rfloor\;
\dfrac{
\dfrac{\lfloor \text{Put-Q} \rfloor \times 3, \lfloor \text{Val} \rfloor}{\langle K_1' \diamond q_1 : \epsilon, q_2 : \epsilon \rangle \vdash\!\!\dashrightarrow \langle () \diamond \Sigma_1 \rangle}
\qquad
\dfrac{\lfloor \text{Sig-Dec} \rfloor, \lfloor \text{Pause} \rfloor}{\langle K_1'' \diamond q_1 : \epsilon, q_2 : \epsilon \rangle \vdash\!\!\dashrightarrow \langle \mathtt{put}\ q_2\ s_1 \diamond \Sigma_2 \rangle}
}{
\langle K_1 \diamond q_1 : \epsilon, q_2 : \epsilon \rangle \vdash\!\!\dashrightarrow \langle () \parallel \mathtt{put}\ q_2\ s_1 \diamond \Sigma_1 \uplus_\Sigma \Sigma_2 \rangle
}$$

The merging then can be calculated by solving the following operation:

$$\Sigma_1 \uplus_\Sigma \Sigma_2 = q_1 : \mathtt{tt} \cdot \mathtt{tt} \cdot \mathtt{ff}, q_2 : \epsilon \uplus_\Sigma q_1 : \epsilon, q_2 : \epsilon$$

Then, we merge individual queue. For q_1, by observing the queues and following Def. 3.97, it is clear that $\widetilde{h}_i = \epsilon$, $\widetilde{h}_j = \mathtt{tt} \cdot \mathtt{tt} \cdot \mathtt{ff}$ and $\widetilde{h} = \epsilon$. Similarly, for q_2, we have that all the queues are empty, thus:

$$(\Sigma_1 \uplus_\Sigma \Sigma_2)(q_1) = \epsilon \cdot (\texttt{tt} \cdot \texttt{tt} \cdot \texttt{ff} \setminus \epsilon) = \texttt{tt} \cdot \texttt{tt} \cdot \texttt{ff}$$
$$(\Sigma_1 \uplus_\Sigma \Sigma_2)(q_2) = \epsilon \cdot (\epsilon \setminus \epsilon) = \epsilon$$

Therefore, $\Sigma_1 \uplus_\Sigma \Sigma_2 = q_1 : \texttt{tt} \cdot \texttt{tt} \cdot \texttt{ff}, q_2 : \epsilon$, which is what one would expect.

Configuration K_2 shows how values can be moved from queue to another. This is an important point in our translation as aπ queues are associated to endpoints and communication between queues is done by moving messages from one queue to another. The execution of K_2 is given below:

$$K_2 \dashrightarrow \langle \texttt{emit}\ s\ (\texttt{pop}\ q_1) \parallel \texttt{await}\ s(x)\ \texttt{in}\ \texttt{put}\ q_2\ x \diamond q_1 : \texttt{tt}, q_2 : \epsilon \rangle$$
$$\dashrightarrow \langle () \parallel \texttt{put}\ q_2\ \texttt{tt} \diamond q_1 : \epsilon, q_2 : \epsilon \rangle \dashrightarrow \langle () \parallel ()) \diamond q_1 : \epsilon, q_2 : \texttt{tt} \rangle$$

Finally, configuration K_3 showcases an important feature of our semantics; let us analyze its big-step reductions:

$$K_3 \dashrightarrow \langle () \parallel \texttt{let}\ y_3 = \texttt{pop}\ q\ \texttt{in}\ () \diamond q : \texttt{tt} \cdot \texttt{tt} \rangle$$
$$\dashrightarrow \langle () \parallel () \diamond q : \texttt{tt} \rangle$$

Notice that the time instant ends when reaching configuration:

$$\langle () \parallel \texttt{let}\ y_3 = \texttt{pop}\ q\ \texttt{in}\ () \diamond q : \texttt{tt} \cdot \texttt{tt} \rangle$$

Therefore, values added to q by the left-hand expression $\texttt{put}\ q\ \texttt{tt}; \texttt{put}\ q\ \texttt{tt}; ()$ are only visible for the right-hand expression in the next time instant. This behavior was hinted before, and is the result of our assumption regarding the evaluation of put and pop operations. Indeed, from our semantics, it can be inferred that values are put in a queue at the end of the instants, whereas the pop operation is evaluated at the beginning.

Finally, configuration K_4 shows a process that does not have semantics according to the rules in Fig. 3.9, Fig. 3.10, and Fig. 3.11, as there is a sub-expression $\texttt{put}\ q\ \texttt{tt} \parallel \texttt{put}\ q\ \texttt{tt}$ that executes two put operations in parallel on the same queue. $\triangle$

3.4.3 *Equivalences for* RMLq

To compare RMLq configurations, we reuse Def. 2.40 as it is. Recalling this definition, we write $D_{\widetilde{x}}[\cdot]$ to denote the context containing all the signal declarations for $\widetilde{x}$. Furthermore, we extend this definition to configurations by writing $\langle D_{\widetilde{x}}[P] \diamond \Sigma \rangle$. We also define an analogous to Def. 2.42:

Definition 3.99 (Congruence Up-To Signal Declaration). We will say that two RMLq expressions e_1 and e_2 are congruent up-to signal declarations, written $e_1 \lesssim e_2$, whenever $e_1 = e'_1 \parallel \cdots \parallel e'_n$, $n \geq 1$ and there exist, possibly empty, signal declaration contexts $D_{\widetilde{x}}, D_{\widetilde{x_1}}, \ldots, D_{\widetilde{x_n}}$ such that:

$$D_{\widetilde{z}}[D_{\widetilde{x_1}}[e'_1] \parallel \cdots \parallel D_{\widetilde{x_n}}[e'_n]] \equiv_{\mathsf{R}} e_2$$

Furthermore, we say that two configurations $K_1 = \langle e_1 \diamond \Sigma_1 \rangle$ and $K_2 = \langle e_2 \diamond \Sigma_2 \rangle$ are congruent up-to signal declaration, written $K_1 \lesssim K_2$, if $e_1 \lesssim e_2$ and $\Sigma_1 = \Sigma_2$.

$$\begin{tikzcd}
\textbf{Operational} & \textbf{Declarative/Hybrid} \\
\mathsf{a}\pi \arrow[r, "(\!|\cdot|\!)"] & \mathsf{RMLq} \\
\pi_{\mathsf{R}}^{\lightning} \arrow[u] \arrow[r, "\llbracket\cdot\rrbracket_f^g"] & \mathsf{RML} \arrow[u] \\
\pi \arrow[u] \arrow[d] & \\
\pi_{\mathsf{OR}}^{\lightning} \arrow[d] \arrow[r, "\llbracket\cdot\rrbracket"] & \mathtt{lcc} \arrow[d] \\
\pi_{\mathsf{E}} \arrow[r, "\llbracket\cdot\rrbracket_f"] & \mathtt{lcc}^{\mathsf{p}}
\end{tikzcd}$$

Figure 3.12: Summary of expressiveness results: blue arrows symbolize extensions, red ones symbolize encodings.

3.5 Summary of Sources, Targets, and Translations

In this section we summarize all the formal languages used in our translations and informally introduce the notations we will use for the translations that will be presented in further chapters. A graphical summary of expressiveness results is given in Fig. 3.12.

We first instantiate Def. 2.1 to formally describe our source languages and target languages. We then summarize the translations and anticipate the notations used for each mapping.

Definition 3.100 (Source Languages). The following are the source languages used in our translations:

1. $\mathcal{L}_{\pi_{\mathsf{OR}}^{\lightning}} \overset{\text{def}}{=} \langle \pi_{\mathsf{OR}}^{\lightning}, \longrightarrow, \equiv_{\mathsf{S}} \rangle$, where: $\pi_{\mathsf{OR}}^{\lightning}$ is the set of well-typed programs induced by the type system in § 3.1.1, $\longrightarrow$ is the reduction semantics in Fig. 2.1, and $\equiv_{\mathsf{S}}$ corresponds to the structural congruence in Def. 2.11.

2. $\mathcal{L}_{\pi_{\mathsf{E}}} \overset{\text{def}}{=} \langle \pi_{\mathsf{E}}, \longrightarrow_{\mathsf{N}}, \equiv_{\mathsf{S}} \rangle$ where: π_{E} is the set of well-typed networks induced by the type system in § 3.1.3, $\longrightarrow_{\mathsf{N}}$ is the reduction semantics in Fig. 3.3, and $\equiv_{\mathsf{S}}$ is defined in § 3.1.3.

3. $\mathcal{L}_{\pi_{\mathsf{R}}^{\lightning}} \overset{\text{def}}{=} \langle \pi_{\mathsf{R}}^{\lightning}, \longrightarrow, \equiv_{\mathsf{S}} \rangle$, where: $\pi_{\mathsf{R}}^{\lightning}$ is the set of well-typed programs induced by the type system in § 3.1.2, $\longrightarrow$ is the reduction semantics in Fig. 2.1, and $\equiv_{\mathsf{S}}$ corresponds to the structural congruence in Def. 2.11.

4. $\mathcal{L}^{*}_{\pi_{\mathsf{R}}^{\lightning}} \overset{\text{def}}{=} \langle \pi_{\mathsf{R}}^{\lightning}, \hookrightarrow\!\!\!\!\rightarrow, \equiv_{\mathsf{S}} \rangle$, where: $\pi_{\mathsf{R}}^{\lightning}$ is the set of well-typed programs induced by the type system in § 3.1.2, $\hookrightarrow\!\!\!\!\rightarrow$ is the big-step semantics in Def. 3.31, and $\equiv_{\mathsf{S}}$ corresponds to the structural congruence in Def. 2.11.

5. $\mathcal{L}_{\mathsf{a}\pi} \overset{\mathsf{def}}{=} \langle \mathsf{a}\pi, \longrightarrow_{\mathsf{A}}, \equiv_{\mathsf{A}} \rangle$, where: $\mathsf{a}\pi$ corresponds to the set of well-typed programs induced by the type system in § 3.2.3, $\longrightarrow_{\mathsf{A}}$ is the reduction semantics in Fig. 3.5, and $\equiv_{\mathsf{A}}$ is the structural congruence in Def. 3.44.

6. $\mathcal{L}^*_{\mathsf{a}\pi} \overset{\mathsf{def}}{=} \langle \mathsf{a}\pi, \ggg\!\!\rightarrow, \equiv_{\mathsf{A}} \rangle$, where: $\mathsf{a}\pi$ corresponds to the set of well-typed programs induced by the type system in § 3.2.3, $\ggg\!\!\rightarrow$ is the big-step semantics in Def. 3.81, and $\equiv_{\mathsf{A}}$ is the structural congruence in Def. 3.44.

Remark 3.101 (*Starred Languages*). Notice that the starred languages $\mathcal{L}^*_{\pi^{\lightning}_{\mathsf{R}}}$ and $\mathcal{L}^*_{\mathsf{a}\pi}$ *are not* used as sources of new translations. Rather, these languages are obtained by defining a big-step semantics in terms of the reduction semantics of $\pi^{\lightning}_{\mathsf{R}}$ and $\mathsf{a}\pi$, respectively. The big-step semantics and reduction semantics are proven to be semantically corresponding. We shall use starred languages to prove that the translations from $\pi^{\lightning}_{\mathsf{R}}$ and $\mathsf{a}\pi$ into RML and RMLq, respectively are not only refined encodings (cf. Def. 2.6), but also valid (cf. Def. 2.3)

Definition 3.102 (Target Languages). The following are the source languages used in our translations:

1. $\mathcal{L}_{\mathtt{lcc}} \overset{\mathsf{def}}{=} \langle \mathtt{lcc}, \longrightarrow_{1}, \cong_{\mathcal{D}\mathcal{E}} \rangle$, where: $\mathtt{lcc}$ is the set of $\mathtt{lcc}$ processes, $\longrightarrow_{1}$ corresponds to the semantics induced by Fig. 2.5, and $\cong_{\mathcal{D}\mathcal{E}}$ corresponds to the barbed congruence in Def. 2.34—sets $\mathcal{D}$ and $\mathcal{E}$ are properly instantiated in Def. 4.3.

2. $\mathcal{L}_{\mathtt{lcc}^{\mathsf{p}}} \overset{\mathsf{def}}{=} \langle \mathtt{lcc}, \longrightarrow_{1}, \cong_{\mathcal{D}\mathcal{E}} \rangle$, where: $\mathtt{lcc}$ is the set of $\mathtt{lcc}$ processes, $\longrightarrow_{1}$ corresponds to the semantics introduced in § 3.3.2, and $\cong_{\mathcal{D}\mathcal{E}}$ corresponds to the barbed congruence in Def. 2.34—sets $\mathcal{D}$ and $\mathcal{E}$ are instantiated in Def. 5.4.

3. $\mathcal{L}_{\mathsf{RML}} \overset{\mathsf{def}}{=} \langle \mathsf{RML}, \longmapsto, \lesssim \rangle$, where: RML corresponds to ReactiveML expressions, $\longmapsto$ corresponds to the semantics defined by Fig. 2.8 and Fig. 2.9, and $\lesssim$ corresponds to the pre-order define in Def. 2.42.

4. $\mathcal{L}_{\mathsf{RMLq}} \overset{\mathsf{def}}{=} \langle \mathsf{RMLq}, \vdash\!\dashrightarrow, \lesssim \rangle$, where: RMLq corresponds to RMLq configurations, $\vdash\!\dashrightarrow$ corresponds to the semantics defined by Fig. 3.9, Fig. 3.10, and Fig. 3.11, and $\lesssim$ corresponds to the pre-order defined in Def. 3.99.

We now proceed to introduce the mappings between languages that will make part of our translations (cf. Def. 2.1). We often use the set of processes to refer to both source and target languages, rather than the notation above (e.g., RML refers to $\mathcal{L}_{\mathsf{RML}}$, $\pi^{\lightning}_{\mathsf{OR}}$ refers to $\mathcal{L}_{\pi^{\lightning}_{\mathsf{OR}}}$). The following mappings correspond to the translations presented in this dissertation:

1. The translation $[\![\cdot]\!] : \pi^{\lightning}_{\mathsf{OR}} \to \mathtt{lcc}$ is studied in Ch. 4.

2. The translation $[\![\cdot]\!]_f : \pi_{\mathsf{E}} \to \mathtt{lcc}^{\mathsf{p}}$, where f is a function that carries information about co-variables, is studied in Ch. 5.

3. The translation $[\![\cdot]\!]^g_f : \pi^{\lightning}_{\mathsf{R}} \to \mathsf{RML}$, where functions f and g carry some necessary renamings, is studied in Ch. 7.

4. The translation $(\!|\cdot|\!) : \mathsf{a}\pi \to \mathsf{RMLq}$ is studied in Ch. 8.

PART II

SESSION-BASED CONCURRENCY AND CONCURRENT CONSTRAINT PROGRAMMING

4
Encoding $\pi_{\mathsf{OR}}^{\lightning}$ in `lcc`

In this chapter we introduce a translation from $\pi_{\mathsf{OR}}^{\lightning}$ (cf. § 3.1.1) into `lcc` (cf. § 2.3). We formally describe it in § 4.1. Next, we prove the static correctness properties (cf. Def. 2.6) in § 4.2. Then, in § 4.3, we prove the operational correspondence property and state that $[\![\cdot]\!]$ is a valid encoding (cf. Def. 2.3). Finally, in § 4.4, we illustrate the use of this encoding using the timed patterns presented in § 1.6.

4.1 The Translation

In this section we formalize the translation mentioned above. In § 4.1.1 we introduce a specialized constraint system for dealing with sessions and instantiate the behavioral equivalences mentioned in Def. 3.102(1). Finally, in § 4.1.2 we present the mapping we shall use to transform π processes into `lcc` processes.

4.1.1 A Session Constraint System and Observational Equivalences

Before formally introducing the translation, we present the constraint system which defines the necessary predicates and deduction rules for modeling communication in $\pi_{\mathsf{OR}}^{\lightning}$.

Definition 4.1 (Session Constraint System). The *session constraint system* is the tuple $\langle \mathcal{C}, \Sigma, \vdash_{\mathcal{S}} \rangle$, where

- Σ is the set of predicates given in Fig. 4.1;
- $\mathcal{C}$ is the set of constraints obtained by using linear logic operators $!$, $\otimes$ and $\exists$ over the predicates of Σ;
- $\vdash_{\mathcal{S}}$ is given by the rules in Fig. 2.4, extended with the syntactic equality '$=$' .

$$\Sigma ::= \mathsf{rcv}(x,y) \;\big|\; \mathsf{snd}(x,y) \;\big|\; \mathsf{sel}(x,l) \;\big|\; \mathsf{bra}(x,l) \;\big|\; \{x{:}y\}$$

Figure 4.1: Session constraint system: Predicates (cf. Def. 4.1).

Some intuitions on the definition above follow. The first four predicates in Fig. 4.1 serve as acknowledgments of actions in the source $\pi_{\mathsf{OR}}^{\lightning}$ process: predicate $\mathsf{rcv}(x,y)$ signals an input action on x of a value denoted by y; conversely, predicate $\mathsf{snd}(x,y)$ signals an output action on x of a value denoted by y. Predicates $\mathsf{sel}(x,l)$ and $\mathsf{bra}(x,l)$ signal selection and branching actions on x involving label l, respectively. Finally, predicate $\{x{:}y\}$ connects variables x and y as required in the translation of the restriction operator in $\pi_{\mathsf{OR}}^{\lightning}$.

Having introduced the constraint system, we now define two sets of observables which will be important for the translation. Below, we define the *output* and *complete* observables of `lcc` processes that use the constraint system in Def. 4.1.

Definition 4.2 (Complete and Output Observables). Let $\mathcal{C}$ be the constraint system in Def. 4.1. We define $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}$, the set of *output observables* of `lcc` processes as follows:

$$\begin{aligned}\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}} &\stackrel{\text{def}}{=} \{\exists\widetilde{z}.\mathsf{snd}(x,v) \;\big|\; x\in\mathcal{V}_s \wedge x\in\widetilde{z}\wedge(v\in\widetilde{z}\vee v\notin\mathcal{V}_s)\}\\ &\quad\cup\{\exists\widetilde{z}.\mathsf{sel}(x,l)\;\big|\; x\in\mathcal{V}_s\wedge l\in\mathcal{B}_\pi\wedge x\in\widetilde{z}\}\end{aligned}$$

We also define $\mathcal{D}^\star_{\pi_{\mathsf{OR}}^{\lightning}}$, the set of *complete observables* of `lcc`, which extends $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}$ as follows:

$$\begin{aligned}\mathcal{D}^\star_{\pi_{\mathsf{OR}}^{\lightning}} &\stackrel{\text{def}}{=} \mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\cup\{\mathtt{tt}\}\cup\{\exists\widetilde{z}.\mathsf{rcv}(x,y)\;\big|\; x,y\in\mathcal{V}_s\wedge x\neq y\wedge x\in\widetilde{z}\}\\ &\quad\cup\{\exists\widetilde{z}.\mathsf{bra}(x,l)\;\big|\; x\in\mathcal{V}_s\wedge l\in\mathcal{B}_\pi\wedge x\in\widetilde{z}\}\end{aligned}$$

Notice that constraints such as $\{x{:}y\}$ are not part of the observables. Considering that in the translation co-variable predicates are persistent, the information of co-variables can be derived by using other constraints. In particular, as we will show later, if $\exists x,y.\mathsf{snd}(x,v)$ and $\exists x,y.\mathsf{rcv}(y,v)$ are in the set of complete observables of some process P then constraint $!\{x{:}y\}$ must be in the corresponding constraint store too. This will become clear when we analyze the shape of translated processes (cf. Def. 4.13).

Using the sets of observables defined above, we can then instantiate sets $\mathcal{D}$ and $\mathcal{E}$ for the barbed congruence for `lcc` mentioned in Def. 3.102(1) (cf. Def. 2.34). In particular, we set $\mathcal{D}=\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}$, and $\mathcal{E}=\mathcal{C}$ (cf. Def. 4.1). The barbed congruence is then defined below:

Definition 4.3 (Weak o-barbed bisimilarity and congruence). We define *weak o-barbed bisimilarity* and *weak o-barbed congruence* as follows:

1. Weak o-barbed bisimilarity, denoted $\approx_1^{\pi_{\mathsf{OR}}^{\lightning}}$, arises from Def. 2.33 as the weak $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\mathcal{C}$-barbed bisimilarity.

2. Weak o-barbed congruence, denoted $\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}$, arises from Def. 2.34 as the weak $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\mathcal{C}$-barbed congruence.

$$
\begin{aligned}
[\![x\langle v\rangle.P]\!] &\overset{\mathsf{def}}{=} \overline{\mathsf{snd}(x,v)} \parallel \forall z\big(\mathsf{rcv}(z,v) \otimes \{x{:}z\} \rightarrow [\![P]\!]\big) \qquad (z \notin \mathsf{fv}(P))\\
[\![x(y).P]\!] &\overset{\mathsf{def}}{=} \forall y,w\big(\mathsf{snd}(w,y) \otimes \{w{:}x\} \rightarrow \overline{\mathsf{rcv}(x,y)} \parallel [\![P]\!]\big) \qquad (w,z \notin \mathsf{fv}(P))\\
[\![x \triangleleft l.P]\!] &\overset{\mathsf{def}}{=} \overline{\mathsf{sel}(x,l)} \parallel \forall z\big(\mathsf{bra}(z,l) \otimes \{x{:}z\} \rightarrow [\![P]\!]\big) \qquad (z \notin \mathsf{fv}(P))\\
[\![x \triangleright \{l_i{:}P_i\}_{i\in I}]\!] &\overset{\mathsf{def}}{=} \forall l,w\big(\mathsf{sel}(w,l) \otimes \{w{:}x\} \rightarrow \overline{\mathsf{bra}(x,l)} \parallel \\
&\qquad\qquad \prod_{i\in I} \forall\epsilon(l = l_i \rightarrow [\![P_i]\!])\big) \qquad (w,z \notin \mathsf{fv}(P))\\
[\![v?\,(P){:}(Q)]\!] &\overset{\mathsf{def}}{=} \forall\epsilon(v = \mathtt{tt} \rightarrow [\![P]\!]) \parallel \forall\epsilon(v = \mathtt{ff} \rightarrow [\![Q]\!])\\
[\![(\boldsymbol{\nu}xy)P]\!] &\overset{\mathsf{def}}{=} \exists x,y.\,(!\overline{\{x{:}y\}} \parallel [\![P]\!])\\
[\![*\,x(y).P]\!] &\overset{\mathsf{def}}{=} !\,[\![x(y).P]\!]\\
[\![P \mid Q]\!] &\overset{\mathsf{def}}{=} [\![P]\!] \parallel [\![Q]\!]\\
[\![\mathbf{0}]\!] &\overset{\mathsf{def}}{=} \overline{\mathtt{tt}}
\end{aligned}
$$

Figure 4.2: Translation from $\pi_{\mathsf{OR}}^{\lightning}$ into lcc (cf. Def. 4.4).

4.1.2 *Mapping* π *Processes Into* `lcc` *Processes*

With the observational equivalences defined above, we are now ready to introduce our translation. We first give the formal definition and then some intuitions follow.

Definition 4.4 (Translation of $\pi_{\mathsf{OR}}^{\lightning}$ into lcc). Let $\mathcal{L}_{\pi_{\mathsf{OR}}^{\lightning}} = \langle \pi_{\mathsf{OR}}^{\lightning}, \longrightarrow, \equiv_{\mathsf{S}}\rangle$ and $\mathcal{L}_{\mathtt{lcc}} = \langle \mathtt{lcc}, \longrightarrow_1, \cong_1^{\pi_{\mathsf{OR}}^{\lightning}}\rangle$ (cf. Def. 3.100(1) and Def. 3.102(1), respectively). The translation from $\mathcal{L}_{\pi_{\mathsf{OR}}^{\lightning}}$ into $\mathcal{L}_{\mathtt{lcc}}$ is given as the pair $\langle [\![\cdot]\!], \varphi_{[\![\cdot]\!]}\rangle$, where:

(a) $[\![\cdot]\!]$ is the process mapping defined in Fig. 4.2.

(b) $\varphi_{[\![\cdot]\!]}(x) = x$, i.e., the identity function.

In Fig. 4.2, the mapping uses the predicates shown in Fig. 4.1 as acknowledgment messages to ensure correct synchronizations. Below, we discuss some of the translation cases:

- The output process $x\langle v\rangle.P$ is translated by using both tell and ask constructs:

$$\overline{\mathsf{snd}(x,v)} \parallel \forall z\big(\mathsf{rcv}(z,v) \otimes \{x{:}z\} \rightarrow [\![P]\!]\big)$$

  Intuitively, the translation of an output posts predicate $\mathsf{snd}(x,v)$ in the store, signaling that an output has taken place, and can be received by an input process. The sequential behavior of the $\pi_{\mathsf{OR}}^{\lightning}$ output is then modeled by only allowing the continuation to be activated after the translation has received predicate $\mathsf{rcv}(y,v)$, which signals that the message has been correctly received by its co-variable (e.g., predicate $\{x{:}y\}$). Therefore, input-output interactions are represented in $[\![\cdot]\!]$ as two-step synchronizations.

- The translation of an input process $x(y).Q$ is complementary to the translation of the output process:

$$\forall y, w\big(\mathsf{snd}(w, y) \otimes \{w{:}x\} \rightarrow \overline{\mathsf{rcv}(x, y)} \parallel [\![P]\!]\big)$$

  The translation above does not post any information at the beginning of its execution. In fact, whenever a predicate $\mathsf{snd}(x, v)$ is detected by the outermost abstraction of the translation, $\mathsf{snd}(x, v)$ is consumed to obtain both the subject x and the object y. Then, the co-variable restriction is checked: this restriction is used to enforce synchronization between intended endpoints. Subsequently, the translation emits a message $\mathsf{rcv}(\cdot, \cdot)$ and spawns its continuation.

- The translation of branching-select synchronizations follows a similar strategy, using $\mathsf{bra}(\cdot, \cdot)$ and $\mathsf{sel}(\cdot, \cdot)$ as acknowledgment messages. In this case, the exchanged value is one of the pairwise distinct labels, say l_j; depending on the received label, the translation of branching will spawn exactly one continuation, as expected. The continuations corresponding to labels different from l_j get blocked, as their equality guard can never be satisfied. Similarly, the translation of conditionals makes both branches available for execution; we use a parameterized ask as guard to ensure that only one of them will be executed.

- The translation of the restriction $(\boldsymbol{\nu}xy)P$ provides infinitely many copies of the co-variable constraint $\{x{:}y\}$, using hiding to appropriately regulate the scope of the involved endpoints. The translation of replicated processes simply corresponds to the replication of the translation of the given input-guarded process. Finally, the translations of parallel composition and inaction are self-explanatory.

We reaffirm the intuitions for our translation by presenting some interesting examples.

Example 4.5. We recall a non-replicated version of process P_3 in Ex. 2.25:

$$P_3' = (\boldsymbol{\nu}wz)(\boldsymbol{\nu}xy)(x\langle z\rangle.w(u').\mathbf{0} \mid y(u).u\langle \mathtt{tt}\rangle.\mathbf{0})$$

whose translation $[\![P_3]\!]$ is given below:

$$\begin{aligned}
&\exists x, y, w, z.\big(\,!\,\overline{\{x{:}y\}} \parallel !\,\overline{\{w{:}z\}} \parallel \overline{\mathsf{snd}(x, z)} \parallel \\
&\qquad \forall w_1\big(\mathsf{rcv}(w_1, z) \otimes \{w_1{:}x\} \rightarrow \\
&\qquad\qquad \forall w_2, u'\big(\mathsf{snd}(w_2, u') \otimes \{w_2{:}w\} \rightarrow \overline{\mathsf{rcv}(w, u')} \parallel \overline{\mathtt{tt}}\big)\big) \\
&\qquad \forall w_3, u\big(\mathsf{snd}(w_3, u) \otimes \{w_3{:}y\} \rightarrow \overline{\mathsf{rcv}(y, u)} \parallel \overline{\mathsf{snd}(u, \mathtt{tt})} \parallel \\
&\qquad\qquad \forall w_4\big(\mathsf{rcv}(w_4, \mathtt{tt}) \otimes \{w_4{:}u\} \rightarrow \overline{\mathtt{tt}}\big)\big)\big)
\end{aligned}$$

where, using the semantics in Fig. 2.5, it can be shown that:

$$\begin{aligned}
[\![P_3']\!] \longrightarrow_1^2 \exists x, y, w, z.\big(&\,!\,\overline{\{x{:}y\}} \parallel !\,\overline{\{w{:}z\}} \parallel \\
&\underbrace{\forall w_2, u'\big(\mathsf{snd}(w_2, u') \otimes \{w_2{:}w\} \rightarrow \overline{\mathsf{rcv}(w, u')} \parallel \overline{\mathtt{tt}}\big)}_{[\![w(u').\mathbf{0}]\!]} \parallel \\
&\underbrace{\overline{\mathsf{snd}(z, \mathtt{tt})} \parallel \forall w_4\big(\mathsf{rcv}(w_4, \mathtt{tt}) \otimes \{w_4{:}z\} \rightarrow \overline{\mathtt{tt}}\big)}_{[\![u\langle \mathtt{tt}\rangle.\mathbf{0}]\!]}\big)
\end{aligned}$$

which can then reduce as expected.

The most important takeaway from this example is that it shows how the translation captures *delegation*, an important property of the π-calculus that allows the reconfiguration of the communication structure (i.e., *mobility*). Above, we can see how channel endpoint z is being sent over x, to be received by endpoint y, which then enables the communication between w and z. △

In the next example, we show how certain forms of nondeterminism are captured by our translation. In particular, we show that our translation captures processes where nondeterministic behavior comes from a client trying to interact with two or more replicated processes.

Example 4.6. Let us consider the $\pi^{\sharp}_{\mathsf{OR}}$ program P_{17} below, which is not encodable in [LOP09]:

$$P_{17} = (\boldsymbol{\nu}xy)(x\langle v_1\rangle.Q_1 \mid *\, y(z_1).Q_2 \mid *\, y(z_2).Q_3) \tag{4.1}$$

where the following reduction is possible:

$$P_{17} \longrightarrow (\boldsymbol{\nu}xy)(Q_1 \mid Q_2\{v_1/z_1\} \mid *\, y(z_2).Q_3 \mid *\, y(z_1).Q_2) = P'_{17}$$

and whose translation follows:

$$\begin{aligned}\llbracket P_{17}\rrbracket = \exists x, y.\big(\ !\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x, v_1)} \parallel \forall z(\mathsf{rcv}(z, v_1) \otimes \{x{:}z\} \rightarrow \llbracket Q_1\rrbracket) \parallel \\ !\,\forall z, w(\mathsf{snd}(w, z_1) \otimes \{w{:}y\} \rightarrow \overline{\mathsf{rcv}(y, z_1)} \parallel \llbracket Q_2\rrbracket) \parallel \\ !\,\forall z, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\} \rightarrow \overline{\mathsf{rcv}(y, z_2)} \parallel \llbracket Q_3\rrbracket)\big)\end{aligned}$$

Fig. 4.3 shows how this reduction proceeds in lcc. Similarly, the other possible reduction (the one that interacts with $*\, y(z_2).Q_3$) is also captured by $\llbracket\cdot\rrbracket$. △

We conclude this section by introducing the notion of *target terms*. Intuitively, a target term refers to any process obtained by the execution of a translated process.

Definition 4.7 (Target Terms). We define *target terms* as the set of lcc processes that are induced by the translation of well-typed $\pi^{\sharp}_{\mathsf{OR}}$ programs and is closed under τ-transitions: $\{S \mid \llbracket P\rrbracket \overset{\tau}{\Longrightarrow}_1 S \text{ and } \vdash P\}$. We shall use $S, S', \ldots$ to range over target terms.

4.2 Static Correctness

In this section we prove that $\llbracket\cdot\rrbracket$ is both name invariant and compositional. These are two of the required properties to state that our encoding is valid (cf. Def. 2.3). First, we prove that the translation is name invariant with respect to the renaming policy in Def. 4.4.

Theorem 4.8 (Name Invariance for $\llbracket\cdot\rrbracket$). *Let P be a well-typed $\pi^{\sharp}_{\mathsf{OR}}$ process. Also, let σ and x be a substitution satisfying the renaming policy for $\llbracket\cdot\rrbracket$ (Def. 4.4(b)), and a variable in $\pi^{\sharp}_{\mathsf{OR}}$, respectively.*
Then $\llbracket P\sigma\rrbracket = \llbracket P\rrbracket\sigma'$, with $\varphi_{\llbracket\cdot\rrbracket}(\sigma(x)) = \sigma'(\varphi_{\llbracket\cdot\rrbracket}(x))$ and $\sigma = \sigma'$.

Proof. By structural induction on P. □

$$\begin{aligned}
[\![P_{17}]\!] &\equiv \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\{x{:}y\} \otimes \mathsf{snd}(x, v_1)} \parallel \\
&\qquad\qquad !\,\forall z, w(\mathsf{snd}(w, z_1) \otimes \{w{:}y\} \to \overline{\mathsf{rcv}(y, z_1)} \parallel [\![Q_2]\!]) \parallel \\
&\qquad\qquad \forall z(\mathsf{rcv}(z, v_1) \otimes \{x{:}z\} \to [\![Q_1]\!]) \parallel \\
&\qquad\qquad !\,\forall z, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\} \to \overline{\mathsf{rcv}(y, z_2)} \parallel [\![Q_3]\!])\big) \\
&\longrightarrow_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{rcv}(y, v_1)} \parallel [\![Q_2]\!]\{v_1, x/z_1, w\} \parallel [\![*\, y(z_1).Q_2]\!] \parallel \\
&\qquad\qquad \forall z(\mathsf{rcv}(z, v_1) \otimes \{x{:}z\} \to [\![Q_1]\!]) \parallel \\
&\qquad\qquad !\,\forall z, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\} \to \overline{\mathsf{rcv}(y, z_2)} \parallel [\![Q_3]\!])\big) \\
&\equiv \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\{x{:}y\} \otimes \mathsf{rcv}(y, v_1)} \parallel \forall z(\mathsf{rcv}(z, v_1) \otimes \{x{:}z\} \to [\![Q_1]\!]) \parallel \\
&\qquad\qquad [\![*\, y(z_1).Q_2]\!] \parallel \\
&\qquad\qquad [\![Q_2]\!]\{v_1, x/z_1, w\} \parallel \forall z, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\} \to \overline{\mathsf{rcv}(y, z_2)} \parallel [\![Q_3]\!])\big) \\
&\longrightarrow_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![Q_1]\!]\{z/y\} \parallel [\![*\, y(z_1).Q_2]\!] \parallel \\
&\qquad\qquad [\![Q_2]\!]\{v_1, x/z_1, w\} \parallel \\
&\qquad\qquad !\,\forall z, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\} \to \overline{\mathsf{rcv}(y, z_2)} \parallel [\![Q_3]\!])\big) = [\![P'_{17}]\!]
\end{aligned}$$

Figure 4.3: One possible evolution of the `lcc` translation of the $\pi_{\mathsf{OR}}^{\lightning}$ program (4.1) (cf. Ex. 4.6).

Next, we prove that $[\![\cdot]\!]$ is compositional with respect to the restriction and parallel composition operator. Below, we assume the expected extension of $[\![\cdot]\!]$ to evaluation contexts by decreeing $[\![-]\!] = -$.

Theorem 4.9 (Compositionality for $[\![\cdot]\!]$). *Let P and $E[-]$ be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ process and an $\pi_{\mathsf{OR}}^{\lightning}$ evaluation context as in Def. 2.12, respectively. Then we have: $[\![E[P]]\!] = [\![E]\!]\big[[\![P]\!]\big]$.*

Proof. By induction on the structure of P and a case analysis on $E[-]$, using Def. 2.12. □

4.3 Operational Correspondence

In this section we prove that our translation satisfy the operational correspondence property required by valid encodings (cf. Def. 2.3). In § 4.3.1 we present further discussions on the observational equivalences in Def. 4.3; also, we present an analysis on the syntactic shape of translated programs. Finally, we introduce so-called *junk* processes—processes which do not have any behavior in our translation.

In § 4.3.2 we present the proof of operational completeness for our translation. In § 4.3.3 we introduce all the necessary machinery for proving operational soundness: first, we give invariants for translated pre-redexes, redexes, and well-typed programs. Next, we present a diamond property for translated terms and present the proof of operational soundness.

4.3.1 Preliminaries

Observational Equivalences

Using the weak o-barbed equivalences in Def. 4.3 we can equate (i) lcc processes obtained from the translation of an $\pi_{\mathsf{OR}}^{\lightning}$ process with (ii) so-called *junk processes* (cf. Def. 4.14) and *intermediate redexes* (cf. Def. 4.25) that do not correspond to the translation of any source process, but that may be reached by executing a process in (i). More precisely, given an $\pi_{\mathsf{OR}}^{\lightning}$ process P that reduces to Q via some communication rule (i.e., $\lfloor \text{Com} \rfloor$, $\lfloor \text{Rep} \rfloor$ or $\lfloor \text{Sel} \rfloor$), we will show that if $[\![P]\!] \xRightarrow{\tau}_1 T$ then T can reduce to an S' that is $\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}$-equivalent to $[\![Q]\!]$. The following example considers $\pi_{\mathsf{OR}}^{\lightning}$ processes that make a selection using Rule $\lfloor \text{Sel} \rfloor$ and will intuitively show how junk and intermediate redexes behave under $\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}$:

Example 4.10. For this example, let us consider the following process, which models a simple transaction between a client an a store. We have decided to use a less abstract process in this example, as we believe this helps with better understanding the behavioral equivalences.

$$P_{18} = (\boldsymbol{\nu}xy)(x \triangleleft buy.\, x\langle 5406\rangle.\, x(inv).\mathbf{0} \,|\, y \triangleright \{buy{:}\, y(w).y\langle \mathsf{invoice}\rangle.\mathbf{0}, quit{:}\, y(w').\mathbf{0}\}) \quad (4.2)$$

In P_{18} above, we have a client (i.e., leftmost process) that wants to buy some item from a store (i.e., rightmost process). Intuitively, the client selects to buy, and sends its credit card number, before receiving an invoice. Dually, the store is waiting for a selection to be made. If the *buy* is picked, the store awaits for the credit card number, before emitting an invoice. The translation of P_{18} is then given below:

$$\begin{aligned}
[\![P_{18}]\!] = \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{sel}(x, buy)} \parallel \forall u_1(\mathsf{bra}(u_1, buy) \otimes \{x{:}u_1\} \to \overline{\mathsf{snd}(x, 5406)} \parallel \\
\forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(inv).\mathbf{0}]\!])) \parallel \\
\forall l, w(\mathsf{sel}(w, l) \otimes \{w{:}y\} \to \overline{\mathsf{bra}(y, l)} \parallel \\
\forall \epsilon(l = buy \to \forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \to \overline{\mathsf{rcv}(y, w)} \parallel \\
[\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!])) \parallel \\
\forall \epsilon(l = quit \to \forall w_2, u(\mathsf{snd}(w_2, u) \otimes \{w_2{:}y\} \to \overline{\mathsf{rcv}(y, u)} \parallel [\![y(w').\mathbf{0}]\!]))))
\end{aligned}$$

Combining Def. 2.30 and Def. 4.2, we have the following observables:

$$\begin{aligned}
\mathcal{O}^{\mathcal{D}^\star_{\pi_{\mathsf{OR}}^{\lightning}}}([\![P_{18}]\!]) =&\{(\exists x, y.\mathsf{sel}(x, buy)), (\exists x, y.\mathsf{bra}(y, buy)), (\exists x, y.\mathsf{snd}(x, 5406)), \\
&(\exists x, y.\mathsf{rcv}(y, 5406)), (\exists x, y.\mathsf{snd}(y, \mathsf{invoice})), (\exists x, y.\mathsf{rcv}(x, \mathsf{invoice})), \\
&(\exists x, y.\mathtt{tt})\} \\
\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}([\![P_{18}]\!]) =&\{(\exists x, y.\mathsf{sel}(x, buy)), (\exists x, y.\mathsf{snd}(x, 5406)), (\exists x, y.\mathsf{snd}(y, \mathsf{invoice}))\}
\end{aligned}$$

Below, we analyze the single τ-transition for the translation of P_{18}; we assume that $[\![P_{18}]\!] \longrightarrow_1 S_1$, where:

$$\begin{aligned}
S_1 = \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{bra}(y, buy)} \parallel \forall u_1(\mathsf{bra}(u_1, buy) \otimes \{x{:}u_1\} \to \overline{\mathsf{snd}(x, 5406)} \parallel \\
\forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(inv).\mathbf{0}]\!])) \parallel
\end{aligned}$$

$$\begin{aligned}&\forall\epsilon(buy = buy \to \forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \to \overline{\mathsf{rcv}(y, w)} \parallel \\ &\qquad\qquad [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!])) \parallel \\ &\forall\epsilon(buy = quit \to \forall w_2, u(\mathsf{snd}(w_2, u) \otimes \{w_2{:}y\} \to \overline{\mathsf{rcv}(y, u)} \parallel [\![y(w').\mathbf{0}]\!])))\end{aligned}$$

Let us now consider the output observables of S_1:

$$\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(S_1) = \{\exists x, y.\mathsf{snd}(x, 5406), \exists x, y.\mathsf{snd}(y, \mathsf{invoice})\}$$

For the sake of comparison, consider the reduction of P_{18}, using Rule $\lfloor\textsc{Sel}\rfloor$, which discards the second labeled branch:

$$P_{18} \longrightarrow (\boldsymbol{\nu} xy)(x\langle 5406\rangle.\, x(inv).\mathbf{0} \mid y(w).y\langle \mathsf{invoice}\rangle.\mathbf{0}) = P'_{18}$$

The translation of P'_{18} is as follows (we also show its reduction):

$$\begin{aligned}[\![P'_{18}]\!] = {}& \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x, 5406)} \parallel \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(inv).\mathbf{0}]\!]) \parallel \\ &\qquad \forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \to \overline{\mathsf{rcv}(y, w)} \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!])\big) \\ \longrightarrow_1 {}& \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(inv).\mathbf{0}]\!]) \parallel \overline{\mathsf{rcv}(y, 5406)} \parallel \\ &\qquad [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!]\big) = S'\end{aligned}$$

and it is easy to see that:

$$\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(S_1) = \mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}([\![P'_{18}]\!]) = \{\exists x, y.\mathsf{snd}(x, 5406), \exists x, y.\mathsf{snd}(y, \mathsf{invoice})\}$$

We now show that $S_1 \longrightarrow_1^2 S_2$ and $\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(S_1) = \mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(S_2)$. This step illustrates the fact that intermediate steps reduce to processes that are o-barbed congruent to their respective translations:

$$\begin{aligned}S_1 \longrightarrow_1 {}& \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x, 5406)} \parallel \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(inv).\mathbf{0}]\!]) \parallel \\ &\qquad \forall\epsilon(buy = buy \to \forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \to \overline{\mathsf{rcv}(y, w)} \parallel \\ &\qquad\qquad [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!])) \parallel \\ &\qquad \forall\epsilon(buy = quit \to \forall w_2, u(\mathsf{snd}(w_2, u) \otimes \{w_2{:}y\} \to \overline{\mathsf{rcv}(y, u)} \parallel [\![y(w').\mathbf{0}]\!]))\big) \\ \longrightarrow_1 {}& \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x, 5406)} \parallel \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(inv).\mathbf{0}]\!]) \parallel \\ &\qquad \forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \to \overline{\mathsf{rcv}(y, w)} \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!]) \parallel \\ &\qquad \forall\epsilon(buy = quit \to \forall w_2, u(\mathsf{snd}(w_2, u) \otimes \{w_2{:}y\} \to \overline{\mathsf{rcv}(y, u)} \parallel \\ &\qquad\qquad [\![y(w').\mathbf{0}]\!]))\big) = S_2\end{aligned}$$

where:

$$\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(S_1) = \mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(S_2) = \mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}([\![P'_{18}]\!]) = \{\exists x, y.\mathsf{snd}(x, 5406), \exists x, y.\mathsf{snd}(y, \mathsf{invoice})\}$$

We now informally argue that $S_2 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P'_{18}]\!]$. First, we show that $S_2 \approx_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P'_{18}]\!]$ and then argue that for every $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\mathcal{C}$-context $C[-]$, $C[S_2] \approx_1^{\pi_{\mathsf{OR}}^{\lightning}} C[[\![P'_{18}]\!]]$. To justify the former claim, consider the relation

$$\mathcal{R} = \{(S_2, [\![P'_{18}]\!]), (S_3, S'), (S_4, S'_1), (S_5, S'_2), (S_6, S'_3)\}$$

where:

$$\begin{aligned}
S_2 &\longrightarrow_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(inv).\mathbf{0}]\!]) \parallel \overline{\mathsf{rcv}(y, 5406)} \parallel \\
&\qquad\qquad [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!] \parallel \\
&\qquad\qquad \forall \epsilon(buy = quit \to \forall w_2, u(\mathsf{snd}(w_2, u) \otimes \{w_2{:}y\} \to \overline{\mathsf{rcv}(y, u)} \parallel [\![y(w').\mathbf{0}]\!]))\big) \\
&= S_3 \\
&\longrightarrow_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![x(inv).\mathbf{0}]\!] \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!] \parallel \\
&\qquad\qquad \forall \epsilon(buy = quit \to \forall w_2, u(\mathsf{snd}(w_2, u) \otimes \{w_2{:}y\} \to \overline{\mathsf{rcv}(y, u)} \parallel [\![y(w').\mathbf{0}]\!]))\big) \\
&= S_4 \\
&\longrightarrow_1 S_5 \\
&\longrightarrow_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathtt{tt}} \parallel \overline{\mathtt{tt}} \parallel \\
&\qquad\qquad \forall \epsilon(buy = quit \to \forall w_2, u(\mathsf{snd}(w_2, u) \otimes \{w_2{:}y\} \to \overline{\mathsf{rcv}(y, u)} \parallel [\![y(w).\mathbf{0}]\!]))\big) \\
&= S_6 \\
S' &\longrightarrow_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![x(inv).\mathbf{0}]\!] \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!]\big) = S'_1 \\
&\longrightarrow_1 S'_2 \\
&\longrightarrow_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathtt{tt}} \parallel \overline{\mathtt{tt}}\big) = S'_3
\end{aligned}$$

Notice that we have left out the expansion of the term $[\![x(inv).\mathbf{0}]\!] \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!]$ in both S'_1 and S_4. Moreover, for simplicity, we have omitted the shapes of S_5 and S'_2.

We can see that $\mathcal{R}$ is a weak o-barbed bisimulation. Now, to prove $S \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P'_{18}]\!]$ we need to show that for each $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\mathcal{C}$-context there exists a weak o-barbed bisimulation that makes the processes equivalent. Def. 2.32 ensures that contexts can only be formed with $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\mathcal{C}$-processes. Hence, we need to only check processes that add/consume constraints that may in turn trigger new process reductions. To see this point, consider process S_6: a context that adds the (inconsistent) constraint $l_1 = l_2$ would wrongly trigger a behavior excluded by the source reduction of Q into Q'. One key point in our formal development concerns excluding this possibility (see Def. 4.2). △

The Shape of Translated Programs

We now introduce so-called *process enablers* for $\pi_{\mathsf{OR}}^{\lightning}$; intuitively, they represent all the necessary endpoint connections for an $\pi_{\mathsf{OR}}^{\lightning}$ process to reduce.

Definition 4.11 (Enablers for $\pi_{\mathsf{OR}}^{\lightning}$ Processes). Let P be a $\pi_{\mathsf{OR}}^{\lightning}$ process. We say that the vectors of variables $\widetilde{x}, \widetilde{y}$ *enable* P if there is some P' such that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})P \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})P'$.

The enablers of a process lead to an $\pi_{\mathsf{OR}}^{\lightning}$ evaluation context $E[-] = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(-)$ (cf. Def. 2.12). The translation of context $E[-]$ is so common that we introduce the following notation for it:

Notation 4.12. Let $E[-] = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(-)$ be an $\pi_{\mathsf{OR}}^{\lightning}$ context, as in Def. 2.12. We will write

$C_{\widetilde{x}\widetilde{y}}[-]$ to denote the `lcc` translation of E:

$$[\![E[-]]\!] \overset{\mathsf{def}}{=} \exists \widetilde{x}, \widetilde{y}. \big(\overline{! \bigotimes_{\substack{x_i \in \widetilde{x}, \\ y_i \in \widetilde{y}}} \{x_i{:}y_i\}} \parallel - \big)$$

We restrict our attention to well-typed programs (Not. 2.21). Programs encompass "complete" protocol implementations, i.e., processes that contain all the parties and sessions required in the system. Considering programs is convenient because their syntax facilitates reasoning about their behavior. The first invariant of our translation concerns the shape of translated $\pi_{\mathsf{OR}}^{\lightning}$ programs:

Lemma 4.13 (Translated Form of a Program). *Let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program (cf. Not. 2.21). Then*

$$[\![P]\!] \equiv C_{\widetilde{x}\widetilde{y}}[[\![R_1]\!] \parallel \cdots \parallel [\![R_n]\!]]$$

where $n \geq 1$ and $x_1, \ldots, x_n \in \widetilde{x}$, $y_1, \ldots, y_n \in \widetilde{y}$. Note that each $R_i, 1 \leq i \leq n$ is a pre-redex (Def. 2.19) or a conditional process in P.

Proof. Follows directly from Def. 3.14 and Fig. 4.2. □

Junk Processes

It is common for translations to induce *junk processes* that do not add any meaningful (source) behavior to translated processes. In our setting, junk processes behave like $\overline{\mathtt{tt}}$, modulo $\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}$. Junk can be characterized syntactically: they result as leftovers of the translation of conditionals and branching constructs.

Definition 4.14 (Junk). Let P and J be `lcc` processes. Also, let b be a boolean and l_i, l_j be two distinct labels. We say that J is junk, if it belongs to the following grammar:

$$J, J' ::= \forall \epsilon((b = \neg b) \rightarrow P) \mid \forall \epsilon((l_j = l_i) \rightarrow P) \mid \overline{\mathtt{tt}} \mid J \parallel J'$$

The following statements will explain why junk processes cannot introduce any observable behavior in translated processes. This entails showing that for any junk J, $J \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} \overline{\mathtt{tt}}$. The proof is divided in three statements: (1) we show that no constraint in the session constraint system (Def. 4.1) allows a junk process to reduce; (2) we show that junk processes cannot reduce and that $\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(J) = \mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(\overline{\mathtt{tt}})$; finally, (3) we prove that J and $\overline{\mathtt{tt}}$ are behaviorally equivalent under any $\mathcal{DE}$-context (cf. Def. 2.32).

Lemma 4.15. *Let J be junk. Then: (1) $J \not\xrightarrow{\tau}_1$ (and) (2) there is no $c \in \mathcal{C}$ (cf. Def. 4.1) such that $J \parallel \overline{c} \xrightarrow{\tau}_1$.*

Proof. We prove each item individually. All of them follow by induction on the structure of J. For details see App. B.1. □

Lemma 4.16 (Junk Observables). *For every junk process J and every $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\mathcal{C}$-context $C[-]$, we have that:*

1. $\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(J) = \emptyset$ *(and)*

2. $\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(C[J]) = \mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}}(C[\overline{\mathtt{tt}}])$.

Proof. We prove each item separately. Each item follows by induction on the structure of J. For details see App. B.1 □

Lemma 4.17 (Junk Behavior). *For every junk J, every $\mathcal{D}_{\pi_{\mathsf{OR}}^{\lightning}}\mathcal{C}$-context $C[-]$, and every process P, we have $C[P \parallel J] \approx_1^{\pi_{\mathsf{OR}}^{\lightning}} C[P]$.*

Proof. By coinduction, i.e., by exhibiting a weak o-barbed bisimulation containing $(C[P \parallel J], C[P])$. For details see App. B.1. □

Corollary 4.18. *For every junk process J (cf. Def. 4.14) and every $\mathtt{lcc}$ process P, we have $P \parallel J \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} P$.*

Cor. 4.18 follows directly from Lem. 4.17 and Def. 2.34. Also, notice that non-trivial junk processes only appear as a byproduct of the translation of branching/selection processes and conditionals; other forms of synchronization do not generate junk. This is shown in the following lemma:

Lemma 4.19 (Occurrences of Junk). *Let R be a redex (Def. 2.19).*

1. *If $R = x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i \in I}$, with $j \in I$ then: $[\![(\boldsymbol{\nu} xy)R]\!] \xrightarrow{\tau}{}_1^3 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![Q_j]\!] \parallel J\big)$, where $J = \prod_{i \in I'} \forall \epsilon(l_j = l_i \rightarrow [\![Q_i]\!])$, with $I' = I \setminus \{j\}$, and $\exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![Q_j]\!] \parallel J\big) \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![Q_j]\!]\big)$.*
2. *If $R = b?\,(P_1) : (P_2)$, $b \in \{\mathtt{tt}, \mathtt{ff}\}$, then $[\![R]\!] \xrightarrow{\tau}{}_1 [\![P_i]\!] \parallel J$, $i \in \{1, 2\}$ with $J = \forall \epsilon(b = \neg b \rightarrow [\![P_j]\!])$, $j \neq i$, and $[\![P_i]\!] \parallel J \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P_i]\!]$.*
3. *If $R = x\langle v\rangle.P \mid y(z).Q$ then $[\![(\boldsymbol{\nu} xy)R]\!] \xrightarrow{\tau}{}_1^2 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} \exists x, y.\big([\![P]\!] \parallel [\![Q\{^v\!/z\}]\!] \parallel J\big)$ with $J = \overline{\mathtt{tt}}$.*
4. *If $R = x\langle v\rangle.P \mid *\, y(z).Q$ then $[\![(\boldsymbol{\nu} xy)R]\!] \xrightarrow{\tau}{}_1^2 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} \exists x, y.\big([\![P]\!] \parallel [\![Q\{^v\!/z\}]\!] \parallel [\![*\, y(z).P]\!] \parallel J\big)$ with $J = \overline{\mathtt{tt}}$.*

Proof. Each item follows from the translation definition (cf. Def. 4.4 and Fig. 4.2). Items (1) and (2) refer to reductions that induce junk (no junk is generated in Items (3) and (4)); those cases rely on the definition of weak o-barbed congruence (cf. Def. 4.3) and Cor. 4.18. For details see App. B.1. □

4.3.2 Operational Completeness

Below, we present the operational completeness result for $[\![\cdot]\!]$. Notice that no further machinery is required for the proof.

Theorem 4.20 (Completeness for $[\![\cdot]\!]$). *Let $[\![\cdot]\!]$ be the translation in Def. 4.4. Also, let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program. Then, if $P \longrightarrow^* Q$ then $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$.*

Proof. By induction on the length of the reduction $\longrightarrow^*$, with a case analysis on the last applied rule. The base case is whenever $P \longrightarrow^0 P$, and it is trivially true since $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 [\![P]\!]$. For the inductive step, assume by IH, that $P \longrightarrow^* P_0 \longrightarrow Q$ and that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P_0]\!]$. We then have to prove that $[\![P_0]\!] \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$. There are nine cases since cases for Rules (Res), (Par) and (Str) are immediate by IH. We only detail the case for Rule $\lfloor$Repl$\rfloor$ as all the other cases are similar. For additional details see App. B.2.

Rule $\lfloor$Repl$\rfloor$:

1. Assume $P_0 = (\boldsymbol{\nu}xy)(x\langle v\rangle.P' \mid *\, y(z).P'' \mid S)$, with S collecting all the processes that may contain x and y. Notice that by typing, S can only contain (replicated) input processes on y.
2. By (1) $P_0 \longrightarrow (\boldsymbol{\nu}xy)(P' \mid P''\{v/z\} \mid *\, y(z).P'' \mid S) = Q$ using Rule $\lfloor$Rep$\rfloor$.
3. By definition of $[\![\cdot]\!]$:

$$\begin{aligned}
[\![P_0]\!] &= \exists x,y.\big(\overline{!\{x{:}y\}} \parallel \overline{(\mathsf{snd}(x,v)} \parallel \forall z_1((\mathsf{rcv}(z_1,v)\otimes\{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad !\forall z_2,w(\mathsf{snd}(w,z_2)\otimes\{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y,z_2)} \parallel [\![P'']\!]) \parallel [\![S]\!]\big) \\
&\equiv \exists x,y.\big(\overline{!\{x{:}y\}} \parallel \overline{(\mathsf{snd}(x,v)} \parallel \forall z_1((\mathsf{rcv}(z_1,v)\otimes\{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad \forall z_2,w(\mathsf{snd}(w,z_2)\otimes\{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y,z_2)} \parallel [\![P'']\!])) \parallel \\
&\qquad !\forall z_2,w(\mathsf{snd}(w,z_2)\otimes\{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y,z_2)} \parallel [\![P'']\!]) \parallel [\![S]\!]\big)
\end{aligned}$$

4. Let $R = !\forall z_2,w(\mathsf{snd}(w,z_2)\otimes\{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y,z_2)} \parallel [\![P'']\!]))$. By using the rules of structural congruence and reduction of $\mathtt{lcc}$ the following transitions can be shown:

$$\begin{aligned}
[\![P_0]\!] &\equiv \exists x,y.\big(\overline{!\{x{:}y\}\otimes\mathsf{snd}(x,v)} \parallel \forall z_1((\mathsf{rcv}(z_1,v)\otimes\{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad \forall z_2,w(\mathsf{snd}(w,z_2)\otimes\{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y,z_2)} \parallel [\![P'']\!])) \parallel R \parallel [\![S]\!]\big) \\
&\xrightarrow{\tau}_1 \exists x,y.\big(\overline{!\{x{:}y\}} \parallel \forall z_1((\mathsf{rcv}(z_1,v)\otimes\{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \overline{\mathsf{rcv}(y,v)} \parallel \\
&\qquad [\![P''\{v,x/z_2,w\}]\!]) \parallel R \parallel [\![S]\!]\big) \\
&\equiv \exists x,y.\big(\overline{!\{x{:}y\}\otimes\mathsf{rcv}(y,v)} \parallel \forall z_1((\mathsf{rcv}(z_1,v)\otimes\{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad [\![P''\{v,x/z_2,w\}]\!] \parallel R \parallel [\![S]\!]\big) \\
&\xrightarrow{\tau}_1 \exists x,y.\big(\overline{!\{x{:}y\}} \parallel [\![P'\{y/z_1\}]\!] \parallel [\![P''\{v,x/z_2,w\}]\!] \parallel R \parallel [\![S]\!]\big)
\end{aligned}$$

5. By Fig. 4.2:

$$\begin{aligned}
&\exists x,y.\big(\overline{!\{x{:}y\}} \parallel [\![P'\{y/z_1\}]\!] \parallel [\![P''\{v,x/z_2,w\}]\!] \parallel R \parallel [\![S]\!]\big) \\
&\quad = \exists x,y.\big(\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P''\{v/z_2\}]\!] \parallel R \parallel [\![S]\!]\big)
\end{aligned}$$

   since $w \notin \mathsf{fv}(P'')$ and $z_1 \notin \mathsf{fv}(P'')$
6. Finally, observe that:

$$\begin{aligned}
[\![Q]\!] &= [\![(\boldsymbol{\nu}xy)(P' \mid P''\{v/z\} \mid *\, y(z).P'' \mid S)]\!] \\
&= \exists x,y.\big((\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P''\{v/z_2\}]\!] \parallel R \parallel [\![S]\!])\big)
\end{aligned}$$

□

4.3.3 Operational Soundness

In this section we prove that $[\![\cdot]\!]$ is operationally sound. Since this proof is rather involved, we first present the statement and sketch the proof informally. Then, we develop all the details.

Theorem 4.21 (Soundness for $[\![\cdot]\!]$). *Let $[\![\cdot]\!]$ be the translation in Def. 4.4. Also, let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program. For every S such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S$ there are Q, S' such that $P \longrightarrow^* Q$ and $S \overset{\tau}{\Longrightarrow}_1 S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$.*

We shall refer to the class of `lcc` processes that is induced by $[\![\cdot]\!]$ as *target terms* (cf. Def. 4.7). Thus, Thm. 4.21 will ensure that target terms never exhibit behavior that can not be attributed to some $\pi_{\mathsf{OR}}^{\lightning}$ process.

The proof of operational soundness requires additional technical machinery. The main challenge is to precisely identify which $\pi_{\mathsf{OR}}^{\lightning}$ process is being mimicked at a given point, and then to prove that this representation does not add undesired behaviors. To tackle this issue, we draw inspiration from [PN12], which characterizes translations semantically by defining pre-processing and post-processing steps, according to the effect they have over translated source terms and the simulation of the behavior of the source language.

We will characterize `lcc` processes semantically, via *complete* and *output observables* (cf. Def. 4.2). Then, to analyze translated processes we will define sets of *immediate observables*, which will enable us to characterize both translated processes and so-called *intermediate redexes* resulting from them (cf. Def. 4.25). These new observables will only contain barbs up to structural congruence, rather than to τ-transitions. Given a redex R, its set of intermediate redexes is denoted $\{\![R]\!\}$, with elements denoted $(\![R]\!)^{k}_{\widetilde{x}\widetilde{y}}$.

Since the proof of operational soundness requires several auxiliary results, we first give a high-level description of the proof argument before presenting the proof in full detail.

Proof Sketch for Thm. 4.21

We rely crucially on two lemmas (Lem. 4.33 and Lem. 4.34), which show how the shape of $\pi_{\mathsf{OR}}^{\lightning}$ processes can be inferred from their corresponding `lcc` translated process, via immediate observables (cf. Def. 4.22).

We also use several properties of target terms (cf. Def. 4.7). This is achieved by comparing the constraint stores of target terms after a τ-transition (cf. Not. 4.29). We use the fact that target terms have a specific shape (cf. Def. 4.30). Using Lem. 4.31, by recognizing the shape of the constraint store we identify an originating `lcc` process. Notice that junk will be removed via Cor. 4.35.

More in details, the proof of Thm. 4.21 is by induction on n, the length of the reduction $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_1$. The base case $(n = 0)$ is immediate; for the inductive step $(n > 0)$ we proceed as follows:

(1) Since $n \geq 1$, there exists a target term S_0 such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_0 \longrightarrow_1 S_1$.

(2) By IH, there exist Q_0 and S_0' such that $P \longrightarrow^* Q_0$ and $S_0 \overset{\tau}{\Longrightarrow}_1 S_0'$, with $S_0' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q_0]\!]$. Then, by Lem. 4.48, the sequence of transitions $S_0 \overset{\tau}{\Longrightarrow}_1 S_0'$ is executing actions that correspond to closing labels in the labeled semantics in Fig. 4.7.

(3) By Lem. 4.30, we have that $S_0 = C_{\widetilde{x}\widetilde{y}}[S'_1 \parallel \cdots \parallel S'_n \parallel J]$, where $S'_i = [\![R_i]\!]$ or $S'_i = (\!|R_i|\!)^k_{\widetilde{x}\widetilde{y}}$ for some R_i, $k \in \{1, 2, 3\}$.

(4) We analyze the transition $S_0 \longrightarrow_1 S_1$ in Item (1). By Item (3), S_0 has a specific shape and so will S_1. There are then two possible shapes for the transition, depending on whether one or two components evolve (we ignore junk processes involved, thanks to Lem. 4.19):

   (a) $C_{\widetilde{x}\widetilde{y}}[S'_1 \parallel \cdots \parallel S'_h \parallel \cdots \parallel S'_n] \longrightarrow_1 C_{\widetilde{x}\widetilde{y}}[S'_1 \parallel \cdots \parallel S''_h \parallel \cdots \parallel S'_n]$

   (b) $C_{\widetilde{x}\widetilde{y}}[S'_1 \parallel \cdots \parallel S'_{h_1} \parallel \cdots \parallel S'_{h_2} \parallel \cdots \parallel S'_n] \longrightarrow_1 C_{\widetilde{x}\widetilde{y}}[S'_1 \parallel \cdots \parallel S''_{h_1} \parallel \cdots \parallel S''_{h_2} \parallel \cdots \parallel S'_n]$.

(5) In both cases, Lem. 4.33 and Lem. 4.34 will allow us to identify which source reduction (in Q_0) is being partially simulated by $S_0 \longrightarrow_1 S_1$. (It is 'partial' because an $\pi^{\lightning}_{\mathsf{OR}}$ reduction is mimicked by at least two transitions in $\texttt{lcc}$.) Hence, we can characterize the $\pi^{\lightning}_{\mathsf{OR}}$ process Q for which $Q_0 \longrightarrow Q$.

(6) We are left to show the existence of S'_1, given that $S_0 \longrightarrow_1 S_1$ and $S_0 \overset{\tau}{\Longrightarrow}_1 S'_0$ (Items (1) and (2), resp.). This follows from Lem. 4.47, which is a diamond property for $\texttt{lcc}$ processes induced by so-called *closing* and *opening* labeled the transitions (cf. Def. 4.41) and the shape of S_0 identified in Item (3), which is preserved in S_1 by Item (4). These facts combined ensure that the same transition from S_0 to S_1 can take place from S'_0. Therefore, there is an S'_1 such that $S'_0 \longrightarrow_1 S'_1$ and that the same transitions from S_0 to S'_0 can be made by S_1. Therefore, $S_1 \overset{\tau}{\Longrightarrow}_1 S'_1$.

(7) Finally, since $S'_0 \cong^{\pi^{\lightning}_{\mathsf{OR}}}_1 [\![Q_0]\!]$ (IH, Item (2)) and by the reduction and transition identified in Items (5) and (6), resp., we can infer that $S'_1 \cong^{\pi^{\lightning}_{\mathsf{OR}}}_1 [\![Q]\!]$.

Using this proof sketch as a guide, we now introduce all the ingredients of the proof.

Invariants for Translated Pre-Redexes and Redexes

In this section we introduce invariants for translated pre-redexes and redexes. First, we introduce the set of immediate observables of an $\texttt{lcc}$ process. Intuitively, this denotes the current store of a process (i.e., all the constraints that can be consumed in a single transition).

Definition 4.22 (Immediate Observables of an $\texttt{lcc}$ Process). Let P be an $\texttt{lcc}$ process and $\mathcal{C}$ be a set of constraints. The set of *immediate observables* of P up to $\mathcal{C}$, denoted $\mathcal{I}^{\mathcal{C}}(P)$, is defined in Fig. 4.4.

Note that the immediate observables are defined over a subset of $\texttt{lcc}$ processes. We leave out processes that are not induced by the translation, such as $!(P \parallel P)$ or $!(P + P)$. The definition is parametric in $\mathcal{C}$, which we will instantiate with the set $\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}$ of complete observables (cf. Def. 4.2).

We will analyze the translation using so-called *invariants*, i.e., properties that hold for every target term. Based on source $\pi^{\lightning}_{\mathsf{OR}}$ processes, we will define these invariants

$$
\begin{aligned}
\mathcal{I}^{\mathcal{C}}(\overline{c}) &\overset{\mathsf{def}}{=} \{c\}, \text{if } c \in \mathcal{C}\\
\mathcal{I}^{\mathcal{C}}(\forall \widetilde{x}(c \to P)) &\overset{\mathsf{def}}{=} \emptyset\\
\mathcal{I}^{\mathcal{C}}(P + Q) &\overset{\mathsf{def}}{=} \mathcal{I}^{\mathcal{C}}(P) \cup \mathcal{I}^{\mathcal{C}}(Q)\\
\mathcal{I}^{\mathcal{C}}(P \parallel Q) &\overset{\mathsf{def}}{=} \mathcal{I}^{\mathcal{C}}(P) \cup \mathcal{I}^{\mathcal{C}}(Q)\\
\mathcal{I}^{\mathcal{C}}(\exists \widetilde{z}.(P)) &\overset{\mathsf{def}}{=} \{\exists \widetilde{z}.c \mid c \in \mathcal{I}^{\mathcal{C}}(P)\}\\
\mathcal{I}^{\mathcal{C}}(!\, P) &\overset{\mathsf{def}}{=} \begin{cases} \emptyset & \text{if } P = \forall \widetilde{x}(c \to P)\\ \{c\} & \text{if } P = \overline{c}\\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.4: Immediate observables in $[\![\cdot]\!]$ (cf. Def. 4.22).

bottom-up, starting from translations of pre-redexes (i.e., a prefixed process that does not contain parallel composition at the top-level, cf. Def. 2.19), redexes, and translated programs. The first invariant will clarify how the immediate observables of the translation of some pre-redex P gives information about the nature of P itself. Notice that in the results below we use Not. 2.24.

Lemma 4.23 (Invariants of $[\![\cdot]\!]$ for Pre-Redexes and the Inaction). *Let P be a pre-redex or the inactive process in $\pi^{\lightning}_{\mathsf{OR}}$. Then the following properties hold:*

1. *If $\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![P]\!]) = \{\mathsf{snd}(x,v)\}$ then $P = x\langle v\rangle.P_1$, for some P_1.*
2. *If $\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![P]\!]) = \{\mathsf{sel}(x,l)\}$ then $P = x \triangleleft l.P_1$, for some P_1.*
3. *If $\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![P]\!]) = \{\mathtt{tt}\}$ then $P = \mathbf{0}$.*
4. *If $\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![P]\!]) = \emptyset$ then $P = \diamond\, y(z).P_1$ or $P = x \triangleright \{l_1 : P_i\}_{i\in I}$, for some P_i. Moreover, $[\![P]\!] \not\longrightarrow_1$.*

Proof. By assumption $P = \mathbf{0}$ or P is a pre-redex (Def. 2.19): $P = x\langle v\rangle.P_1$, $P = x \triangleleft l.P_1$, $P = y(z).P_1$, $P = *y(z).P_1$ or $P = x \triangleright \{l_1 : P_i\}_{i\in I}$. Given these six possible forms for P, we then check the immediate observables (cf. Def. 4.22) of their lcc translations (cf. Fig. 4.2):

$$
\begin{aligned}
\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![x\langle v\rangle.P_1]\!]) &= \{\mathsf{snd}(x,v)\} & \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![x(y).P_1]\!]) &= \emptyset\\
\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![x \triangleleft l.P_1]\!]) &= \{\mathsf{sel}(x,l)\} & \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![* x(y).P_1]\!]) &= \emptyset\\
\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![\mathbf{0}]\!]) &= \{\mathtt{tt}\} & \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![x \triangleright \{l_1 : P_i\}_{i\in I}]\!]) &= \emptyset
\end{aligned}
$$

This way, the thesis holds. □

When immediate observables do not provide enough information on the shape of a pre-redex (cf. Lem. 4.23(4)), we can characterize the minimal parallel context that induces immediate observables:

Lemma 4.24 (Invariants of $[\![\cdot]\!]$ for Input-Like Pre-Redexes). *Let P be a pre-redex such that $\mathcal{I}^{\mathcal{D}^\star_{\pi_{\mathsf{OR}}^{\lightning}}}([\![P]\!]) = \emptyset$. Then one of the following holds:*

1. *If $[\![P]\!] \parallel \overline{\mathsf{sel}(x, l_j) \otimes \{y{:}x\}} \xrightarrow{\tau}_1 S$ then $\mathsf{bra}(y, l_j) \in \mathcal{I}^{\mathcal{D}^\star_{\pi_{\mathsf{OR}}^{\lightning}}}(S)$ and $P = y \triangleright \{l_i : P_i\}_{i \in I}$, with $j \in I$.*

2. *If $[\![P]\!] \parallel \overline{\mathsf{snd}(x, v) \otimes \{y{:}x\}} \xrightarrow{\tau}_1 S$ then $\mathsf{rcv}(y, v) \in \mathcal{I}^{\mathcal{D}^\star_{\pi_{\mathsf{OR}}^{\lightning}}}(S)$ and $P = \diamond\, y(z).P_1$.*

Proof. Using the previous definitions. See App. B.3 for details. $\square$

We now define the set of *intermediate* lcc *redexes* of a communicating redex (cf. Def. 2.19). Intuitively, these are lcc processes obtained through the transitions of a target term:

Definition 4.25 (Intermediate Redexes). Let R be a communicating redex in $\pi_{\mathsf{OR}}^{\lightning}$ enabled by $\widetilde{x}, \widetilde{y}$. The *set of intermediate* lcc *redexes* of R, denoted $\{\![R]\!\}$, is defined as follows:

$$\begin{aligned}
\{\![x\langle v\rangle.P \,|\, y(z).Q]\!\} &\stackrel{\mathsf{def}}{=} \{\overline{\mathsf{rcv}(y,v)} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{z{:}x\} \to [\![P]\!]) \parallel [\![Q\{v/z\}]\!]\} \\
\{\![x\langle v\rangle.P \,|\, *\, y(z).Q]\!\} &\stackrel{\mathsf{def}}{=} \{\overline{\mathsf{rcv}(y,v)} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{z{:}x\} \to [\![P]\!]) \parallel [\![Q\{v/z\}]\!] \parallel \\
&\qquad [\![*y(w).Q]\!]\} \\
\{\![x \triangleleft l.P \,|\, y \triangleright \{l_i : Q_i\}_{i\in I}]\!\} &\stackrel{\mathsf{def}}{=} \{\overline{\mathsf{bra}(y,l_j)} \parallel \forall z(\mathsf{bra}(z,l_j) \otimes \{z{:}x\} \to [\![P]\!]) \parallel \\
&\qquad \forall \epsilon(l_j = l_j \to [\![Q_j]\!]) \parallel J, \\
&\qquad [\![P]\!] \parallel \forall \epsilon(l_j = l_j \to [\![Q_j]\!]) \parallel J, \\
&\qquad \overline{\mathsf{bra}(y,l_j)} \parallel \forall z(\mathsf{bra}(z,l_j) \otimes \{z{:}x\} \to [\![P]\!]) \parallel [\![Q_j]\!] \parallel J \,\big|\, \\
&\qquad J \text{ as in Def. 4.14}\}
\end{aligned}$$

Observe that the set of intermediate redexes is a singleton, except for the translation of selection and branching. We now introduce a convenient notation for these redexes:

Notation 4.26. We denote the elements of $\{\![R]\!\}$ as $(\!|R|\!)^k_{\widetilde{x}\widetilde{y}}$, with $k \in \{1, 2, 3\}$ as in Fig. 4.5.

This notation aims to clarify the behavior of intermediate redexes, particularly in the case of the selection and branching. This will become much more apparent in the following invariant, which describes how translated redexes interact.

Lemma 4.27 (Invariants for Redexes and Intermediate Redexes). *Let R be a redex enabled by $\widetilde{x}, \widetilde{y}$, such that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'$. Then one of the following holds:*

1. *If $R \equiv_{\mathsf{S}} v?\,(P_1) : (P_2)$ and $v \in \{\mathtt{tt}, \mathtt{ff}\}$, then $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R]\!] \xrightarrow{\tau}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} (\boldsymbol{\nu}\widetilde{x}\widetilde{y})[\![P_i]\!]$, with $i \in \{1, 2\}$.*

$$\llparenthesis R \rrparenthesis^{1}_{\widetilde{x}\widetilde{y}} \overset{\mathsf{def}}{=} \begin{cases} \overline{\mathsf{rcv}(y,v)} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{z{:}x\} \rightarrow \llbracket P \rrbracket) \parallel \\ \llbracket Q\{v/x\} \rrbracket & \text{if } R = x\langle v\rangle.P \mid y(z).Q \\ \overline{\mathsf{rcv}(y,v)} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{z{:}x\} \rightarrow \llbracket P \rrbracket) \parallel \\ \llbracket Q\{v/x\} \rrbracket \parallel \llbracket * y(w).Q \rrbracket & \text{if } R = x\langle v\rangle.P \mid \ * y(z).Q \\ \overline{\mathsf{bra}(y,l_j)} \parallel \forall z(\mathsf{bra}(z,l_j) \otimes \{z{:}x\} \rightarrow \llbracket P \rrbracket) \parallel \\ \forall \epsilon(l_j = l_j \rightarrow \llbracket Q_j \rrbracket) \parallel J & \text{if } R = x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \end{cases}$$

$$\llparenthesis R \rrparenthesis^{2}_{\widetilde{x}\widetilde{y}} \overset{\mathsf{def}}{=} \begin{cases} \llbracket P \rrbracket \parallel \forall \epsilon(l_j = l_j \rightarrow \llbracket Q_j \rrbracket) \parallel J & \text{if } R = x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llparenthesis R \rrparenthesis^{3}_{\widetilde{x}\widetilde{y}} \overset{\mathsf{def}}{=} \begin{cases} \overline{\mathsf{bra}(y,l_j)} \parallel \\ \forall z(\mathsf{bra}(z,l_j) \otimes \{z{:}x\} \rightarrow \llbracket P \rrbracket) \parallel \llbracket Q_j \rrbracket \parallel J & \text{if } R = x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Figure 4.5: Elements in the set of intermediate redexes for $\llbracket \cdot \rrbracket$ (cf. Not. 4.26)

$$\begin{tikzcd}
 & & C_{\widetilde{x}\widetilde{y}}[\llparenthesis R \rrparenthesis^{2}_{\widetilde{x}\widetilde{y}}] \arrow[dr, "\tau"] & \\
\llbracket(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R\rrbracket \arrow[r, "\tau"] & C_{\widetilde{x}\widetilde{y}}[\llparenthesis R \rrparenthesis^{1}_{\widetilde{x}\widetilde{y}}] \arrow[ur, "\tau"] \arrow[dr, "\tau"] & & S \cong^{\pi_{\mathsf{OR}}^{\lightning}}_{1} \llbracket(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'\rrbracket \\
 & & C_{\widetilde{x}\widetilde{y}}[\llparenthesis R \rrparenthesis^{3}_{\widetilde{x}\widetilde{y}}] \arrow[ur, "\tau"] &
\end{tikzcd}$$

Figure 4.6: Lem. 4.27(3).

2. *If* $R \equiv_{\mathsf{S}} x\langle v\rangle.P \mid \diamond\, y(w).Q$, *then* $\llbracket(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R\rrbracket \longrightarrow_1 \equiv C_{\widetilde{x}\widetilde{y}}[\llparenthesis R \rrparenthesis^{1}_{\widetilde{x}\widetilde{y}}] \longrightarrow_1 \cong^{\pi_{\mathsf{OR}}^{\lightning}}_{1} \llbracket(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'\rrbracket$.
3. *If* $R \equiv_{\mathsf{S}} x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$, *with* $j \in I$, *then we have the reductions in Fig. 4.6.*

Proof. This proof proceeds by using the translation (cf. Fig. 4.2) the `lcc` semantics (cf. Fig. 2.5). For details see App. B.3. □

The following corollary states that every intermediate redex reduces to some target term; it follows directly from Lem. 4.27 and Def. 4.25.

Corollary 4.28. *For every intermediate redex* $S \in \{\![R]\!\}$ *(cf. Def. 4.25), there exist some* $\pi_{\mathsf{OR}}^{\lightning}$ *process* R' *and* $k \in \{1,2\}$ *such that* $S \longrightarrow^{k}_{1} \llbracket R' \rrbracket$ *and* $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'$.

We now introduce some useful notation to be used in later proofs. In particular, we introduce shorthands for the set of immediate observables of a target term.

Notation 4.29. We define the following conventions:

- $\mathcal{I}_S$ will be a short-hand notation for the set $\mathcal{I}^{\mathcal{D}^{\star}_{\pi_{\mathsf{OR}}^{\lightning}}}(S)$ (cf. Def. 4.22).
- By a slight abuse of notation, we will write $c_{\widetilde{z}} \in \mathcal{I}_S$ instead of $\exists \widetilde{z}.c \in \mathcal{I}_S$.

Notice that this notation conveniently captures the constraints that are consumed as a result of a τ-transition. In turn, such consumed constraints will allow us to recognize which $\pi_{\mathsf{OR}}^{\lightning}$ process is simulated by the translation. In particular, observe that every τ-transition of a target term modifies the constraint store (and the immediate observables) in a specific way: it will either (i) generate new constraints (if the transition is induced by the translation of conditionals and labeled choices) or (ii) consume some existing constraints (if the transition is induced by other kinds of source synchronizations). While case (i) will be formalized by Lem. 4.33, case (ii) will be covered by Lem. 4.34.

Invariants for Translated Well-Typed Programs

The following lemma will allow us to determine the syntactic structure of a given target term. It states that any target term corresponds to the parallel composition of the translation of processes, intermediate redexes and junk, all enclosed within a context that provides the required co-variable constraints. Given a program P, we say that R_k is a (pre)redex *reachable* from P if $P \longrightarrow^* (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(R_k \mid R)$, for some R.

Lemma 4.30. *Let P be a well-typed program. If $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S$ then*

$$S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n \parallel J]$$

where $n \geq 1$, J is some junk, and for all $i \in \{1, \ldots, n\}$ we have $U_i = \overline{\texttt{tt}}$ or one the following:

1. *$U_i = [\![R_k]\!]$, where R_k is a conditional redex (cf. Def. 2.19) reachable from P;*
2. *$U_i = [\![R_k]\!]$, where R_k is a pre-redex reachable from P;*
3. *$U_i \in \{\![R_k \mid R_j]\!\}$ (cf. Def. 4.25), where redex $R_k \mid R_j$ is reachable from P.*

Proof. By induction on the length k of the reduction $\overset{\tau}{\Longrightarrow}_1$. For details see App. B.4 □

The next lemma provides two insights: first, it gives a precise characterization of a target term whenever constraints are being added to the store and there is no constraint consumption. Second, it captures the fact that τ-transitions consume one constraint at a time.

Lemma 4.31. *Let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program. Then, for every S, S' such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S \xrightarrow{\tau}_1 S'$ one of the following holds:*

(a) *$\mathcal{I}_S \subseteq \mathcal{I}_{S'}$ (cf. Not. 4.29) and one of the following holds:*

1. *$S \equiv C_{\widetilde{x}\widetilde{y}}[[\![b?\,(P_1) \!:\! (P_2)]\!] \parallel U]$ and $S' = C_{\widetilde{x}\widetilde{y}}[[\![P_i]\!] \parallel U]$, with $i \in \{1, 2\}$;*
2. *$S \equiv C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$ and $S' = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{x}\widetilde{y}} \parallel U]$;*
3. *$S \equiv C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel U]$ and $S' = C_{\widetilde{x}\widetilde{y}}[[\![P']\!] \parallel [\![Q_j]\!] \parallel U]$.*

(b) *$\mathcal{I}_S \not\subseteq \mathcal{I}_{S'}$ and $|\mathcal{I}_S \setminus \mathcal{I}_{S'}| = 1$.*

Proof. We first use Lem. 4.30 to characterize every parallel sub-process U_i of S; then, by a case analysis on the shape of the U_i that originated the transition $S \xrightarrow{\tau}_1 S'$ it is shown how each case will fall under either (a) or (b). For details see App. B.4. □

Notice that the uniqueness of the consumed constraints is ensured by the typing of the $\pi_{\mathsf{OR}}^{\lightning}$ program P. In particular, for every constraint $\mathsf{snd}(x, v)$, $\mathsf{rcv}(x, v)$, $\mathsf{sel}(x, l)$, and $\mathsf{bra}(x, l)$ that is being added to the store in parallel, subject x can only appear once. This follows from the fact that typing ensures that target terms do not contain output races. For the translation, the absence of output races in source terms implies that at any given point during execution there can only be a single constraint $\mathsf{snd}(x, v)$ or $\mathsf{sel}(x, l)$ to be consumed by an abstraction. This abstraction in turn, will only generate a single constraint $\mathsf{rcv}(y, v)$ or $\mathsf{bra}(y, l)$, respectively.

Proposition 4.32. *Let $\gamma \in \{\mathsf{rcv}, \mathsf{snd}, \mathsf{sel}, \mathsf{bra}\}$ be a predicate as in Fig. 4.1 and m be a value or label. If S is a target term (cf. Def. 4.7) then $S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{c_1} \parallel \cdots \parallel \overline{c_n} \parallel Q]$ with $n \geq 1$, where Q only contains abstractions. Furthermore, for every $i, j \in \{1, \ldots, n\}$ such that $i \neq j$, $c_i = \gamma(x_1, m_1)$, and $c_j = \gamma(x_2, m_2)$, it holds that $x_1 \neq x_2$.*

Proof. The first part of the statement follows immediately by applying the structural congruence of lcc in S. The second part of the proof is by contradiction. We assume that $S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{c_1} \parallel \cdots \parallel \overline{c_n} \parallel Q]$, where Q only contain abstractions and that there exist $i, j \in \{1, \ldots, n\}$ such that $i \neq j$, $c_i = \gamma(x_1, m_1)$, $c_j = \gamma(x_2, m_2)$, and $x_1 = x_2$. We proceed by a case analysis on γ; there are four cases, we only show the cases $\gamma = \mathsf{snd}$ and $\gamma = \mathsf{rcv}$.
Suppose that $\gamma = \mathsf{snd}$. By assumption, $S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{c_1} \parallel \cdots \parallel \overline{\mathsf{snd}(x, v_1)} \parallel \cdots \parallel \overline{\mathsf{snd}(x, v_2)} \parallel \cdots \parallel \overline{c_n} \parallel Q]$; moreover, by Def. 4.7, S must come from the translation of a well-typed term. By Fig. 4.2, it must be the case that:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{c_1} \parallel \cdots \parallel [\![x\langle v_1\rangle.P_1]\!] \parallel \cdots \parallel [\![x\langle v_2\rangle.P_2]\!] \parallel \cdots \parallel \overline{c_n} \parallel Q']$$

for some Q' that does not contain the abstractions used to build the translations $[\![x\langle v_k\rangle.P_k]\!]$, $k \in \{1, 2\}$. This implies, by Fig. 4.2, that S comes from an $\pi_{\mathsf{OR}}^{\lightning}$ process that contains two outputs on the same channel x in parallel. This contradicts the well-formedness assumption that follows from Thm. 3.15 is violated, finishing the proof.
Suppose that $\gamma = \mathsf{rcv}$. The proof has the same structure as the one above. The only difference is that rather than the translation of output processes, we must consider intermediate redexes (cf. Fig. 4.5). Similarly as above, we then will find that the well-formedness assumption induced by typing is violated, thus reaching a contradiction. □

As already discussed, in mimicking the behavior of an $\pi_{\mathsf{OR}}^{\lightning}$ process, the constraint store of its corresponding target process in lcc may either add or consume constraints:

- Lem. 4.33, given below, covers the case where an lcc transition adds information to the store: by Lem. 4.31(a) the target term must then correspond to either the translation of a conditional redex or to an intermediate redex of a branching/selection interaction.

- Lem. 4.34 covers the case where the lcc transition consumes information in the constraint store (cf. by Lem. 4.31(b)).

As such, Lem. 4.33 and Lem. 4.34 cover the complete spectrum of possible transitions for target terms.

Lemma 4.33 (Invariants of Target Terms (I): Adding Information). *Let P be a well-typed $\pi^{\lightning}_{\mathsf{OR}}$ program. For any S, S' such that $[\![P]\!] \xRightarrow{\tau}_1 S \xrightarrow{\tau}_1 S'$ and $\mathcal{I}_S \subseteq \mathcal{I}_{S'}$ (cf. Not. 4.29) one of the following holds, for some U:*

1. *$S \equiv C_{\widetilde{z}}[[\![b?\,(P_1):(P_2)]\!] \parallel U \parallel J_1]$ and $S' = C_{\widetilde{z}}[[\![P_i]\!] \parallel \forall\epsilon(b = \neg b \rightarrow P_j) \parallel U \parallel J_1]$ with $i, j \in \{1,2\}, i \neq j$;*

2. *$[\![P]\!] \xRightarrow{\tau}_1 S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![x \triangleleft l_j.P' \parallel y \triangleright \{l_i\, Q_i\}_{i\in I}]\!] \parallel U \parallel J_1]$ and either:*

   (*a*) *All of the following hold:*

   (*i*) $S_0 \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U \parallel J_1] \xrightarrow{\tau}_1 S$,

   (*ii*) $S = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel U \parallel J_1]$ (*and*)

   (*iii*) $S' = C_{\widetilde{x}\widetilde{y}}[[\![P']\!] \parallel [\![Q_j]\!] \parallel U \parallel J_1 \parallel J_2]$.

   (*b*) *All of the following hold:*

   (*i*) $S_0 \xrightarrow{\tau}_1 S = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U \parallel J_1]$,

   (*ii*) $S' = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{x}\widetilde{y}} \parallel U \parallel J_1]$ (*and*)

   (*iii*) $S' \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[[\![P']\!] \parallel [\![Q_j]\!] \parallel U \parallel J_1 \parallel J_2]$.

   where $J_2 = \prod_{k\in I\setminus\{j\}} \forall\epsilon(l_j = l_k \rightarrow [\![P_k]\!])$.

Proof. By induction on the length of the transition $\xRightarrow{\tau}_1\xrightarrow{\tau}_1$. For detail see App. B.4. □

We state our next invariant. Notice that Lem. 4.31(b) clarifies the behavior of the immediate observables (cf. Def. 4.22) in a single transition whenever a constraint has been consumed.

Lemma 4.34 (Invariants of Target Terms (II): Consuming Information). *Let P be a well-typed $\pi^{\lightning}_{\mathsf{OR}}$ program. For any S, S' such that $[\![P]\!] \xRightarrow{\tau}_1 S \xrightarrow{\tau}_1 S'$ and $\mathcal{I}_S \not\subseteq \mathcal{I}_{S'}$ the following holds, for some U:*

1. *If $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{snd}(x_1, v)^k_{\widetilde{x}\widetilde{y}}\}$ then all the following hold:*

   (*a*) $S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x_1{:}y_1\}} \parallel [\![x_1\langle v\rangle.P_1 \mid \diamond\, y_1(z).P_2]\!] \parallel U]$;

   (*b*) $S' = C_{\widetilde{x}\widetilde{y}}[(\!|x_1\langle v\rangle.P_1 \mid \diamond\, y_1(z).P_2|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$;

   (*c*) $S' \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1 \mid P_2\{v/z\}]\!] \parallel S'' \parallel U]$, *where* $S'' = *\,[\![y(z).P_2]\!]$ *or* $S'' = \overline{\mathtt{tt}}$.

2. *If $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{rcv}(x_1, v)^k_{\widetilde{x}\widetilde{y}}\}$ then there exists S_0 such that $[\![P]\!] \xRightarrow{\tau}_1 S_0 \xrightarrow{\tau}_1 S$ and all of the following hold:*

   (*a*) $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x_1{:}y_1\}^m} \parallel [\![y_1\langle v\rangle.P_1 \mid \diamond\; x_1(z).P_2]\!] \parallel U]$;

   (*b*) $S = C_{\widetilde{x}\widetilde{y}}[(\!|y_1\langle v\rangle.P_1 \mid \diamond\, x_1(z).P_2|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$;

   (*c*) $S' = C_{\widetilde{x}\widetilde{y}}[[\![P_1 \mid P_2\{v/z\}]\!] \parallel S'_1 \parallel U]$, *where* $S'_1 = *\,[\![y(z).P_2]\!]$ *or* $S'_1 = \overline{\mathtt{tt}}$.

3. *If $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{sel}(x_1, l_j)^k_{\widetilde{x}\widetilde{y}}\}$ then all of the following hold:*

   (*a*) $S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x_1{:}y_1\}} \parallel [\![x_1 \triangleleft l.P_1 \mid y_1 \triangleright \{l_i : P_i\}_{i\in I}]\!] U]$;

(*b*) $S' = C_{\widetilde{x}\widetilde{y}}[(\!|x_1 \triangleleft l.P_1 \mid y_1 \triangleright \{l_i : P_i\}_{i \in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$;

(*c*) $S_1 \xrightarrow{\tau}_1^2 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}\widetilde{y}}[[\![P_1 \mid P_j]\!] \parallel U']$, *with* $U' \equiv U \parallel \prod_{h \in I} \forall \epsilon(l_h = l_j \rightarrow [\![Q_h]\!])$.

4. *If* $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{bra}(x, l_j)^k_{\widetilde{x}\widetilde{y}}\}$, *then there exists* $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![x \triangleleft l_j.Q \mid y \triangleright \{l_i\ Q_i\}_{i \in I}]\!] \parallel U]$ *such that* $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_0$ *and either:*

   (*a*) *All of the following hold:*

   (*i*) $S_0 \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.Q \mid x \triangleright \{l_i : Q_i\}_{i \in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U] \xrightarrow{\tau}_1 S$,

   (*ii*) $S = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.Q \mid x \triangleright \{l_i : Q_i\}_{i \in I}|\!)^3_{\widetilde{x}\widetilde{y}} \parallel U]$ (*and*)

   (*iii*) $S' = C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![Q \mid Q_j]\!] \parallel U']$.

   (*b*) *All of the following hold:*

   (*i*) $S_0 \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P \mid x \triangleright \{l_i : Q_i\}_{i \in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U] \equiv S$,

   (*ii*) $S' = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P \mid x \triangleright \{l_i : Q_i\}_{i \in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel U]$ (*and*)

   (*iii*) $S' \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![P \mid Q_j]\!] \parallel U']$.

   with $U' \equiv U \parallel \prod_{h \in I} \forall \epsilon(l_h = l_j \rightarrow [\![Q_h]\!])$.

Proof. By induction on the transition $\overset{\tau}{\Longrightarrow}_1 \xrightarrow{\tau}_1$. For details see App. B.4. □

By combining previous results we obtain the following corollary, which allows us to remove junk processes from any target term.

Corollary 4.35. *Let* P *be a well-typed* $\pi_{\mathsf{OR}}^{\lightning}$ *program. If* $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S$ *then there exist* S' *and* J *such that* $S = C_{\widetilde{x}\widetilde{y}}[S' \parallel J] \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}y}[S']$.

Proof. Since P is well-typed, by Lem. 4.13, $[\![P]\!] = C_{\widetilde{x}\widetilde{y}}[[\![P']\!]]$. By applying Lem. 4.33 and Lem. 4.34, we know that for every S, such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S$ it holds that $S = C_{\widetilde{x}\widetilde{y}}[S' \parallel J]$, where $S' = U_1 \parallel \cdots \parallel U_n$, $n \geq 1$, with $U_i = [\![R_i]\!]$ for some $\pi_{\mathsf{OR}}^{\lightning}$ pre-redex R_i or $U_i = (\!|R_i|\!)^k_{\widetilde{x}\widetilde{y}}$, for some $\pi_{\mathsf{OR}}^{\lightning}$ pre-redex R_i and $k \in \{1, 2, 3\}$. Finally, by Cor. 4.18, we can conclude that $C_{\widetilde{x}\widetilde{y}}[S' \parallel J] \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}y}[S']$. □

A Diamond Property for Target Terms

Proving soundness for our translation involves proving a sort of *diamond property* over target terms. First, we present how our translation captures the nondeterminism allowed by typing in $\pi_{\mathsf{OR}}^{\lightning}$.

Example 4.36. Let us recall process P_{17} from Ex. 4.6, which is well-typed:

$$P_{17} = (\boldsymbol{\nu} xy)(x\langle v_1 \rangle.Q_1 \mid * y(z_1).Q_2 \mid * y(z_2).Q_3)$$

this process is not confluent if $Q_2 \neq Q_3$, since there are two possible reductions:

$$P_{17} \longrightarrow (\boldsymbol{\nu} xy)(Q_1 \mid Q_2\{v_1/z_1\} \mid * y(z_2).Q_3 \mid * y(z_1).Q_2)$$
$$P_{17} \longrightarrow (\boldsymbol{\nu} xy)(Q_1 \mid * y(z_1).Q_2 \mid Q_3\{v_1/z_2\} \mid * y(z_2).Q_3)$$

Its translation is as follows:

$$\begin{aligned}\llbracket P_7 \rrbracket = C_{xy}[&\overline{\mathsf{snd}(x, v_1)} \parallel \forall z'\big(\mathsf{rcv}(z', v_1) \otimes \{x{:}z'\} \rightarrow \llbracket Q_1 \rrbracket\big) \parallel \\ &!\,\forall z_1, w\big(\mathsf{snd}(w, z_1) \otimes \{y{:}w\} \rightarrow \overline{\mathsf{rcv}(y, z_1)} \parallel \llbracket Q_2 \rrbracket\big) \parallel \\ &!\,\forall z_2, w'\big(\mathsf{snd}(w', z_2) \otimes \{y{:}w'\} \rightarrow \overline{\mathsf{rcv}(y, z_2)} \parallel \llbracket Q_3 \rrbracket\big)]\end{aligned}$$

and the following transitions are possible:

$$\begin{aligned}\llbracket P_{17} \rrbracket \longrightarrow_1 C_{xy}[&\forall z'\big(\mathsf{rcv}(z', v_1) \otimes \{x{:}z'\} \rightarrow \llbracket Q_1 \rrbracket\big) \parallel \\ &\overline{\mathsf{rcv}(y, v_1)} \parallel \llbracket Q_2 \rrbracket\{v_1/z_1\} \parallel !\,\forall z_2, w'\big(\mathsf{snd}(w', z_2) \otimes \{y{:}w'\} \rightarrow \llbracket Q_3 \rrbracket\big) \parallel \\ &!\,\forall z_1, w\big(\mathsf{snd}(w, z_1) \otimes \{y{:}w\} \rightarrow \llbracket Q_2 \rrbracket\big)] \\ \llbracket P_{17} \rrbracket \longrightarrow_1 C_{xy}[&\forall z'\big(\mathsf{rcv}(z', v_1) \otimes \{x{:}z'\} \rightarrow \llbracket Q_1 \rrbracket\big) \parallel \\ &!\,\forall z_1, w\big(\mathsf{snd}(w, z_1) \otimes \{y{:}w\} \rightarrow \llbracket Q_2 \rrbracket\big) \parallel \overline{\mathsf{rcv}(y, v_1)} \parallel \llbracket Q_3 \rrbracket\{v_2/z_2\} \parallel \\ &!\,\forall z_2, w'\big(\mathsf{snd}(w', z_2) \otimes \{y{:}w'\} \rightarrow \llbracket Q_3 \rrbracket\big)]\end{aligned}$$

Notice that they unequivocally correspond to the following intermediate processes, respectively:

$$\begin{aligned}\exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel (\!|x\langle v_1\rangle.Q_1 \mid *\,y(z_1).Q_2|\!)^1_{xy} \parallel \llbracket *\,y(z_2).Q_3 \rrbracket\big) = S_1 \\ \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel (\!|x\langle v_1\rangle.Q_1 \mid *\,y(z_2).Q_3|\!)^1_{xy} \parallel \llbracket *\,y(z_1).Q_2 \rrbracket\big) = S_1'\end{aligned}$$

In this case, the intermediate process corresponds to a 'committed' state in which there is only one process that can consume constraint $\mathsf{snd}(y, v_1)$, which forces the translation to finish the synchronization in the translation of the correct source process. △

Our diamond property concerns τ-transitions originating from intermediate processes from the same target term (cf. Def. 4.7). Informally speaking, this property, given by Lem. 4.48, confirms that τ-transitions originated from intermediate processes that reach the translation of some $\pi_{\mathsf{OR}}^{\lightning}$ process do not preclude the execution of translated terms. First, we define a notation for distinguishing τ-transitions:

Definition 4.37 (Labeled τ-Transitions for $\llbracket\cdot\rrbracket$ Target Terms). Let S be a target term (cf. Def. 4.7). Also, let $\{\mathtt{IO}, \mathtt{SL}, \mathtt{RP}, \mathtt{CD}, \mathtt{IO}_1, \mathtt{RP}_1, \mathtt{SL}_1, \mathtt{SL}_2, \mathtt{SL}_3\}$ be a set of labels ranged over by $\alpha, \alpha_1, \alpha_2, \alpha', \ldots$. We define labeled transition $\xrightarrow{\alpha}_1$ by the rules in Fig. 4.7. We assume that $U = U_1 \parallel \cdots \parallel U_n$ with $n \geq 0$.

Notation 4.38. We will write, e.g., $\alpha(x, y)$ to denote a named transition as in the previous definition. Moreover, for transitions due to a conditional expressions (without endpoints), we will write $\mathtt{CD}(-)$.

This notation serves to distinguish transitions depending on the action that originates it. For example, a transition simulates the first part of a synchronization between endpoints x, y will be denoted $\xrightarrow{\mathtt{IO}(x,y)}_1$; the completion of such synchronization is represented by transition $\xrightarrow{\mathtt{IO}_1(x,y)}_1$. An example follows:

$$\lfloor \mathrm{IO} \rfloor\ C_{\widetilde{x}\widetilde{y}}[[\![x\langle v\rangle.P_1]\!] \parallel [\![y(z).P_2]\!] \parallel U] \xrightarrow{\mathtt{IO}(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^1_{xy} \parallel U]$$

$$\lfloor \mathrm{RP} \rfloor\ C_{\widetilde{x}\widetilde{y}}[[\![x\langle v\rangle.P_1]\!] \parallel [\![* \, y(z).P_2]\!] \parallel U] \xrightarrow{\mathsf{RP}(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid * \, y(z).P_2|\!)^1_{xy} \parallel U]$$

$$\lfloor \mathrm{SL} \rfloor\ C_{\widetilde{x}\widetilde{y}}[[\![x \triangleleft l_j.P_1]\!] \parallel [\![y \triangleright \{l_i : P_i\}_{i\in I}]\!] \parallel U] \xrightarrow{\mathsf{SL}(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_j.P_1 \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^1_{xy} \parallel U]$$

$$\lfloor \mathrm{CDT} \rfloor\ C_{\widetilde{x}\widetilde{y}}[[\![\mathtt{tt}?\,(P_1)\!:\!(P_2)]\!] \parallel U] \xrightarrow{\mathsf{CD}(-)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel \forall\epsilon(\mathtt{tt} = \mathtt{ff} \to [\![P_2]\!]) \parallel U]$$

$$\lfloor \mathrm{CDF} \rfloor\ C_{\widetilde{x}\widetilde{y}}[[\![\mathtt{ff}?\,(P_1)\!:\!(P_2)]\!] \parallel U] \xrightarrow{\mathsf{CD}(-)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_2]\!] \parallel \forall\epsilon(\mathtt{ff} = \mathtt{tt} \to [\![P_1]\!]) \parallel U]$$

$$\lfloor \mathrm{IO1} \rfloor\ C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^1_{xy} \parallel U] \xrightarrow{\mathtt{IO}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel [\![P_2]\!]\{v/z\} \parallel U]$$

$$\lfloor \mathrm{RP1} \rfloor\ C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid * \, y(z).P_2|\!)^1_{xy} \parallel U] \xrightarrow{\mathsf{RP}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel [\![P_2]\!]\{v/z\} \parallel [\![* \, y(z).P_2]\!] \parallel U]$$

$$\lfloor \mathrm{SL1} \rfloor\ C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^1_{xy} \parallel U] \xrightarrow{\mathsf{SL}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_j.P_1 \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^2_{xy} \parallel U]$$

$$\lfloor \mathrm{SL2} \rfloor\ C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^1_{xy} \parallel U] \xrightarrow{\mathsf{SL}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_j.P_1 \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^3_{xy} \parallel U]$$

$$\lfloor \mathrm{SL3} \rfloor\ \frac{J = \prod_{i\in I\setminus\{j\}} \forall\epsilon(l_j = l_i \to \overline{\mathsf{bra}(y,l_j)} \parallel [\![P_i]\!])}{C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^2_{xy} \parallel U] \xrightarrow{\mathsf{SL}_2(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P]\!] \parallel [\![P_j]\!] \parallel J \parallel U]}$$

$$\lfloor \mathrm{SL4} \rfloor\ \frac{J = \prod_{i\in I\setminus\{j\}} \forall\epsilon(l_j = l_i \to \overline{\mathsf{bra}(y,l_j)} \parallel [\![P_i]\!])}{C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^3_{xy} \parallel U] \xrightarrow{\mathsf{SL}_3(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P]\!] \parallel [\![P_j]\!] \parallel J \parallel U]}$$

Figure 4.7: Labeled Transitions for $[\![\cdot]\!]$ (cf. Def. 4.37).

Example 4.39. Recall process P_{18} from Ex. 4.10:

$$P_{18} = (\boldsymbol{\nu} xy)(x \triangleleft \mathit{buy}.\, x\langle 5406\rangle.\, x(\mathit{inv}).\mathbf{0} \mid y \triangleright \{\mathit{buy} : y(w).y\langle \mathsf{invoice}\rangle.\mathbf{0}, \mathit{quit} : y(w').\mathbf{0}\})$$

Using the notation in Def. 4.37, it is possible to express the transitions of $[\![P_{18}]\!]$ as follows:

$$\begin{aligned}
[\![P_{18}]\!] \xrightarrow{\mathsf{SL}(x,y)}_1 \exists x, y.\big(\,& !\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{bra}(y, \mathit{buy})} \parallel \\
& \forall u_1(\mathsf{bra}(u_1, \mathit{buy}) \otimes \{x{:}u_1\} \to \overline{\mathsf{snd}(x, 5406)} \parallel \\
& \qquad \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(\mathit{inv}).\mathbf{0}]\!])) \parallel \\
& \forall\epsilon(\mathit{buy} = \mathit{buy} \to \forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \to \overline{\mathsf{rcv}(y, w)} \parallel \\
& \qquad [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!])) \parallel \\
& \forall\epsilon(\mathit{buy} = \mathit{quit} \to \forall w_2, w'(\mathsf{snd}(w_2, w') \otimes \{w{:}y\} \to \overline{\mathsf{rcv}(y, w')} \parallel \\
& \qquad [\![y(w').\mathbf{0}]\!])))
\end{aligned}$$

$$\begin{aligned}
\xrightarrow{\mathsf{SL}_1(x,y)}_1 \exists x, y.\big(\,& !\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x, 5406)} \parallel \\
& \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \to [\![x(\mathit{inv}).\mathbf{0}]\!]) \parallel \\
& \forall\epsilon(\mathit{buy} = \mathit{buy} \to \forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \to \overline{\mathsf{rcv}(y, w)} \parallel \\
& \qquad [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!])) \parallel \\
& \forall\epsilon(\mathit{buy} = \mathit{quit} \to \forall w_2, w'(\mathsf{snd}(w_2, w') \otimes \{w{:}y\} \to
\end{aligned}$$

$$\overline{\mathsf{rcv}(y, w')} \parallel [\![y(w').\mathbf{0}]\!])))$$

$$\xrightarrow{\mathtt{SL}_2(x,y)}_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x, 5406)} \parallel \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \rightarrow [\![x(inv).\mathbf{0}]\!]) \parallel$$
$$\forall w_1, w(\mathsf{snd}(w_1, w) \otimes \{w_1{:}y\} \rightarrow \overline{\mathsf{rcv}(y, w)} \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!]) \parallel$$
$$\forall \epsilon(buy = quit \rightarrow \forall w_2, w'(\mathsf{snd}(w_2, w') \otimes \{w{:}y\} \rightarrow \overline{\mathsf{rcv}(y, w')} \parallel [\![y(w').\mathbf{0}]\!])))$$

$$\xrightarrow{\mathtt{IO}(x,y)}_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel \forall u_2(\mathsf{rcv}(u_2, 5406) \otimes \{x{:}u_2\} \rightarrow [\![x(inv).\mathbf{0}]\!]) \parallel \overline{\mathsf{rcv}(y, w)} \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!] \parallel$$
$$\forall \epsilon(buy = quit \rightarrow \forall w_2, w'(\mathsf{snd}(w_2, w') \otimes \{w{:}y\} \rightarrow \overline{\mathsf{rcv}(y, w')} \parallel [\![y(w').\mathbf{0}]\!])))$$

$$\xrightarrow{\mathtt{IO}_1(x,y)}_1 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![x(inv).\mathbf{0}]\!] \parallel [\![y\langle \mathsf{invoice}\rangle.\mathbf{0}]\!] \parallel$$
$$\forall \epsilon(buy = quit \rightarrow \forall w_2, w'(\mathsf{snd}(w_2, w') \otimes \{w{:}y\} \rightarrow \overline{\mathsf{rcv}(y, w')} \parallel [\![y(w').\mathbf{0}]\!])))$$

△

The following lemma asserts that the labeled lcc transitions correspond with lcc τ-transitions. Thus, this result allows us to state that labeled transitions and τ-transitions are interchangeable.

Lemma 4.40. *Let S be a target term (cf. Def. 4.7) and x, y be endpoints. Then, $S \xrightarrow{\tau}_1 S'$ if and only if $S \xrightarrow{\alpha(x,y)}_1 S'$ where $\alpha \in \{\mathtt{IO}, \mathtt{SL}, \mathtt{RP}, \mathtt{CD}, \mathtt{IO}_1, \mathtt{RP}_1, \mathtt{SL}_1, \mathtt{SL}_2, \mathtt{SL}_3\}$.*

Proof. We prove both sides:

⇒) By Cor. 4.35, $S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n]$, with $U_i = [\![R_i]\!]$ for some pre-redex R_i (cf. Def. 2.19). We take then an arbitrary U_i, $i \in \{1, \ldots, n\}$. We apply a case analysis on U_i. There are 11 cases corresponding to each possible shape of U_i. We only show three cases; the rest are similar:

Case $U_i = [\![x\langle v\rangle.P_1]\!]$**:** We distinguish two sub-cases that depend on whether there exists U_j such that $U_j = [\![y(z).P_2]\!]$ and $x \in \widetilde{x}, y \in \widetilde{y}$ or not. The latter case is vacuously true, as there would not be any transition to check. We show the former case:

Sub-case $\exists U_j.(U_j = [\![y(z).P_2]\!] \wedge x \in \widetilde{x}, y \in \widetilde{y})$**:**

(1) $S \equiv C_{\widetilde{x}\widetilde{y}}[[\![x\langle v\rangle.P_1]\!] \parallel [\![y(z).P_2]\!] \parallel U_1 \parallel \cdots \parallel U_n]$ by Assumption.

(2) $S \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U_1 \parallel \cdots \parallel U_n] = S'$ by applying Rule (C:Sync) from Fig. 2.5, (1), and Assumption.

(3) $S \xrightarrow{\mathtt{IO}_1(x,y)}_1 S'$ by Def. 4.37, (1), and (2).

Case $U_i = [\![x(y).P_1]\!]$**:** Symmetric to the previous case, as $U_j = [\![y\langle v\rangle.P_2]\!]$.

Case $U_i = (\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^1_{\widetilde{x}\widetilde{y}}$**:**

(1) $S \equiv C_{\widetilde{xy}}[(\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^{1}_{\widetilde{xy}} \parallel U_1 \parallel \cdots \parallel U_n]$ by Assumption.

(2) $S \xrightarrow{\tau}_1 C_{\widetilde{xy}}[[\![P_1]\!] \parallel [\![P_2]\!]\{v/z\} \parallel U_1 \parallel \cdots \parallel U_n] = S'$ by applying Rule (C:Sync) from Fig. 2.5, (1), and Assumption.

(3) $S \xrightarrow{\mathtt{IO}_1(x,y)}_1 S'$ by Def. 4.37, (1), and (2).

$\Leftarrow$) This direction proceeds by applying a case analysis on label $\alpha(x,y)$. Each case then will proceed by applying Rule (C:Sync) in Fig. 2.5 and showing that the transition yields the correct process.

$\square$

We further categorize labels in *opening* and *closing* labels. Formally:

Definition 4.41 (Opening and Closing Labels). Consider the set of labels defined in Def. 4.37. We will say that $O = \{\mathtt{IO}, \mathtt{SL}, \mathtt{RP}, \mathtt{SL}_1\}$ is the set of *opening labels* and write ω to refer to its elements. Similarly, we will call $C = \{\mathtt{IO}_1, \mathtt{RP}_1, \mathtt{CD}, \mathtt{SL}_2, \mathtt{SL}_3\}$ the set of *closing labels* and write κ to refer to its elements.

Intuitively, a transition with an opening label always goes to an intermediate process, whereas a closing label leads to the translation of a $\pi_{\mathsf{OR}}^{\lightning}$ process. Notice that label CD does not have intermediate processes, but rather goes directly into the translation of the continuation; this is why it is considered a closing label itself. Similarly, label $\mathtt{SL}_1$ is opening because it reaches an intermediate process, rather than a translation. Now we introduce some notation for dealing with sequences of labels:

Notation 4.42.

- We will write $\gamma(\widetilde{xy})$ to denote finite sequences $\alpha_1(x_1, y_1), \ldots, \alpha_m(x_n, y_n)$, with $n, m \geq 1$.
- We will write $S \xRightarrow{\gamma(\widetilde{xy})}_1 S'$ only if there exist target terms $S_1, \ldots, S_m$ such that $S \xrightarrow{\alpha_1(x_1,y_1)}_1 S_1 \xrightarrow{\alpha_2(x_2,y_2)}_1 \cdots S_{m-1} \xrightarrow{\alpha_m(x_n,y_n)}_1 S_m = S'$ and $\gamma(\widetilde{xy}) = \alpha_1(x_1, y_2), \ldots, \alpha_m(x_n, y_n)$.
- Given $\gamma(\widetilde{xy})$, we will write $\alpha(x,y) \in \gamma(\widetilde{xy})$ to denote that $\alpha(x,y)$ is in the sequence $\gamma(\widetilde{xy})$.
- Given $\gamma(\widetilde{xy})$, we will write $\gamma(\widetilde{xy}) \setminus \alpha_i(x_j, y_j)$ to denote the sequence obtained from $\gamma(\widetilde{xy})$ by removing $\alpha_i(x_j, y_j)$.
- Given $\gamma(\widetilde{xy})$, we will say that $\gamma(\widetilde{xy})$ is an opening (resp. closing) sequence if every $\alpha \in \gamma(\widetilde{xy})$ is an opening (resp. closing) label (cf. Def. 4.41).

In general, we need to recognize when a synchronization starts and when it ends in the translation. The intuition is that an opening label starts a synchronization and a closing label ends it by reaching the translation of some $\pi_{\mathsf{OR}}^{\lightning}$ program. We capture these *complete synchronizations* in the following definition:

Definition 4.43 (Complete Synchronizations). Let S_0 be a target term such that $S_0 \xRightarrow{\gamma(\widetilde{xy})}_1 S_1$. Then we have:

1. If there exist $\gamma_1(\widetilde{xy})$ and $\gamma_2(\widetilde{xy})$ such that either:
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathtt{IO}(x,y)\gamma_2(\widetilde{xy})\mathtt{IO}_1(x,y)$ or
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathtt{IO}(x,y)\gamma_2(\widetilde{xy})\mathsf{RP}_1(x,y)$

   then we say that $\gamma(\widetilde{xy})$ is a *complete synchronization with respect to* $\mathtt{IO}(x,y)$.
2. If there exist $\gamma_1(\widetilde{xy})$ and $\gamma_2(\widetilde{xy})$ such that either:
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathsf{RP}(x,y)\gamma_2(\widetilde{xy})\mathsf{RP}_1(x,y)$
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathsf{RP}(x,y)\gamma_2(\widetilde{xy})\mathtt{IO}_1(x,y)$

   then we say that $\gamma(\widetilde{xy})$ is a *complete synchronization with respect to* $\mathsf{RP}(x,y)$.
3. If there exist $\gamma_1(\widetilde{xy})$, $\gamma_2(\widetilde{xy})$ and $\gamma_3(\widetilde{xy})$ such that either:
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathsf{SL}(x,y)\gamma_2(\widetilde{xy})\mathsf{SL}_1(x,y)\gamma_3(\widetilde{xy})\mathsf{SL}_2(x,y)$
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathsf{SL}(x,y)\gamma_2(\widetilde{xy})\mathsf{SL}_1(x,y)\gamma_3(\widetilde{xy})\mathsf{SL}_3(x,y)$

   then we say that $\gamma(\widetilde{xy})$ is a *complete synchronization with respect to* $\mathsf{SL}(x,y)$.
4. If there exist $\gamma_1(\widetilde{xy})$ and $\gamma_2(\widetilde{xy})$ such that either:
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathsf{SL}_1(x,y)\gamma_2(\widetilde{xy})\mathsf{SL}_2(x,y)$ or
   - $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathsf{SL}_1(x,y)\gamma_2(\widetilde{xy})\mathsf{SL}_3(x,y)$

   then we say that $\gamma(\widetilde{xy})$ is a *complete synchronization with respect to* $\mathsf{SL}_1(x,y)$.
5. If there exists $\gamma_1(\widetilde{xy})$ such that $\gamma(\widetilde{xy}) = \gamma_1(\widetilde{xy})\mathsf{CD}(-)$ then we say that $\gamma(\widetilde{xy})$ is a *complete synchronization* with respect to $\mathsf{CD}(-)$.

The case of the conditional redexes is 'unique' in our setting: it is the only one whose translation needs a single lcc step to reach the translation of its continuation. Therefore, we consider every conditional transition a complete synchronization. Some examples of complete synchronizations follow:

Example 4.44. Consider the following target terms:

$$\begin{aligned}
S_1 &= C_{\widetilde{xy}}[[\![x_1\langle v\rangle.P_1]\!] \parallel [\![y_1(z).y_2 \triangleright \{l_i : Q_i\}_{i\in I}]\!] \parallel [\![x_2 \triangleleft l.Q]\!]] \\
S_2 &= C_{\widetilde{xy}}[[\![x_1\langle v\rangle.P_1]\!] \parallel [\![y_1(z).P_2]\!] \parallel [\![y_1(z').P_3]\!]] \\
S_3 &= C_{xy}[(\!|x \triangleleft l_j.P_1 \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^1_{xy}] \\
S_4 &= C_{\widetilde{xy}}[[\![x_1\langle v\rangle.P_1]\!] \parallel [\![* \, y_1(z).P_2]\!]]
\end{aligned}$$

The following transitions are complete synchronizations with respect to the first label in the sequence for processes S_1, S_2, and S_3:

$$\begin{aligned}
S_1 &\xrightarrow{\mathtt{IO}(xy)}_1 C_{\widetilde{xy}}[(\!|x_1\langle v\rangle.P_1 \mid y_1(z).x_2 \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{xy} \parallel [\![x_2 \triangleleft l.Q]\!]] \\
&\xrightarrow{\mathtt{IO}_1(xy)}_1 C_{\widetilde{xy}}[[\![P_1]\!] \parallel [\![x_2 \triangleright \{l_i : Q_i\}_{i\in I}]\!]\{^v\!/z\} \parallel [\![x_2 \triangleleft l.Q]\!]] \\
S_2 &\xrightarrow{\mathtt{IO}(xy)}_1 C_{\widetilde{xy}}[(\!|x_1\langle v\rangle.P_1 \mid y_1(z).P_2|\!)^1_{xy} \parallel [\![y_1(z').P_3]\!]]
\end{aligned}$$

$$
\begin{aligned}
&\xrightarrow{\mathtt{IO}_1(xy)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel [\![P_2]\!]\{^{v}\!/z\} \parallel [\![y_1(z').P_3]\!]] \\
S_3 &\xrightarrow{\mathsf{SL}_1(xy)}_1 C_{xy}[(\!|x \triangleleft l_j.P_1 \mid y \triangleright \{l_i : P_i\}_{i\in I}|\!)^3_{xy}] \\
&\xrightarrow{\mathsf{SL}_3(xy)}_1 C_{xy}[[\![P_1]\!] \parallel [\![Q_j]\!]]
\end{aligned}
$$

Process S_4 allows us to explain an interesting observation about the labeled semantics introduced for our translation. In particular, consider that:

$$S_4 \equiv C_{\widetilde{x}\widetilde{y}}[[\![x_1\langle v\rangle.P_1]\!] \parallel [\![y_1(z).P_2]\!] \parallel [\![* \, y_1(z).P_2]\!]]$$

which means that there are two possible labeled transitions for S_4:

$$
\begin{aligned}
S_4 &\xrightarrow{\mathsf{RP}(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x_1\langle v\rangle.P_1 \mid * \, y_1(z).P_2|\!)^1_{xy}] = S_4' \\
S_4 &\xrightarrow{\mathtt{IO}(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x_1\langle v\rangle.P_1 \mid y_1(z).P_2|\!)^1_{xy} \parallel [\![* \, y_1(z).P_2]\!]] = S_4''
\end{aligned}
$$

Then it is possible to show that $S_4' \equiv S_4''$ and that each process can complete the synchronization by taking either an $\mathtt{IO}_1(x,y)$ label or an $\mathsf{RP}_1(x,y)$ label, reaching the same process:

$$
\begin{aligned}
S_4' &\xrightarrow{\mathtt{IO}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel [\![P_2]\!]\{^{v}\!/z\} \parallel [\![* \, y_1(z).P_2]\!]] \\
S_4' &\xrightarrow{\mathsf{RP}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel [\![P_2]\!]\{^{v}\!/z\} \parallel [\![* \, y_1(z).P_2]\!]] \\
S_4'' &\xrightarrow{\mathtt{IO}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel [\![P_2]\!]\{^{v}\!/z\} \parallel [\![* \, y_1(z).P_2]\!]] \\
S_4'' &\xrightarrow{\mathsf{RP}_1(x,y)}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1]\!] \parallel [\![P_2]\!]\{^{v}\!/z\} \parallel [\![* \, y_1(z).P_2]\!]]
\end{aligned}
$$

△

Using complete synchronizations, we can then describe the *open labels* of a sequence of transitions:

Definition 4.45 (Open Labels of a Sequence of Transitions). Let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program such that $[\![P]\!] = S_0 \xRightarrow{\gamma(\widetilde{x}\widetilde{y})}_1 S_n$, with $n = |\gamma(\widetilde{x}\widetilde{y})|$. We define the *open labels* of $\gamma(\widetilde{x}\widetilde{y})$, written $open(\gamma(\widetilde{x}\widetilde{y}))$, as the longest sequence $\beta_1 \ldots \beta_m$ (with $m \leq n$) that preserves the order in $\gamma(\widetilde{x}\widetilde{y})$ and such that for every β_i (with $1 \leq i \leq m$):

(1) $\beta_i = \alpha_j$, for some opening label $\alpha_j \in \gamma(\widetilde{x}\widetilde{y})$;

(2) there is not a subsequence $\gamma(\widetilde{x}\widetilde{y})$ that is a complete synchronization with respect to β_i (cf. Def. 4.43).

Then, we define the *complementary execution sequence* of an opening label, which intuitively provides the missing transition labels such that the transition produces a complete synchronization.

Definition 4.46 (Complementary Execution Sequence). Let ω be any opening label. We say that the *complementary execution sequence* of ω, written $\omega\downarrow$, is defined as follows:

$$\mathtt{IO}(xy)\downarrow = \mathtt{IO}_1(xy) \qquad\qquad \mathsf{RP}(xy)\downarrow = \mathsf{RP}_1(xy)$$

$$\begin{tikzcd}
 & \llbracket P \rrbracket \arrow[r, Rightarrow, "\gamma_0(\widetilde{xy})", "\text{Assump.}"'] \arrow[dl, "\text{IH}"', "*"] & S_0 \arrow[r, "\alpha_{n+1}", "\text{Assump.}"'] \arrow[d, Rightarrow, "\text{IH}"', "\gamma_0(\widetilde{xy})\downarrow"] & S \arrow[d, Rightarrow, dotted, "\gamma_0(\widetilde{xy})\downarrow"] & \\
Q_0 \arrow[r, dash, dashed, "\llbracket \cdot \rrbracket"] \arrow[drrrr, dotted, "*"] & \llbracket Q_0 \rrbracket \arrow[r, dash, dashed, "\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}", "\text{IH}"'] & S_0' \arrow[r, dotted, "\alpha_{n+1}"] & S_0'' \arrow[r, dotted, "\alpha_{n+1}\downarrow"] & S' \arrow[d, dotted, "\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}"] \\
 & & & & Q
\end{tikzcd}$$

Figure 4.8: Diagram of the proof of Lem. 4.48. The dotted arrows represent the reductions and equivalences that must be proven.

$$\mathsf{SL}(xy)\downarrow = \mathsf{SL}_1(xy)\mathsf{SL}_2(xy) \qquad\qquad \mathsf{SL}_1(xy)\downarrow = \mathsf{SL}_2(xy)$$

Furthermore, let $S_1 \xRightarrow{\gamma(\widetilde{xy})}_1 S_2$ be transition sequence such that $open(\gamma(\widetilde{xy})) = \omega_1 \ldots \omega_n$, with $n \geq 1$. We define $\gamma(\widetilde{xy})\downarrow$ as $\omega_1\downarrow \ldots \omega_n\downarrow$.

The following lemma provides a diamond property for opening and closing transitions. It states that closing actions do not interfere with opening transitions.

Lemma 4.47. *Let S be a target term such that $S \xrightarrow{\omega}_1 S_1$ and $S \xRightarrow{\gamma(\widetilde{xy})}_1 S_2$, where $\gamma(\widetilde{xy})$ is a closing sequence (cf. Not. 4.42). Then, there exists S_3 such that $S_1 \xRightarrow{\gamma(\widetilde{xy})}_1 S_3$ and $S_2 \xrightarrow{\omega}_1 S_3$.*

Proof. By induction on the length n of $|\gamma(\widetilde{xy})|$. For details see App. B.5. □

The next lemma will show that every target term obtained from a translated program can reach the translation of a $\pi_{\mathsf{OR}}^{\lightning}$ program just by closing the remaining open communications in the target term.

Lemma 4.48. *For every well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program P and for every sequence of labels $\gamma(\widetilde{xy})$ such that $\llbracket P \rrbracket \xRightarrow{\gamma(\widetilde{xy})}_1 S$, there exist Q, S', and $\gamma'(\widetilde{xy})$ such that $P \longrightarrow^* Q$ and $S \xRightarrow{\gamma'(\widetilde{xy})}_1 S'$, with $\gamma'(\widetilde{xy}) = \gamma(\widetilde{xy})\downarrow$ (cf. Def. 4.46). Moreover, $\llbracket Q \rrbracket \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} S'$.*

Proof. By induction on $|\gamma(\widetilde{xy})|$ and a case analysis on the last label of the sequence. The base case is immediate since $\llbracket P \rrbracket \xRightarrow{\gamma(\widetilde{xy})}_1 \llbracket P \rrbracket$ and $P \longrightarrow^* P$. The proof for the inductive hypothesis can be seen in Fig. 4.8. The dotted arrows are the reductions that must be proven to exist. For details see App. B.5. □

Proof of Operational Soundness

Having detailed all the ingredients required in the proof of operational soundness, we repeat the statement of Thm. 4.21 (given in Page 133) and detail its proof, which formalizes the sketch discussed in § 4.3.3:

Theorem 4.21 (Soundness for $[\![\cdot]\!]$). *Let $[\![\cdot]\!]$ be the translation in Def. 4.4. Also, let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program. For every S such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S$ there are Q, S' such that $P \longrightarrow^* Q$ and $S \overset{\tau}{\Longrightarrow}_1 S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$.*

Proof. By induction on k, the length of the reduction $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S$, followed by a case analysis on the constraints that may have been consumed in the very last reduction.

Base Case: Then $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 [\![P]\!]$. The thesis follows from the reflexivity of $\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}$, since $[\![P]\!] \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P]\!]$.

Inductive Step: Assume $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_0 \longrightarrow_1 S$ (with $k-1$ steps between $[\![P]\!]$ and S_0). By IH, there exist Q_0 and S_0' such that $P \longrightarrow^* Q_0$ and $S_0 \overset{\tau}{\Longrightarrow}_1 S_0' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q_0]\!]$. Observe that by combining the IH and Lem. 4.48, we have that the sequence $S_0 \overset{\tau}{\Longrightarrow}_1 S_0'$ contains only closing labels. We must prove that there exist Q and S' such that $P \longrightarrow^* Q$ and $S \overset{\tau}{\Longrightarrow}_1 S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$. We analyze $\mathcal{I}_{S_0}$ and $\mathcal{I}_S$ (cf. Def. 4.22) according to two cases: $\mathcal{I}_{S_0} \subseteq \mathcal{I}_S$ and $\mathcal{I}_{S_0} \not\subseteq \mathcal{I}_S$, which use Lem. 4.33 and Lem. 4.34, respectively:

Case $\mathcal{I}_{S_0} \subseteq \mathcal{I}_S$: By Lem. 4.33 there are two sub-cases depending on the shape of S_0:

Sub-case 1: $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[[\![b?\,(Q_1) : (Q_2)]\!] \parallel U]$, $b \in \{\mathtt{tt}, \mathtt{ff}\}$, for some U. By Lem. 4.33 and inspection on the translation definition (Fig. 4.2), it must be the case that $Q_0 \equiv_{\mathsf{S}} (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(b?\,(Q_1) : (Q_2) \mid R)$, for some R. We distinguish two sub-subcases, depending on b:

Sub-subcase $b = \mathtt{tt}$: We proceed as follows:

(1) $S_0 \overset{\tau}{\Longrightarrow}_1 S_0' \equiv C_{\widetilde{x}\widetilde{y}}[[\![\mathtt{tt}?\,(Q_1) : (Q_2)]\!] \parallel U']$, for some U' By IH.

(2) $S_0' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q_0]\!] = C_{\widetilde{x}\widetilde{y}}[[\![\mathtt{tt}?\,(Q_1) : (Q_2)]\!] \parallel [\![R]\!]]$ by IH.

(3) $S_0 \longrightarrow_1 C_{\widetilde{x}\widetilde{y}}[[\![Q_1]\!] \parallel \forall\epsilon(\mathtt{tt} = \mathtt{ff} \rightarrow [\![Q_2]\!]) \parallel U] = S$ by Lem. 4.33.

(4) $S_0 \longrightarrow_1 S \overset{\tau}{\Longrightarrow}_1 C_{\widetilde{x}\widetilde{y}}[[\![Q_1]\!] \parallel \forall\epsilon(\mathtt{tt} = \mathtt{ff} \rightarrow [\![Q_2]\!]) \parallel U'] = S'$ by (3), (1).

(5) $S_0' \longrightarrow_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}\widetilde{y}}[[\![Q_1]\!] \parallel U'] \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} S'$ by Fig. 2.5, Cor. 4.35, and (4).

(6) $[\![Q_0]\!] \longrightarrow_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}\widetilde{y}}[[\![Q_1]\!] \parallel [\![R]\!]] = W$ by (2), Fig. 2.5, and applying Cor. 4.35.

(7) $S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} W = C_{\widetilde{x}\widetilde{y}}[[\![Q_1]\!] \parallel [\![R]\!]]$ by (2),(5), and Lem. 4.47 with $S = S_0, S_1 = S, S_2 = [\![Q_0]\!]$.

(8) $Q_0 \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(Q_1 \mid R) = Q$ by applying Rule $\lfloor \textsc{IfT} \rfloor$ from Fig. 2.1

(9) $[\![Q]\!] = C_{\widetilde{x}\widetilde{y}}[[\![Q_1]\!] \parallel [\![R]\!]] = W$ by Fig. 4.2, (8), and (6).

(10) $S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$ by (7), and (9).

Sub-subcase $b = \mathtt{ff}$**:** This case is analogous to the one above.

Sub-case 2: By Lem. 4.33, there is a W such that $W \longrightarrow_1^h S_0$ with $h \in \{1, 2\}$, where:

$$W = C_{\widetilde{x}\widetilde{y}}[[\![x \triangleleft l_k.Q' \mid x \triangleright \{l_h : Q_h\}_{h\in I}]\!] \parallel U]$$

for some U. We distinguish cases according to h:

Sub-subcase $h = 2$**:**

(1) $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_k.Q' \mid x \triangleright \{l_h : Q_h\}_{h\in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel U]$ by Lem. 4.33.

(2) $Q_0 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(Q' \mid Q_k \mid R)$ by (1) and IH.

(3) $S_0 \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}\widetilde{y}}[[\![Q' \mid Q_k]\!] \parallel U'] = S_0'$ by IH, Fig. 4.2, and Cor. 4.35.

(4) $S_0' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q_0]\!] = C_{\widetilde{x}\widetilde{y}}[[\![Q']\!] \parallel [\![Q_k]\!] \parallel [\![R]\!]]$ by IH, (3) and (2).

(5) $S_0 \longrightarrow_1 C_{\widetilde{x}\widetilde{y}}[[\![Q' \mid Q_k]\!] \parallel \prod_{h\in I\setminus\{k\}} \forall\epsilon(l_k = l_h \rightarrow [\![Q_h]\!]) \parallel U] = S$ by Fig. 2.5 and (1).

(6) $S_0 \longrightarrow_1 S \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} S_0' = S'$ by (5), (3), and Lem. 4.47 with $S = S_0, S_1 = S, S_2 = [\![Q_0]\!]$.

(7) $Q_0 \longrightarrow^* Q_0 = Q$ by Fig. 2.1.

(8) $S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$ by (6), (7), and (4).

Sub-subcase $h = 1$**:**

(1) $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_k.Q' \mid x \triangleright \{l_h : Q_h\}_{h\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$ by Lem. 4.33.

(2) $Q_0 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(Q' \mid Q_k \mid R)$ by (1) and Assumption.

(3) $S_0 \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}\widetilde{y}}[[\![Q' \mid Q_k]\!] \parallel U'] = S_0'$ by IH, Fig. 4.2, and Cor. 4.35.

(4) $S_0' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q_0]\!] = C_{\widetilde{x}\widetilde{y}}[[\![Q']\!] \parallel [\![Q_k]\!] \parallel [\![R]\!]]$ by IH, (3), and (2).

(5) $S_0 \longrightarrow_1 C_{\widetilde{x}\widetilde{y}}[(\!|x \triangleleft l_k.Q' \mid x \triangleright \{l_h : Q_h\}_{h\in I}|\!)^3_{\widetilde{x}\widetilde{y}} \parallel U] = S$ by Fig. 2.5.

(6) $S \longrightarrow_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} C_{\widetilde{x}\widetilde{y}}[[\![Q' \mid Q_k]\!] \parallel U] \overset{\tau}{\Longrightarrow}_1 C_{\widetilde{x}\widetilde{y}}[[\![Q' \mid Q_k]\!] \parallel U'] = S'$ by Fig. 2.5, Cor. 4.35, and (3).

(7) $S_0 \longrightarrow_1 S \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} S_0' = S'$ by (5), (6), and Lem. 4.47, (3) with $S = S_0, S_1 = S, S_2 = [\![Q_0]\!]$.

(8) $Q_0 \longrightarrow^* Q_0 = Q$ by Fig. 2.1.

(9) $S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$ by (7), (8), and (4).

Case $\mathcal{I}_{S_0} \not\subseteq \mathcal{I}_S$**:** By Lem. 4.34 we distinguish sub-cases depending on the constraints in $\mathcal{I}_{S_0} \setminus \mathcal{I}_S$. By Prop. 4.32, constraints are unique and therefore, $\mathcal{I}_{S_0} \setminus \mathcal{I}_S$ correctly accounts for the specific consumed constraint. There are four cases, as indicated by Lem. 4.34:

Sub-case $\mathsf{snd}(x, v) \in \mathcal{I}_{S_0} \setminus \mathcal{I}_S$**:** By Lem. 4.34 we have, for some U:

(a) $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![x\langle v\rangle.Q_1 \mid \diamond\ y(z).Q_2]\!] \parallel U]$.

(b) $S \equiv C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.Q_1 \mid \diamond\ y(z).Q_2|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$.

We distinguish cases depending on $\diamond\, y(z).Q_2$ (cf. Not. 2.24):

Sub-subcase $\diamond\, y(z).Q_2 = y(z).Q_2$**:** We proceed as follows:

(1) $S_0 \overset{\tau}{\Longrightarrow}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![x\langle v\rangle.Q_1 \mid y(z).Q_2]\!] \parallel U'] = S_0'$ by IH.

(2) $S_0' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(x\langle v\rangle.Q_1 \mid y(z).Q_2 \mid R)]\!] = [\![Q_0]\!]$ by IH.

(3) $S_0 \longrightarrow_1 S \longrightarrow_1 C_{\widetilde{x}\widetilde{y}}[[\![Q_1 \mid Q_2\{v/z\}]\!] \parallel U]$ by (a), (b), and Fig. 2.5.

(4) $S \overset{\tau}{\Longrightarrow}_1 C_{\widetilde{x}\widetilde{y}}[[\![Q_1 \mid Q_2\{v/z\}]\!] \parallel U'] = S'$ by (3) and (1).

(5) $S_0' \longrightarrow_1^2 C_{\widetilde{x}\widetilde{y}}[[\![Q_1 \mid Q_2\{v/z\}]\!] \parallel U'] = S'$ by (1), Fig. 2.5, and (4).

(6) $[\![Q_0]\!] \longrightarrow_1^2 [\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(Q_1 \mid Q_2\{v/z\} \mid R)]\!] = W$ by (2) and Fig. 2.5.

(7) $S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} W$ by (2), (5), and Lem. 4.47 with $S = S_0, S_1 = S, S_2 = [\![Q_0]\!]$.

(8) $Q_0 \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(Q_1 \mid Q_2\{v/z\} \mid R) = Q$ by applying Rule $\lfloor\textsc{Com}\rfloor$ from Fig. 2.1.

(9) $W = [\![Q]\!]$ by (8) and (6).

(10) $S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$ by (7) and (9).

Sub-subcase $\diamond\, y(z).Q_2 = *\, y(z).Q_2$**:** Similar to the case above, using Rule $\lfloor\textsc{Repl}\rfloor$ instead of Rule $\lfloor\textsc{Com}\rfloor$.

Sub-case $\mathsf{rcv}(x, v) \in \mathcal{I}_{S_0} \setminus \mathcal{I}_S$**:** By Lem. 4.34, there exists:

$$W = C_{\widetilde{x}\widetilde{y}}[[\![x\langle v\rangle.Q_1 \mid \diamond\ y(z).Q_2]\!] \parallel U]$$

such that $W \longrightarrow_1 S_0$. We distinguish cases depending on $\diamond\, y(z).Q_2$ (cf. Not. 2.24):

Sub-subcase $\diamond\, y(z).Q_2 = y(z).Q_2$**:** We proceed as follows:

(1) $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.Q_1 \mid \diamond\ y(z).Q_2|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$ by Lem. 4.34.

(2) $Q_0 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(Q_1 \mid Q_2\{v/z\} \mid R)$ by (1).

(3) $S_0 \overset{\tau}{\Longrightarrow}_1 C_{\widetilde{x}\widetilde{y}}[[\![Q_1 \mid Q_2\{v/z\} \mid R]\!] \parallel U'] = S_0'$ by (1), Fig. 2.5, and IH.

(4) $S_0' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q_0]\!]$ by IH.

(5) $S_0 \longrightarrow_1 C_{\widetilde{x}\widetilde{y}}[[\![Q_1 \mid Q_2\{v/z\} \mid R]\!] \parallel U] = S$ by Fig. 2.5.

(6) $S_0 \longrightarrow_1 S \overset{\tau}{\Longrightarrow}_1 S_0' = S'$ by (5), (3), and Lem. 4.47 with $S = S_0$, $S_1 = S$, $S_2 = [\![Q_0]\!]$.

(7) $Q_0 \longrightarrow^* Q_0 = Q$ by Fig. 2.1.

(8) $S' \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$ (6), (7), and (4).

Sub-subcase $\diamond\, y(z).Q_2 = *\, y(z).Q_2$**:** Similar as above.

Sub-case $\mathsf{sel}(x, l) \in \mathcal{I}_{S_0} \setminus \mathcal{I}_S$**:** As above.

Sub-case $\mathsf{bra}(x, l) \in \mathcal{I}_{S_0} \setminus \mathcal{I}_S$**:** As above.

□

We have proven that the translation is name invariant (cf. Thm. 4.8), compositional (cf. Thm. 4.9), operationally complete (cf. Thm. 4.20), and operationally sound (Thm. 4.21). Considering this, we may now state that $[\![\cdot]\!]$ is a valid encoding, according to Def. 2.3.

Corollary 4.49. *The translation* $\langle [\![\cdot]\!], \varphi_{[\![\cdot]\!]} \rangle$ *(cf. Def. 4.4) is a valid encoding (cf. Def. 2.3).*

Proof. Direct consequence of Thm. 4.8, Thm. 4.9, Thm. 4.20 and Thm. 4.21. □

4.4 Timed Patterns Revisited: $[\![\cdot]\!]$

In this section we recall the timed patterns introduced in § 1.6 and show that we can represent the *request-response timeout* pattern, the *messages in a time-frame* pattern, and the *action duration* pattern by using the encoding $[\![\cdot]\!]$ (cf. Fig. 4.2). In § 4.4.1 we give an overview of the ingredients needed to represent these patterns using the encoding. In § 4.4.2, § 4.4.3, and § 4.4.4 we present the request-response timeout pattern, the messages in a time-frame pattern, and the action duration pattern. Finally, in § 4.4.5, we conjecture that the repeated constraint pattern (cf. § 1.6) is not representable using the encoding.

4.4.1 Overview: Exploiting Compositionality via Decompositions

Our goal is to use the encoding $[\![\cdot]\!]$ to show that `lcc` can be used as a foundation for the unified view we advocate. As shown before, the encoding satisfies correctness properties that ensure that source specifications in $\pi_{\mathsf{OR}}^{\lightning}$ can be represented in `lcc` and that their behavior is preserved (cf. Cor. 4.49). In this section we are interested in using the encoding $[\![\cdot]\!]$ to represent requirements that may appear in message-passing components but are not explicitly representable in $\pi_{\mathsf{OR}}^{\lightning}$. An example of such requirements is: *"an acknowledgment message* `ACK` *should be sent (by the server) no later than three time units after receiving the request message* `REQ`*".* As mentioned above, this kind of behavior is not straightforwardly representable in $\pi_{\mathsf{OR}}^{\lightning}$. In fact, using $\pi_{\mathsf{OR}}^{\lightning}$ we could only represent the interaction in the previous requirement as a *request-acknowledgment handshake*:

$$P_h = (\boldsymbol{\nu} xy)(x\langle \mathsf{REQ}\rangle.x(z).\mathbf{0} \mid y(z').y\langle \mathsf{ACK}\rangle.\mathbf{0}) \tag{4.3}$$

In the process, x represents the channel endpoint used by the client, while y represents the channel endpoint of the server. The client sends `REQ` and then awaits for `ACK`, which is sent by the server.

To represent requirements such as the one mentioned above using $[\![\cdot]\!]$, the key idea is to consider encoded terms as "code snippets" which can be used inside a larger `lcc` specification. Then, because of the correctness properties of the encoding, these snippets represent processes that will execute correct communication behavior. For example, we could write the following `lcc` process using $[\![P_h]\!]$:

$$Q = \forall \epsilon\big(\mathit{month}(m) = \mathit{january} \rightarrow [\![P_h]\!]\big)$$

Process Q checks that the current month (stored in some variable m) is *january* and only when this constraint is true it executes the behavior in $[\![P_h]\!]$. The operational correspondence property of the encoding ensures that the behavior of Q corresponds to that of the source process P_h, provided that the constraint is satisfied (cf. Thm. 4.20 and Thm. 4.21). This idea is useful to represent communication behavior that depends on *contextual information*, i.e., information external to the program.

Notice that Q uses the encoding of the complete process P_h. Still, in some cases it is desirable to represent constraints over the prefixes of P_h, rather than the complete $\pi_{\mathsf{OR}}^{\lightning}$ process itself. For example, to represent a timed requirement we could think of a $\pi_{\mathsf{OR}}^{\lightning}$ extension that allows to write:

$$P_h' = (\boldsymbol{\nu}xy)(x\langle \mathsf{REQ}\rangle.x^3(z).\mathbf{0} \mid y(z').y\langle \mathsf{ACK}\rangle.\mathbf{0}) \tag{4.4}$$

where $x^3(z).\mathbf{0}$ intuitively says that the input action should take place within three time units of the output action. This establishes a *timed requirement* connecting two different communication actions.

To enable the specification of processes such as P_h' using our encoding, it is convenient to consider $\pi_{\mathsf{OR}}^{\lightning}$ processes that have been *decomposed* in such a way that every action (i.e., prefix) appears in an independent parallel process. Such a decomposition should preserve the order in which actions are executed (as dictated by session types). In this way, the compositionality property of the encoding gives us control over specific parts of the translated $\pi_{\mathsf{OR}}^{\lightning}$ process, allowing us to insert constraints on the translated prefixes (cf. Thm. 4.9).

Such decompositions have been studied as *trios processes* and *minimal session types* [Par00, APV19]. The idea is to transform a π-calculus process P into a different (but equivalent) process formed only by *trios*, i.e., sequential processes with at most three prefixes (i.e., $\alpha.\beta.\gamma.\mathbf{0}$). Each trio is in charge of emulating a specific prefix of P; these trios must interact in such a way that every sub-term is executed at the right time, i.e., the *causal order* of interactions in P is preserved. Building upon this idea, the work on minimal session types considers processes that are typable with session types that describe a single action followed by end.

We do not intend to study in depth decompositions for $\pi_{\mathsf{OR}}^{\lightning}$; rather, we shall use these ideas to generate processes in which each prefix is represented by a parallel subprocess, and where the causal order of the initial process is preserved. Then, using the compositionality guarantees of Thm. 4.9, we obtain an appropriate granularity for the analysis of code snippets (obtained from trios) in `lcc` specifications.

For the sake of illustration, let us consider the decomposition of well-typed programs from the finite fragment of $\pi_{\mathsf{OR}}^{\lightning}$ without selection and branching (i.e., output, input, restriction, parallel composition, and inaction). This decomposition is based on the work presented in [Par00, APV19]. Since the idea of trios processes requires polyadic communication, we shall assume the following shorthands:

$$\begin{aligned} x\langle \widetilde{v}\rangle.P &= x\langle v_1\rangle.\dots.x\langle v_n\rangle.P \quad (|\widetilde{v}| = n \land n \geq 1)\\ x(\widetilde{y}).P &= x(y_1).\dots.x(y_n).P \quad (|\widetilde{y}| = n \land n \geq 1)\end{aligned}$$

We also use functions $\mathit{size}(\cdot)$ and $\mathsf{size}(\cdot)$ that return the size of process P and type T, respectively (cf. Fig. 4.9).

Remark 4.50. We decompose only well-typed programs (cf. Not. 2.21). Thm. 3.15 ensures that for every program P to be decomposed, $P = (\boldsymbol{\nu}x_1y_1)\dots(\boldsymbol{\nu}x_ny_n)Q$, with $n \geq 0$. Moreover, typability ensures there exists a context $\Gamma = \{x_1 : T_1, y_1 : \overline{T_1}, \dots, x_n : T_n, y_n : \overline{T_n}\}$ such that $\Gamma \vdash Q$. Finally, for simplicity we shall also assume no restriction $(\boldsymbol{\nu}x'y')$ occurs in Q.

Using the auxiliary functions in Fig. 4.9, we define the following decomposition function:

$$\mathit{size}(P) = \begin{cases} 1 + \mathit{size}(Q) & \text{if } P = x\langle v\rangle.Q \text{ or } P = x(y).Q \\ 1 + \mathit{size}(Q_1) + \mathit{size}(Q_2) & \text{if } P = Q_1 \mid Q_2 \\ \mathit{size}(Q) & \text{if } P = (\boldsymbol{\nu} xy)Q \\ 1 & \text{if } P = \mathbf{0} \end{cases}$$

$$\mathsf{size}(T) = \begin{cases} 1 + \mathsf{size}(T') & \text{if } T = q!T.T' \text{ or } T = q?T.T' \\ 1 & \text{if } T = \mathtt{bool} \\ 0 & \text{if } T = \mathsf{end} \end{cases}$$

Figure 4.9: Size of a process (resp. type) in the finite fragment of $\pi^{\sharp}_{\mathsf{OR}}$.

P	$\mathfrak{B}^{k}_{\widetilde{u}}(P)$	Side Conditions
$x_{i,j}\langle v\rangle.Q$	$c_k(\widetilde{u_1}x_{i,j}\widetilde{u_2}).x_{i,j}\langle\widetilde{v}\rangle.d_{k+1}\langle\widetilde{u_1}\widetilde{u_2}\rangle.\mathbf{0} \mid$ $\mathfrak{B}^{k+1}_{\widetilde{u_1}\widetilde{u_2}\setminus\widetilde{v}}(Q\{^{x_{i,j+1}}/_{x_{i,j}}\})$	$\widetilde{u} = \widetilde{u_1}x_{i,j}\widetilde{u_2}$ $i \in \{1,\dots,n\}$ $j \in \{1,\dots,m\}$
$x_{i,j}(y).Q$	$c_k(\widetilde{u_1}x_{i,j}\widetilde{u_2}).x_{i,j}(y\widetilde{z}).d_{k+1}\langle\widetilde{u_1}\widetilde{u_2}y\widetilde{z}\rangle.\mathbf{0} \mid$ $\mathfrak{B}^{k+1}_{\widetilde{u_1}\widetilde{u_2}y\widetilde{z}}(Q\{^{x_{i,j+1}}/_{x_{i,j}}\})$	$\widetilde{u} = \widetilde{u_1}x_{i,j}\widetilde{u_2}$ $i \in \{1,\dots,n\}$ $j \in \{1,\dots,m\}$
$P \mid Q$	$c_k(\widetilde{u_1}\widetilde{u_2}).d_{k+1}\langle\widetilde{u_1}\rangle.d_{l+1}\langle\widetilde{u_2}\rangle.\mathbf{0} \mid$ $\mathfrak{B}^{k+1}_{\widetilde{u_1}}(Q_1) \mid \mathfrak{B}^{l+1}_{\widetilde{u_2}}(Q_2)$	$l = k + \mathit{size}(Q_1)$ $\mathsf{fv}_{\pi}(Q_1) \in \widetilde{u_1}$ $\mathsf{fv}_{\pi}(Q_2) \in \widetilde{u_2}$
$\mathbf{0}$	$c_k(\widetilde{u}).\mathbf{0}$	

Figure 4.10: Breakdown function for the processes in the finite fragment of $\pi^{\sharp}_{\mathsf{OR}}$.

Definition 4.51 (Decomposition). Let $P = (\boldsymbol{\nu} x_1y_1)\dots(\boldsymbol{\nu} x_ny_n)Q$ be a well-typed program in the finite fragment of $\pi^{\sharp}_{\mathsf{OR}}$ and $\Gamma = \{x_1 : T_1, y_1 : \overline{T_1}, \dots, x_n : T_n, y_n : \overline{T_n}\}$ be a context such that $\Gamma \vdash Q$. The decomposition of P, denoted $\mathfrak{D}(P)$, is defined as

$$\mathfrak{D}(P) = (\boldsymbol{\nu}\widetilde{cd})(\boldsymbol{\nu}\widetilde{u})(d_1\langle\widetilde{u}\rangle.\mathbf{0} \mid \mathfrak{B}^{1}_{\widetilde{u}}(Q\{\widetilde{^{w'}}/_{\widetilde{w}}\}))$$

where:

(1) $\widetilde{u} = \widetilde{x_1}\widetilde{y_1}\dots\widetilde{x_n}\widetilde{y_n}$, $\widetilde{w} = x_1y_1\dots x_ny_n$, and $\widetilde{w'} = x_{1,1}y_{1,1}\dots x_{n,1}y_{n,1}$.

(2) $\widetilde{x_i} = x_{i,1}\dots x_{i,m}$ and $\widetilde{y_i} = y_{i,1}\dots y_{i,m}$ with $m = \mathsf{size}(T_i)$ for every $i \in \{1,\dots,m\}$.

(3) $\widetilde{c} = c_1\dots c_r$ and $\widetilde{d} = d_1\dots d_r$ with $r = \mathit{size}(P)$.

(4) $\mathfrak{B}^{k}_{\widetilde{u}}(P)$ is defined inductively over the finite fragment of $\pi^{\sharp}_{\mathsf{OR}}$ as in Fig. 4.10.

Let us now analyze the decomposition of P_h obtained from Def. 4.51. We shall use this decomposition to represent the timed behavior required by P'_h. Let $\widetilde{u} =$

$x_{1,1}x_{1,2}y_{1,1}y_{1,2}$, $\widetilde{u_x} = x_{1,1}x_{1,2}$, and $\widetilde{u_y} = y_{1,1}y_{1,2}$.

$$\begin{aligned}\mathfrak{D}(P_h) = {} & (\boldsymbol{\nu}c_1d_1)(\boldsymbol{\nu}c_2d_2)(\boldsymbol{\nu}c_3d_3)(\boldsymbol{\nu}c_4d_4)(\boldsymbol{\nu}c_5d_5)\\ & \quad (\boldsymbol{\nu}c_6d_6)(\boldsymbol{\nu}c_7d_7)(\boldsymbol{\nu}x_{1,1}y_{1,1})(\boldsymbol{\nu}x_{1,2}y_{1,2})\\ & \quad\quad (d_1\langle\widetilde{u}\rangle.\mathbf{0} \mid c_2(\widetilde{u_x}).x_{1,1}\langle\mathsf{REQ}\rangle.d_3\langle x_{1,2}\rangle.\mathbf{0} \mid\\ & \quad\quad c_3(x_{1,2}).x_{1,2}(z).d_4\langle z\rangle.\mathbf{0} \mid c_4(w).\mathbf{0} \mid\\ & \quad\quad c_5(\widetilde{u_y}).y_{1,1}(z').d_6\langle z'y_{1,2}\rangle.\mathbf{0} \mid\\ & \quad\quad c_6(z'y_{1,2}).y_{1,2}\langle\mathsf{ACK}\rangle.d_7\langle z'\rangle.\mathbf{0} \mid c_7(w).\mathbf{0} \mid\\ & \quad\quad c_1(\widetilde{u}).d_2\langle\widetilde{u_x}\rangle.d_5\langle\widetilde{u_y}\rangle.\mathbf{0})\end{aligned} \tag{4.5}$$

It can be shown that $\mathfrak{D}(P_h)$ is typable by using the appropriate environments. The intuitive argument for this is that there are no shared variables and each pairs of co-variables implement complementary behaviors by the definition of the decomposition (cf. Def. 4.51). It is also important to notice the way in which the parallel sub-processes in $\mathfrak{D}(P_h)$ implement the individual prefixes of P_h. For instance:

- $x\langle\mathsf{REQ}\rangle$ is represented by $c_2(\widetilde{u_x}).x_{1,1}\langle\mathsf{REQ}\rangle.d_3\langle x_{1,2}\rangle.\mathbf{0}$.
- $x(z)$ is represented by $c_3(x_{1,2}).x_{1,2}(z).d_4\langle z\rangle.\mathbf{0}$.
- $\mathbf{0}$ is represented by $c_4(w).\mathbf{0}$.

The subprocesses of the decomposition $\mathfrak{D}(P_h)$ that do not correspond to a prefix in P_h are used as auxiliary processes that trigger the prefix representations. It can be shown that $\mathfrak{D}(P_h)$ preserves the causal order of the source specifications [Par00]. Using the semantics in Fig. 2.1:

$$\begin{aligned}\mathfrak{D}(P_h) \longrightarrow {} & (\boldsymbol{\nu}c_1d_1)(\boldsymbol{\nu}c_2d_2)(\boldsymbol{\nu}c_3d_3)(\boldsymbol{\nu}c_4d_4)(\boldsymbol{\nu}c_5d_5)\\ & (\boldsymbol{\nu}c_6d_6)(\boldsymbol{\nu}c_7d_7)(\boldsymbol{\nu}x_{1,1}y_{1,1})(\boldsymbol{\nu}x_{1,2}y_{1,2})\\ & \quad (c_2(\widetilde{u_x}).x_{1,1}\langle\mathsf{REQ}\rangle.d_3\langle x_{1,2}\rangle.\mathbf{0} \mid c_3(x_{1,2}).x_{1,2}(z).d_4\langle z\rangle.\mathbf{0} \mid c_4(w).\mathbf{0} \mid\\ & \quad c_5(\widetilde{u_y}).y_{1,1}(z').d_6\langle z'y_{1,2}\rangle.\mathbf{0} \mid c_6(z'y_{1,2}).y_{1,2}\langle\mathsf{ACK}\rangle.d_7\langle z'\rangle.\mathbf{0} \mid\\ & \quad c_7(w).\mathbf{0} \mid d_2\langle\widetilde{u_x}\rangle.d_5\langle\widetilde{u_y}\rangle.\mathbf{0})\\ \longrightarrow^2 {} & (\boldsymbol{\nu}c_1d_1)(\boldsymbol{\nu}c_2d_2)(\boldsymbol{\nu}c_3d_3)(\boldsymbol{\nu}c_4d_4)(\boldsymbol{\nu}c_5d_5)\\ & (\boldsymbol{\nu}c_6d_6)(\boldsymbol{\nu}c_7d_7)(\boldsymbol{\nu}x_{1,1}y_{1,1})(\boldsymbol{\nu}x_{1,2}y_{1,2})\\ & \quad (x_{1,1}\langle\mathsf{REQ}\rangle.d_3\langle x_{1,2}\rangle.\mathbf{0} \mid c_3(x_{1,2}).x_{1,2}(z).d_4\langle z\rangle.\mathbf{0} \mid c_4(w).\mathbf{0} \mid\\ & \quad y_{1,1}(z').d_6\langle z'y_{1,2}\rangle.\mathbf{0} \mid c_6(z'y_{1,2}).y_{1,2}\langle\mathsf{ACK}\rangle.d_7\langle z'\rangle.\mathbf{0} \mid c_7(w).\mathbf{0})\\ \longrightarrow^3 {} & (\boldsymbol{\nu}c_1d_1)(\boldsymbol{\nu}c_2d_2)(\boldsymbol{\nu}c_3d_3)(\boldsymbol{\nu}c_4d_4)(\boldsymbol{\nu}c_5d_5)\\ & (\boldsymbol{\nu}c_6d_6)(\boldsymbol{\nu}c_7d_7)(\boldsymbol{\nu}x_{1,1}y_{1,1})(\boldsymbol{\nu}x_{1,2}y_{1,2})\\ & \quad (x_{1,2}(z).d_4\langle z\rangle.\mathbf{0} \mid c_4(w).\mathbf{0} \mid y_{1,2}\langle\mathsf{ACK}\rangle.d_7\langle z'\rangle.\mathbf{0} \mid c_7(w).\mathbf{0}) \longrightarrow^3 \mathbf{0}\end{aligned}$$

An interesting aspect of using a decomposition such as the one in Def. 4.51 is that we can prove that $[\![\mathfrak{D}(P_h)]\!] \longrightarrow_1^{*} \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![\mathbf{0}]\!]$. This follows from the operational correspondence property of the encoding (cf. Thm. 4.20 and Thm. 4.21). Taking this into account, we can use the parallel sub-processes in P''_h and place them inside lcc processes allowing them to specify features that are not available for $\pi_{\mathsf{OR}}^{\lightning}$ processes. Let us now define some shorthands for each one of the parallel sub-processes representing prefixes in P''_h:

$$\begin{aligned}
\mathfrak{D}_1 &= c_2(\widetilde{u_x}).x_{1,1}\langle \mathsf{REQ}\rangle.d_3\langle x_{1,2}\rangle.\mathbf{0}\\
\mathfrak{D}_2 &= c_3(x_{1,2}).x_{1,2}(z).d_4\langle z\rangle.\mathbf{0}\\
\mathfrak{D}_3 &= c_5(\widetilde{u_y}).y_{1,1}(z').d_6\langle z'y_{1,2}\rangle.\mathbf{0}\\
\mathfrak{D}_4 &= c_6(z'y_{1,2}).y_{1,2}\langle \mathsf{ACK}\rangle.d_7\langle z'\rangle.\mathbf{0}
\end{aligned}$$

Also, let R represent all the remaining parallel sub-processes that appear in $\mathfrak{D}(P_h)$. Applying the encoding $[\![\cdot]\!]$ (cf. Fig. 4.2) to P''_h and thanks to its compositionality property (cf. Thm. 4.9) we have that:

$$[\![\mathfrak{D}(P_h)]\!] = C_{\widetilde{x}\widetilde{y}}[[\![\mathfrak{D}_1]\!] \parallel [\![\mathfrak{D}_2]\!] \parallel [\![\mathfrak{D}_3]\!] \parallel [\![\mathfrak{D}_4]\!] \parallel [\![R]\!]] \tag{4.6}$$

with $C_{\widetilde{x}\widetilde{y}}[\cdot]$ and where $\widetilde{x}\widetilde{y}$ corresponds to a sequence containing all the variables initialized in P''_h (cf. (4.5)).

With the additional control earned by using the decomposition, we can circumvent the issues introduced by the lack of constructs for sequentiality in `lcc`. We can also use the code snippets generated by $[\![\mathfrak{D}(P_h)]\!]$ to build an `lcc` process that specifies communication behavior that depends on timing constraints. For example, to represent the behavior required in P'_h (cf. (4.4)); i.e., *"an acknowledgment message* ACK *should be sent (by the server) no later than three seconds after receiving the request message* REQ*"* we could use the following process. Assume the existence of a clock u that controls the execution of $\mathfrak{D}_2$ as shown below:

$$Q_1 = C_{\widetilde{x}\widetilde{y}}[[\![\mathfrak{D}_1]\!] \parallel \forall\epsilon(clock(u) \leq 3 \rightarrow [\![\mathfrak{D}_2]\!]) \parallel [\![\mathfrak{D}_3]\!] \parallel [\![\mathfrak{D}_4]\!] \parallel [\![R]\!]]$$

Process Q_1 represents a variant of $[\![\mathfrak{D}(P_h)]\!]$ where process $\mathfrak{D}_2$ appears guarded by the constraint $clock(u) \leq 3$. The function $clock(u)$ obtains the timed elapsed in u. The process above enforces the deadline of three seconds by suspending $[\![\mathfrak{D}_2]\!]$ in case the timing constraint is not met.

Using the ideas presented above we shall now develop `lcc` representations of the timed patterns in § 1.6.

4.4.2 Request-Response Timeout

The request-response timeout pattern in § 1.6 is a generalization of the protocol implemented by process P_h in (4.3). In this section we shall use the encoding in Fig. 4.2 to represent the generalized version of this pattern, as it was presented in § 1.6. We first recall the pattern descriptions from both the client and server side:

(a) *Server side:* After receiving a message REQ from A, B must send the acknowledgment ACK within t time units.

(b) *Client side:* After sending a message REQ to B, A must be able to receive the acknowledgment ACK from B within t time units.

We now present a process that represents this pattern. For this, we use a more general version of P_h in (4.3), annotated to informally describe the timing requirements of the pattern:

$$P_r = (\boldsymbol{\nu} xy)(\underbrace{x\langle \mathsf{REQ}\rangle.x(z).Q_1 \mid y(z).y\langle \mathsf{ACK}\rangle}_{t}.Q_2) \tag{4.7}$$

where the time elapsed between the reception of the request and the acknowledgment must not exceed t time units.

We now present a decomposition for P_r, following Def. 4.51. Assume that R contains the breakdown of processes Q_1 and Q_2—they are not needed in this example.

$$\mathfrak{D}(P_r) = (\boldsymbol{\nu}\widetilde{u})(\mathfrak{D}_1 \mid \mathfrak{D}_2 \mid \mathfrak{D}_3 \mid \mathfrak{D}_4 \mid R) \tag{4.8}$$

where $\widetilde{u}$ can be obtained from Def. 4.51 and each of the parallel sub-processes $\mathfrak{D}_i$ $(i \in \{1,2,3,4\})$ represents a prefix in P_r, as given below:

$$\begin{aligned}
\mathfrak{D}_1 &= c_2(\widetilde{u_x}).x_{1,1}\langle \mathsf{REQ}\rangle.d_3\langle \widetilde{u_x} \setminus x_{1,1}\rangle.\mathbf{0}\\
\mathfrak{D}_2 &= c_3(\widetilde{u_x} \setminus x_{1,1}).x_{1,2}(z\widetilde{z_1}).d_4\langle \widetilde{u_x} z\widetilde{z_1} \setminus x_{1,1}x_{1,2}\rangle.\mathbf{0}\\
\mathfrak{D}_3 &= c_5(\widetilde{u_y}).y_{1,1}(z'\widetilde{z_2}).d_6\langle \widetilde{u_y} z'\widetilde{z_2} \setminus y_{1,1}\rangle.\mathbf{0}\\
\mathfrak{D}_4 &= c_6(\widetilde{u_y} z'\widetilde{z_2} \setminus y_{1,1}).y_{1,2}\langle \mathsf{ACK}\rangle.d_7\langle \widetilde{u_y} z'\widetilde{z_2} \setminus y_{1,1}y_{1,2}\rangle.\mathbf{0}
\end{aligned}$$

By applying the encoding $[\![\cdot]\!]$ and thanks to Thm. 4.9 (i.e., compositionality), we can use the translations of every $\mathfrak{D}_i$ $(i \in \{1,2,3,4\})$ inside an lcc process that represents the request-response timeout pattern:

$$Q_1 = C_{\widetilde{u}}[[\![\mathfrak{D}_1]\!] \parallel \forall\epsilon(\mathit{clock}(x) \leq t \rightarrow [\![\mathfrak{D}_2]\!]) \parallel [\![\mathfrak{D}_3]\!] \parallel [\![\mathfrak{D}_4]\!] \parallel [\![R]\!]]$$

where $\widetilde{u}$ can be obtained by following Def. 4.51.

It is also possible to add additional behavior to process Q_1. For example, we can use the *nondeterministic* construct from lcc to signal that the process reduces to a failure whenever $\mathit{clock}(u) > t$:

$$\begin{aligned}
Q_2 = \exists \widetilde{x}\widetilde{y}.\big([\![\mathfrak{D}_1]\!] \parallel (\forall\epsilon(\mathit{clock}(u) \leq t \rightarrow [\![\mathfrak{D}_2]\!]) + \forall\epsilon(\mathit{clock}(u) > t \rightarrow Q_f)) \parallel\\
[\![\mathfrak{D}_3]\!] \parallel [\![\mathfrak{D}_4]\!] \parallel [\![R]\!]\big)
\end{aligned}$$

In this case, whenever $\mathit{clock}(u) > t$, the process reduces to a process Q_f which represents the actions that must be taken in case the timing constraint is not met (i.e., a failure protocol). For example, if Q_f represents an error process, the behavior would be reminiscing of the input operators with deadlines presented in [BMVY19], which also reduce to a failure whenever the deadline is not met.

Observe that thanks to the operational correspondence property of the encoding (cf. Thm. 4.20 and Thm. 4.21), process Q_2 satisfies the behavior of its source term, provided that the timing constraint is met.

4.4.3 Messages in a Time-Frame

We first recall the messages in a time-frame pattern, as it was presented in § 1.6:

(a) *Interval:* A is allowed to send B at most k messages, and at time intervals of at least t and at most r time units.

(b) *Overall time-frame:* A is allowed to send B at most k messages in the overall time-frame of at least t and at most r time units.

To represent these two variants of the pattern using $[\![\cdot]\!]$, we first present two $\pi^{\lightning}_{\mathsf{OR}}$ processes, annotated to indicate the timing constraints:

$$P_{ti} = (\boldsymbol{\nu} xy)(x\underbrace{\langle \mathsf{M}_1\rangle.x\langle}_{t_1}\underbrace{\mathsf{M}_2\rangle.x\langle}_{t_1}\underbrace{\mathsf{M}_3\rangle.x\langle}_{t_1}\mathsf{M}_4\rangle.P_x \mid P_y) \tag{4.9}$$

$$P_{to} = (\boldsymbol{\nu} xy)(\underbrace{x\langle \mathsf{M}_1\rangle.x\langle \mathsf{M}_2\rangle.x\langle \mathsf{M}_3\rangle.x\langle \mathsf{M}_4\rangle}_{t_2}.P_x \mid P_y) \tag{4.10}$$

Process P_{ti} and P_{to} differ on their timing constraints (i.e., the annotations below each process). They both send four messages that must be received by some P_y (which we leave unspecified, as it is not central in this example). Process P_{ti} is meant to represent the interval pattern, in which we require there to be *at least* t_1 time units between each message. On the other hand, process P_{to} represents the overall time-frame pattern in which all the four messages must be sent in an overall time of t_2 time units. In this example we use *clock*(u_1) and *clock*(u_2) to represent the time elapsed in clocks u_1 and u_2, respectively.

Before presenting a possible lcc specification, we consider the decompositions of $\mathfrak{D}(P_{ti})$ and $\mathfrak{D}(P_{to})$ (cf. Def. 4.51). Assume that R_x contains all the processes related to the breakdown of P_x and that R_y contains all the processes from the decomposition related to the breakdown of P_y. The sequence $\widetilde{u}$ can be derived by following Def. 4.51.

$$\mathfrak{D}(P_{to}) = \mathfrak{D}(P_{ti}) = (\boldsymbol{\nu}\widetilde{u})(\mathfrak{D}_1 \mid \mathfrak{D}_2 \mid \mathfrak{D}_3 \mid \mathfrak{D}_4 \mid R_x \mid R_y) \tag{4.11}$$

where processes $\mathfrak{D}_i$ ($i \in \{1,2,3,4\}$) correspond to:

$$\begin{aligned}
\mathfrak{D}_1 &= c_2(\widetilde{u_x}).x_{1,1}\langle \mathsf{M}_1\rangle.d_3\langle \widetilde{u_x} \setminus x_{1,1}\rangle.\mathbf{0}\\
\mathfrak{D}_2 &= c_3(\widetilde{u_x} \setminus x_{1,1}).x_{1,2}\langle \mathsf{M}_2\rangle.d_4\langle \widetilde{u_x} \setminus x_{1,1}x_{1,2}\rangle.\mathbf{0}\\
\mathfrak{D}_3 &= c_4(\widetilde{u_x} \setminus x_{1,1}x_{1,2}).x_{1,3}\langle \mathsf{M}_3\rangle.d_5\langle \widetilde{u_x} \setminus x_{1,1}x_{1,2}x_{1,3}\rangle.\mathbf{0}\\
\mathfrak{D}_4 &= c_5(\widetilde{u_x} \setminus x_{1,1}x_{1,2}x_{1,3}).x_{1,4}\langle \mathsf{M}_4\rangle.d_6\langle \widetilde{u_x} \setminus x_{1,1}x_{1,2}x_{1,3}x_{1,4}\rangle.\mathbf{0}
\end{aligned}$$

By applying the encoding $[\![\cdot]\!]$ and thanks to its compositionality (cf. Thm. 4.9):

$$[\![\mathfrak{D}(P_{ti})]\!] = [\![\mathfrak{D}(P_{to})]\!] = C_{\widetilde{x}\widetilde{y}}[[\![\mathfrak{D}_1]\!] \parallel [\![\mathfrak{D}_2]\!] \parallel [\![\mathfrak{D}_3]\!] \parallel [\![\mathfrak{D}_4]\!] \parallel [\![R_x]\!] \parallel [\![R_y]\!]]$$

We now represent the variants of the timed pattern above (i.e., (a) and (b)) using the encoding $[\![\cdot]\!]$:

(1) *Interval:* This variant of the pattern requires that for every sent message, the next message is sent with a delay of at least t_1 time units. This means that we want to guard the snippets obtained with the decomposition in such a way that the processes are suspended until the interval t_1 has passed:

$$\begin{aligned}
Q_1 = C_{\widetilde{x}\widetilde{y}}[[\![\mathfrak{D}_1]\!] \parallel \forall\epsilon\big(&clock(u_1) > t_1 \rightarrow [\![\mathfrak{D}_2]\!] \parallel \overline{rst(u_1)} \parallel\\
&\forall\epsilon\big(clock(u_1) > t_1 \rightarrow [\![\mathfrak{D}_3]\!] \parallel \overline{rst(u_1)} \parallel\\
&\quad\forall\epsilon\big(clock(u_1) > t_1 \rightarrow [\![\mathfrak{D}_4]\!] \parallel \overline{rst(u_1)} \parallel\\
&\qquad\forall\epsilon\big(clock(u_1) > t_1 \rightarrow [\![R_x]\!]\big)\big)\big)\big) \parallel [\![R_y]\!]]
\end{aligned}$$

where constraint $rst(u_1)$ tells the store that clock u_1 must be reset. Process Q_1 consists of nested abstractions. Each abstraction is used to make the next synchronization wait until the delay is satisfied. To achieve this, we guard each abstraction with constraint $clock(u_1) > t_1$. In this way, we ensure that the process representing each prefix is delayed accordingly. We also ensure that whenever the timing constraint is met, the clock is reset to allow for the time to count from the start once again.

(2) *Overall Time-Frame:* This variant of the pattern can be represented by changing the timing constraint in Q_1 to $clock(u_2) \leq t_2$ and by not resetting the clock inside every ask process:

$$\begin{aligned} Q_2 = C_{\widetilde{x}\widetilde{y}}[&\forall\epsilon\big(clock(u_2) \leq t_2 \to [\![\mathfrak{D}_1]\!] \parallel \forall\epsilon\big(clock(u_2) \leq t_2 \to [\![\mathfrak{D}_2]\!] \parallel \\ &\forall\epsilon\big(clock(u_2) \leq t_2 \to [\![\mathfrak{D}_3]\!] \parallel \forall\epsilon\big(clock(u_2) \leq t_2 \to [\![\mathfrak{D}_4]\!] \parallel \overline{rst(u_2)}\big)\big) \parallel \\ &\forall\epsilon\big(clock(u_2) \leq t_2 \to [\![R_x]\!]\big)\big)\big) \parallel [\![R_y]\!]] \end{aligned}$$

The overall time of the communications can be checked because we only reset the clock at the end of the complete interaction.

Remark 4.52. Similarly to the lcc representation of the request-response timeout, it is possible to use non-determinism in lcc to extend the behavior of these lcc implementations. Moreover, both Q_1 and Q_2 preserve the behavior of their source process, assuming that all the timing constraints are satisfied. This is because of the operational correspondence property of the encoding (cf. Thm. 4.20 and Thm. 4.21).

4.4.4 Action Duration

We recall the action duration timed pattern as described in § 1.6.

(a) The time elapsed between two actions of the same participant A must not exceed t time units.

Using $\pi^{\natural}_{\mathsf{OR}}$, we can use the following process to represent the action duration pattern by annotating with the appropriate timing constraints:

$$P_a = (\boldsymbol{\nu}xy)(\underbrace{x\langle\mathsf{M}_1\rangle.x\langle\mathsf{M}_2\rangle}_{t}.P_x \mid P_y) \tag{4.12}$$

where we require that there are no more than t time units between the first output and the second one. We also require that the time-frame is $0 \leq clock(u) \leq t$. The decomposition is obtained by applying Def. 4.51 to P_a. As with the previous pattern, assume that R_x and R_y gather all the processes obtained from the decomposition of P_x and P_y. The most important prefixes obtained from the decomposition are described below:

$$\begin{aligned} \mathfrak{D}_1 &= c_2(\widetilde{u_x}).x_{1,1}\langle\mathsf{M}_1\rangle.d_3\langle\widetilde{u_x} \setminus x_{1,1}\rangle.\mathbf{0} \\ \mathfrak{D}_2 &= c_3(\widetilde{u_x} \setminus x_{1,1}).x_{1,2}\langle\mathsf{M}_2\rangle.d_4\langle\widetilde{u_x} \setminus x_{1,1}x_{1,2}\rangle.\mathbf{0} \end{aligned}$$

We can then apply the encoding $[\![\cdot]\!]$ and its compositionality property (cf. Thm. 4.9) to obtain:

$$[\![\mathfrak{D}(P_a)]\!] = C_{\widetilde{x}\widetilde{y}}[[\![\mathfrak{D}_1]\!] \parallel [\![\mathfrak{D}_2]\!] \parallel [\![R_x]\!] \parallel [\![R_y]\!]]$$

Similarly to the previous two patterns, we use linear abstractions to ensure that $[\![\mathfrak{D}_2]\!]$ is only executed within 0 and t time units:

$$Q_1 = C_{\widetilde{x}\widetilde{y}}[[\![\mathfrak{D}_1]\!] \parallel \forall\epsilon\big(clock(u) \leq t \rightarrow [\![\mathfrak{D}_2]\!]\big) \parallel [\![R_y]\!] \parallel [\![R_y]\!]]$$

Above, constraint $clock(u) \leq t$ guards the execution of $[\![\mathfrak{D}_2]\!]$. This means that this process can only execute if the time constraint is satisfied. Notice that it is also possible to consider time-frame different from $0 \leq clock(u) \leq t$ by using constraint $t' \leq clock(u) \leq t$ in the linear abstraction, where t' represents the lower bound of of the time-frame. As with the previous patterns, the operational correspondence property of the encoding ensures that the behavior is preserved, assuming that the timing constraints are met.

4.4.5 Repeated Constraint: Discussion

We first recall the pattern description as given in § 1.6:

(a) `A` must send (and unbounded number of) messages to `B` every t time units.

This pattern induces infinite behavior as it requires an unbounded number of messages to be received within some time-frame. Unfortunately, this implies that the repeated constraint pattern cannot be expressed as a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program (cf. § 3.1.1). To understand why, let us consider the following *untyped* $\pi_{\mathsf{OR}}^{\lightning}$ process:

$$P_c = (\boldsymbol{\nu} loop_1 loop_2)(\boldsymbol{\nu} xy)(\underbrace{loop_1\langle loop_1\rangle.\mathbf{0} \mid * \, loop_2(z).(x\langle \mathsf{M}_1\rangle.loop_1\langle z\rangle.\mathbf{0})}_{t} \mid P_y) \tag{4.13}$$

This process corresponds to the encoding of recursion that uses name passing and replication [AGPV06]. Intuitively, endpoints $loop_1$ and $loop_2$ are used to iterate whenever M_1 is received. The most important thing to notice about P_c is that it has output races. Therefore, this process will be ruled out by the type system in § 3.1.1. Since P_c is untyped, its encoding does not satisfy the correctness criteria. Therefore, we cannot guarantee that $[\![P_c]\!]$ will preserve the desired behavior. Because of this, we conjecture that it is not possible to represented the repeated constraint pattern using the encoding.

5 Encoding π_{E} in $\texttt{lcc}^{\mathsf{p}}$

In this chapter we present a translation from π_{E} into $\texttt{lcc}^{\mathsf{p}}$. In § 5.1 we formalize a secure constraint system, used by $\texttt{lcc}^{\mathsf{p}}$, as well as formally defining the translation. We then prove that the translation is name invariant and compositional (cf. Def. 2.3) in § 5.2. In § 5.3, we prove that the translation is both operationally complete and sound. In this section we summarize our correctness results and conclude that the translation is a valid encoding (cf. Def. 2.3). We conclude with § 5.4, where we show that our translation is well-typed with respect to the type system in § 3.3.

5.1 The Translation

In this section we present a translation from π_{E} into $\texttt{lcc}^{\mathsf{p}}$. The translation uses similar ideas to the ones presented in § 4.1. In § 5.1.1 we present a constraint system aimed at modeling session with security primitives such as public keys, private keys, and encryption. Then, in § 5.1.2, we first instantiate the behavioral equivalences required in Def. 3.102(2), before formally introducing the new translation: $\llbracket\cdot\rrbracket_f$.

5.1.1 *A Constraint System For Secure Sessions*

In the presence of abstractions with local information (cf. § 3.3), processes may query the constraint store about local and global constraints. We must avoid publishing local information (e.g., session identifiers, encryption keys, nonces) in the (global) constraint store. To this end, the translation $\llbracket\cdot\rrbracket_f$ relies on a *security constraint system* that combines local and global information with cryptographic primitives. This is another instantiation of Def. 2.27, which builds upon similar constraint systems given in [OV08b, OV08a, HL09]:

Definition 5.1 (Security Constraint System). Let Σ and Δ be the function symbols and predicates given in Fig. 5.1. The *security constraint system* is the tuple $\langle \mathcal{S}, \Sigma \cup \Delta, \vdash_{\mathcal{S}} \rangle$, where $\mathcal{S}$ is the set of all constraints obtained by using linear operators !, $\otimes$, and $\exists$ over the functions of Σ and predicates of Δ, and where $\vdash_{\mathcal{S}}$ is given by the rules in Fig. 2.4, extended with the non-logical axioms in Fig. 5.2.

We comment on the signatures for functions and predicates given in Fig. 5.1, which differ from those in Fig. 4.1 in several respects. First, Σ includes functions for handling encryption (used to model session initiation) and tuples: given a location n as an (unrestricted) argument, functions p, r, and s return the public, restricted (i.e., private), and symmetric key of n, respectively. We use k to range over function identifiers p, r, and s and write k^{-1} to denote inverse of k, defined as $\mathsf{s}^{-1} = \mathsf{s}$ and $\mathsf{p}^{-1} = \mathsf{r}$, as expected. Function $\mathsf{enc}(k; m)$ returns message m encrypted with a key k, which is restricted. Decryption is enabled via entailment/deduction in the constraint system (Fig. 5.2). Lastly, function $\mathsf{tup}_j(\epsilon; \widetilde{x})$ models a tuple of j (unrestricted) elements ($j \geq 1$). As usual in first-order logic, we will recursively define *terms* to be variables, functions applied to variables and function applied to other terms.

As before, predicates in Δ are used to represent session communication; in this case, we exploit the distinction between restricted and unrestricted variables. We require another set of predicates, different from those in Fig. 4.1 (we use different type font styles to highlight the differences). Predicate $\text{snd}(x; y)$ takes two arguments: a (restricted) session key and an unrestricted message. That is, while the communication subject should be private to the session, its associated communication object can be publicly available. Predicate $\text{rcv}(x, y; \epsilon)$ models the acknowledgment of snd, and contains the session key and the value. In this case, both subject and object are restricted. Predicates sel and bra model communication of a label object. Predicate $\{x{:}y\}$ declares x, y as co-variables; both endpoints are restricted. Novelties with respect to the predicates in Fig. 4.1 are $\text{loc}_\rho(\epsilon; x)$, $ch(x; \epsilon)$, and $out(\epsilon; m)$. While predicate $\text{loc}_\rho(\epsilon; x)$ asserts that the (unrestricted) location x is in the (unrestricted) set of locations ρ, predicate $ch(x; \epsilon)$ declares x as a channel. Finally, predicate $out(\epsilon; m)$ indicates that a message m has been sent through a public, potentially insecure medium; it is used to model service agreement during session establishment.

The following notation will be useful in processes.

Notation 5.2. We shall write function $\mathsf{enc}(k; x)$ as $\{x\}_k$. Similarly, we shall write tuple $\mathsf{tup}_n(\epsilon; x_1, \ldots, x_n)$, with $n \geq 1$ as $\langle x_1, \ldots, x_n \rangle$.

We now comment on the entailment rules given in Fig. 5.2. Rule (E:Loc) is used to verify that a location belongs to a set ρ: if the location is known ($out(\epsilon; n)$) and it belongs to a set ρ then we can obtain predicate $\text{loc}_\rho(\epsilon; n)$. Rule (E:Cov-Comm) states that the co-variable constraint is commutative. Rule (E:Cov) relates the two endpoints, known only to the participants of that session: it states that given an endpoint id $ch(x; \epsilon)$ and the co-variable constraint, we may obtain the id for the other endpoint y. Rule (E:Enc) allows us to encode a message x with a given key y. Rule (E:Dec) expresses that the output of any function of known output values can be inferred using the right key. Rule (E:Tup) allows us to create an n-tuple from a sequence of n-messages. Rule (E:Key) gives the key of a message. Rules (E:Proj1) and (E:Proj2) handle tuples: while Rule (E:Proj1) allows us to project individual elements, discard-

$$\begin{aligned}
\Sigma &::= \mathsf{p}(\epsilon; n) \mid \mathsf{r}(\epsilon; n) \mid \mathsf{s}(\epsilon; n) \mid \mathsf{enc}(k; m) \mid \mathsf{tup}_j(\epsilon; \widetilde{x}) \\
\Delta &::= \mathrm{snd}(x; y) \mid \mathrm{rcv}(x, y; \epsilon) \mid \mathrm{sel}(x; l) \mid \mathrm{bra}(x, l; \epsilon) \mid \{x{:}y\} \mid \\
&\quad\ \mathrm{loc}_\rho(\epsilon; x) \mid ch(x; \epsilon) \mid out(\epsilon; x)
\end{aligned}$$

Figure 5.1: Security constraint system: Function and predicate symbols (cf. Def. 5.1).

$$\text{(E:Loc)}\ \frac{c \vdash_{\mathcal{S}} out(\epsilon; n) \quad n \in \rho}{c \vdash_{\mathcal{S}} \mathrm{loc}_\rho(\epsilon; n)} \qquad \text{(E:Cov-Comm)}\ \frac{c \vdash_{\mathcal{S}} \{x{:}y\}}{c \vdash_{\mathcal{S}} \{y{:}x\}} \qquad \text{(E:Cov)}\ \frac{c \vdash_{\mathcal{S}} ch(x) \quad c \vdash_{\mathcal{S}} \{x{:}y\} \quad x \neq y}{c \vdash_{\mathcal{S}} ch(y)}$$

$$\text{(E:Enc)}\ \frac{c \vdash_{\mathcal{S}} out(\epsilon; x) \quad c \vdash_{\mathcal{S}} out(\epsilon; k)}{c \vdash_{\mathcal{S}} out(\epsilon; \mathsf{enc}(k; x))}$$

$$\text{(E:Dec)}\ \frac{k \in \{\mathsf{s}, \mathsf{p}\} \quad c \vdash_{\mathcal{S}} out(\epsilon; k^{-1}(x)) \quad c \vdash_{\mathcal{S}} out(\epsilon; \mathsf{enc}(k(x); m))}{c \vdash_{\mathcal{S}} out(\epsilon; m) \otimes out(\epsilon; \mathsf{enc}(k(x); m))}$$

$$\text{(E:Tup)}\ \frac{\forall i \in \{1, \dots, j\}.\, c \vdash_{\mathcal{S}} out(\epsilon; e_i)}{c \vdash_{\mathcal{S}} out(\epsilon; \mathsf{tup}_j(\epsilon; e_1, \dots, e_j))} \qquad \text{(E:Key)}\ \frac{c \vdash_{\mathcal{S}} out(\epsilon; x) \quad k \in \{\mathsf{s}, \mathsf{p}, \mathsf{r}\}}{c \vdash_{\mathcal{S}} out(\epsilon; k(x))}$$

$$\text{(E:Proj1)}\ \frac{c \vdash_{\mathcal{S}} out(\epsilon; \mathsf{tup}_j(\epsilon; e_1, \dots, e_i, \dots, e_j))}{c \vdash_{\mathcal{S}} out(\epsilon; e_i)}$$

$$\text{(E:Proj2)}\ \frac{c \vdash_{\mathcal{S}} out(\epsilon; \mathsf{tup}_j(\epsilon; e_1, \dots, e_i, \dots, e_j))}{c \vdash_{\mathcal{S}} out(\epsilon; e_i) \otimes out(\epsilon; \mathsf{tup}_j(\epsilon; e_1, \dots, e_j))}$$

Figure 5.2: Security constraint system: non-logical axioms (cf. Def. 5.1).

ing the remaining elements, Rule (Proj2) allows to project any element of the tuple while preserving the remaining tuple elements.

5.1.2 *Mapping* π_{E} *Processes Into* $\mathtt{lcc}^{\mathsf{p}}$ *Processes*

Before formally presenting $[\![\cdot]\!]_f$, we would like to define the sets $\mathcal{D}$ and $\mathcal{E}$, required to instantiate the barbed congruence required in Def. 3.102(2). Similarly to $[\![\cdot]\!]$, we first characterize a set of output and complete observables. The main differences from Def. 4.2 are: (1) the addition of constraint $\exists \widetilde{z}.out(\epsilon; t)$, where t is a term, and (2) the distinction between restricted/unrestricted variables in each predicate.

Definition 5.3 (Complete and Output Observables for π_{E}). Let $\mathcal{S}$ be the security constraint system in Def. 5.1. We define the set $\mathcal{D}_{\pi_{\mathsf{E}}}$ of *output observables* of $[\![\cdot]\!]_f$ as follows:

$$\begin{aligned}
\mathcal{D}_{\pi_{\mathsf{E}}} \overset{\mathsf{def}}{=}\ & \{\exists \widetilde{z}.\mathrm{snd}(x; v) \mid x, v \in \mathcal{V}_\pi \wedge x \in \widetilde{z} \wedge (v \in \widetilde{z} \vee v \notin \mathcal{V}_\pi)\} \\
& \cup \{\exists \widetilde{z}.\mathrm{sel}(x; l) \mid x \in \mathcal{V}_\pi \wedge l \in \mathcal{B}_\pi \wedge x \in \widetilde{z}\} \cup \{\exists \widetilde{z}.out(\epsilon; t) \mid t \text{ is a term in } \mathcal{S}\}
\end{aligned}$$

We also define $\mathcal{D}^{\star}_{\pi_{\mathsf{E}}}$, the complete observables of $[\![\cdot]\!]_f$, by extending $\mathcal{D}_{\pi_{\mathsf{E}}}$ as follows:

$$\begin{aligned}\mathcal{D}^{\star}_{\pi_{\mathsf{E}}} \overset{\mathsf{def}}{=} \mathcal{D}_{\pi_{\mathsf{E}}} &\cup \{\mathtt{tt}\} \cup \{\exists \widetilde{z}.\mathrm{rcv}(x, y; \epsilon) \mid x \in \mathcal{V}_{\pi} \setminus \{y\} \wedge x \in \widetilde{z}\}\\ &\cup \{\exists \widetilde{z}.\mathrm{bra}(x, l; \epsilon) \mid x \in \mathcal{V}_{\pi} \wedge l \in \mathcal{B}_{\pi} \wedge x \in \widetilde{z}\}\\ &\cup \{\exists \widetilde{z}.\mathrm{loc}_{\rho}(\epsilon; x) \mid x \in \mathcal{V}_{\pi} \wedge \rho \subseteq \Omega_{\pi} \wedge x \in \widetilde{z}\}\\ &\cup \{\exists \widetilde{z}.ch(x; \epsilon) \mid x \in \mathcal{V}_{\pi} \wedge x \in \widetilde{z}\}\end{aligned}$$

Notice that constraint $\exists \widetilde{z}.out(\epsilon; t)$ will also appear in the set of output observables. This is a sensible choice, as these constraints represent the main form of synchronization during session establishment. Considering this, we can then state that: $\mathcal{D} = \mathcal{D}_{\pi_{\mathsf{E}}}$ and $\mathcal{E} = \mathcal{S}$. Using the previous fact, we can now introduce the precise instances of Def. 2.33 and Def. 2.34, required for Def. 3.102(2).

Definition 5.4 (Weak O-Barbed Bisimilarity and Congruence). We define *weak o-barbed bisimilarity* and *weak o-barbed congruence* for $\mathtt{lcc}^{\mathsf{p}}$ processes as follows:

1. Weak o-barbed bisimilarity, denoted $\approx_1^{\pi_{\mathsf{E}}}$, arises from Def. 2.33 as the weak $\mathcal{D}_{\pi_{\mathsf{E}}}\mathcal{S}$-barbed bisimilarity.
2. Weak o-barbed congruence, denoted $\cong_1^{\pi_{\mathsf{E}}}$, arises from Def. 2.34 as the weak $\mathcal{D}_{\pi_{\mathsf{E}}}\mathcal{S}$-barbed congruence.

We now define the translation of π_{E} into $\mathtt{lcc}^{\mathsf{p}}$. One of the challenges associated to a translation of session establishment is that the use of abstractions over constraints containing only unrestricted predicates enables any external process to abstract (private) session keys. To solve this issue, our translation of session establishment includes an explicit authentication protocol: the Needham-Schroeder-Lowe (NSL) protocol [Low96]. The translation is defined next; it is parameterized by a set of pairs of co-variables, denoted f.

Definition 5.5 (Translation of π_{E} into $\mathtt{lcc}^{\mathsf{p}}$). Let $\mathcal{L}_{\pi_{\mathsf{E}}} = \langle \pi_{\mathsf{E}}, \longrightarrow_{\mathsf{N}}, \equiv_{\mathsf{S}} \rangle$ and $\mathcal{L}_{\mathtt{lcc}^{\mathsf{p}}} = \langle \mathtt{lcc}^{\mathsf{p}}, \longrightarrow_{1}, \cong_1^{\pi_{\mathsf{E}}} \rangle$. We define the translation from $\mathcal{L}_{\pi_{\mathsf{E}}}$ into $\mathcal{L}_{\mathtt{lcc}^{\mathsf{p}}}$ as the pair $\langle [\![\cdot]\!]_f, \varphi_{[\![\cdot]\!]_f} \rangle$, where:

(a) $[\![\cdot]\!]_f$ is the process mapping defined in Fig. 5.3.

(b) $\varphi_{[\![\cdot]\!]_f}$ is defined as the identity, as in Def. 4.4(b) .

Before providing detailed intuitions regarding the translation, we first comment on the main differences between $[\![\cdot]\!]_f$ and $[\![\cdot]\!]$ (cf. Fig. 4.2). In a nutshell, these differences concern concern the authentication protocol for session establishment and local information:

(1) Session establishment follows the NSL protocol. A process request starts by creating a fresh endpoint (x) and sending a tuple containing the service name a, a nonce w and the location where the requester resides (n). The tuple is encrypted using the public key of the location of the requested service (m). The requested service then creates a fresh endpoint (y) and receives an encrypted tuple, containing a, w and n. Notice that it is necessary to decrypt this

$$
\begin{aligned}
[\![[\overline{a}^m\langle x\rangle.P]^n]\!]_f &\overset{\mathsf{def}}{=} \exists w,x.\,\big(\overline{out(\epsilon;\{\langle a,w,n\rangle\}_{\mathsf{p}(m)})} \parallel \forall y\big(out(\epsilon;\mathsf{r}(n))\,\mathbf{;}\; out(\epsilon;\{\langle a,w,y,m\rangle\}_{\mathsf{p}(n)})\rightarrow \\
&\qquad \overline{out(\epsilon;\{\langle a,w,x,y\rangle\}_{\mathsf{p}(m)})} \parallel !\,\overline{\{x{:}y\}} \parallel [\![P]\!]_{f\cup\{x:y\}}\big)\big) \quad (w,y\notin \mathsf{fv}(P))\\
[\![[*\,a^{\rho}(y).P]^m]\!]_f &\overset{\mathsf{def}}{=} !\,\big(\exists y.\big(\forall z,n\big(out(\epsilon;\mathsf{r}(m))\,\mathbf{;}\; out(\epsilon;\{\langle a,z,n\rangle\}_{\mathsf{p}(m)})\otimes \mathsf{loc}_{\rho}(\epsilon;n)\rightarrow \overline{out(\epsilon;\{\langle a,z,y,m\rangle\}_{\mathsf{p}(n)})} \parallel \\
&\qquad \forall u(out(\epsilon;\mathsf{r}(m))\,\mathbf{;}\; out(\epsilon;\{\langle a,w,y,u\rangle\}_{\mathsf{p}(m)})\rightarrow [\![P]\!]_{f\cup\{y:u\}}\,)\big)\big) \quad (z,n\notin \mathsf{fv}(P))\\
[\![x\langle v.P\rangle]\!]_f &\overset{\mathsf{def}}{=} \overline{\mathsf{snd}(x;v)} \parallel \forall\epsilon(ch(x;\epsilon)\,\mathbf{;}\;\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{rcv}(f_x,v;\epsilon)\rightarrow [\![P]\!]_f)\\
[\![x(y).P]\!]_f &\overset{\mathsf{def}}{=} \forall y\big(ch(x;\epsilon)\,\mathbf{;}\;\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{snd}(f_x;y)\rightarrow \overline{\mathsf{rcv}(x,y;\epsilon)} \parallel [\![P]\!]_f\big)\\
[\![x\triangleleft l_i.P]\!]_f &\overset{\mathsf{def}}{=} \overline{\mathsf{sel}(x;l)} \parallel \forall\epsilon(ch(x;\epsilon)\,\mathbf{;}\;\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{bra}(f_x,l;\epsilon)\rightarrow [\![P]\!]_f)\\
[\![x\triangleright\{l_i{:}P_i\}_{i\in I}]\!]_f &\overset{\mathsf{def}}{=} \forall l\big(ch(x;\epsilon)\,\mathbf{;}\;\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{sel}(f_x;l)\rightarrow \overline{\mathsf{bra}(x,l;\epsilon)} \parallel \prod_{1\le i\le n} \forall\epsilon(\mathtt{tt}\,\mathbf{;}\; l=l_i\rightarrow [\![P_i]\!]_f)\big)\\
[\![v?\,(P){:}(Q)]\!]_f &\overset{\mathsf{def}}{=} \forall\epsilon\big(\mathtt{tt}\,\mathbf{;}\; v=\mathtt{tt}\rightarrow [\![P]\!]_f\,\big) \parallel \forall\epsilon\big(\mathtt{tt}\,\mathbf{;}\; v=\mathtt{ff}\rightarrow [\![Q]\!]_f\big)\\
[\![P\mid Q]\!]_f &\overset{\mathsf{def}}{=} [\![P]\!]_f \parallel [\![Q]\!]_f\\
[\![*\,x(y).P]\!]_f &\overset{\mathsf{def}}{=} ![\![x(y).P]\!]_f\\
[\![(\boldsymbol{\nu} xy)P]\!]_f &\overset{\mathsf{def}}{=} \exists x,y.\,(!\,\overline{\{x{:}y\}} \parallel [\![P]\!]_{f\cup\{x:y\}})\\
[\![\mathbf{0}]\!]_f &\overset{\mathsf{def}}{=} \overline{\mathtt{tt}}
\end{aligned}
$$

Figure 5.3: Translation from π_{E} to $\mathtt{lcc}^{\mathsf{p}}$ (cf. Def. 5.5). We use f_x to denote $f(x)$.

tuple to extract location n. This is represented by the guard being constraint $out(\epsilon; \{\langle a, z, n\rangle\}_{\mathsf{p}(m)}) \otimes \mathrm{loc}_{\rho}(\epsilon; n)$, which verifies that the location is indeed allowed to access the service, and that the encrypted tuple has the correct structure and content. Subsequently, the requested service encrypts (using the public key of n) and sends a tuple containing the nonce w, endpoints x, y and its own location (m), as well as its own service name a. Lastly, the requester receives, decodes, and sends back endpoints x, y together with the nonce w and service name a, in a tuple encoded using the public key of m to acknowledge that it has received them, thus declaring that x, y will indeed be co-variables. Notice that to unequivocally refer to a nonce, we will write nc, instead of using the conventions for variables in lcc (i.e., $x, y, w, \ldots$).

(2) Abstractions in $\mathtt{lcc}^{\mathsf{p}}$ use local information and secure patterns. Within session communications, we require the knowledge of being a channel to be private $(ch(x; \epsilon))$. This is used in conjunction with the public constraint $\{x{:}f_x\}$ to avoid interferences. Generated after session establishment, co-variable constraints are collected in the set f. This is explicit in the translations of $(\boldsymbol{\nu} xy)P$ and the session establishment constructs. In Fig. 5.3, f_x represents the co-variable of x recorded in f. Also, we assume that if $\{x{:}y\} \in f$ then $f_x = y$ and $f_y = x$.

Before presenting an illustrative example, we give some more intuitions regarding the translation itself. Below, we comment on all the translated constructs:

- The translation of the request construct $[\overline{a}^m\langle x\rangle.P]^n$ can be understood as "low-level implementation", which exposes more details about the session establishment protocol used (in this case NSL). Intuitively, the translation:

$$\begin{aligned} [\![[\overline{a}^m\langle x\rangle.P]^n]\!]_f = \exists w, x.\, \big(\overline{out(\epsilon; \{\langle a, w, n\rangle\}_{\mathsf{p}(m)})} \parallel \qquad & (p_1)\\ \forall y\big(out(\epsilon; \mathsf{r}(n))\, ;\, out(\epsilon; \{\langle a, w, y, m\rangle\}_{\mathsf{p}(n)}) \rightarrow \qquad & (p_2)\\ \overline{out(\epsilon; \{\langle a, w, x, y\rangle\}_{\mathsf{p}(m)})} \parallel !\, \overline{\{x{:}y\}} \parallel [\![P]\!]_{f \cup \{x:y\}}\big)\big) \quad & (p_3) \end{aligned}$$

can be seen as a three-part process in which the first part (p_1) corresponds to sending the require service name a, a nonce w, and the location n of the client. This correspond to the first step described above for the NSL protocol. The second part (p_2) is then awaiting for an encrypted message which will then be decrypted using the private key for location m. In the final part(p_3), an acknowledgment message is sent, as the new endpoints are created (i.e., $!\{x{:}y\}$). The condition $w, y \notin \mathsf{fv}(P)$ ensures that the variables introduced by the translation do not clash with the ones already present in the source.

- The translation of the acceptance construct, which declares a service is then as expected: the complementary of the request translation:

$$\begin{aligned} [\![[*\, a^{\rho}(y).P]^m]\!]_f = !\, \big(\exists y.\big(\forall z, n\big(out(\epsilon; \mathsf{r}(m))\, ; \qquad & \\ out(\epsilon; \{\langle a, z, n\rangle\}_{\mathsf{p}(m)}) \otimes \mathrm{loc}_{\rho}(\epsilon; n) \rightarrow \qquad & (p_1)\\ \overline{out(\epsilon; \{\langle a, z, y, m\rangle\}_{\mathsf{p}(n)})} \parallel \qquad & (p_2)\\ \forall u(out(\epsilon; \mathsf{r}(m))\, ;\, out(\epsilon; \{\langle a, w, y, u\rangle\}_{\mathsf{p}(m)}) \rightarrow \qquad & \end{aligned}$$

$$[\![P]\!]_{f\cup\{y:u\}}\,))) \qquad (p_3)$$

In the $\mathtt{lcc}^{\mathsf{p}}$ process above, the first part p_1 receives the message of a client, verifies that the location of the client is authorized and proceeds to the second part (p_2). In this part, a message containing the fresh channel endpoint y is sent, which then must be acknowledged by the client. In the final part p_3, the translation receives the channel endpoint from the client, and the service can start.

- The translations of output and input are similar to the ones in Fig. 4.2. The only difference is in the fact that channel endpoints are now restricted information (i.e., $out(x;\epsilon)$). Hence, we use function f to carry the necessary information about co-variables which must be propagated inductively in the translation.
- The translations for all the other constructs use a similar structure as in Fig. 4.2, considering the caveat regarding channels, mentioned above.

Example 5.6. Let P be the π_{E} process $\left[\overline{a}^{i_2}\langle x\rangle.P_1\right]^{i_1} \mid \left[* \, a^{\rho}(y).P_2\right]^{i_2}$, with $i_1 \in \rho$. Process P represents a service a that resides in i_1 and is requested by a client at i_2. The translation of P is as follows:

$$\begin{aligned}
[\![P]\!]_f = \exists nc_1, x.\big(&\overline{out(\epsilon;\{\langle a, nc_1, i_1\rangle\}_{\mathsf{p}(i_2)})} \parallel \\
&\qquad \forall z_1\big(out(\epsilon;\mathsf{r}(i_1))\,;\, out(\epsilon;\{\langle a, nc_1, z_1, i_2\rangle\}_{\mathsf{p}(i_1)}) \rightarrow \\
&\qquad\quad \overline{out(\epsilon;\{\langle a, nc_1, x, z_1\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}z_1\}} \parallel [\![P_1]\!]_{g_0\cup\{x:z_1\}}\big)\big) \\
\parallel !\,\exists y.\big(&\forall z_2, w\big(out(\epsilon;\mathsf{r}(i_2))\,;\, out(\epsilon;\{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_{\rho}(w;\epsilon) \rightarrow \\
&\quad \overline{out(\epsilon;\{\langle a, z_2, y, i_2\rangle\}_{\mathsf{p}(w)})} \parallel \\
&\quad \forall z_3(out(\epsilon;\mathsf{r}(i_2))\,;\, out(\{\epsilon;\langle a, z_2, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow [\![P_2]\!]_{g_0\cup\{z_3:y\}}\,)))
\end{aligned}$$

Notice that our translation captures all the steps required to implement the NSL protocol. The first τ-transition is as follows:

$$\begin{aligned}
[\![P]\!]_f \longrightarrow_1 \exists nc_1, x, y.\big(&\forall z_1\big(out(\epsilon;\mathsf{r}(i_1))\,;\, out(\epsilon;\{\langle a, nc_1, z_1, i_2\rangle\}_{\mathsf{p}(i_1)}) \rightarrow \\
&\quad \overline{out(\epsilon;\{\langle a, nc_1, x, z_1\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}z_1\}} \parallel [\![P_1]\!]_{g_0\cup\{x:z_1\}}\big) \\
&\parallel \overline{out(\epsilon;\{\langle a, nc_1, y, i_2\rangle\}_{\mathsf{p}(i_1)})} \\
&\parallel \forall z_3(out(\epsilon;\mathsf{r}(i_2))\,;\, out(\{\epsilon;\langle a, nc_1, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow \\
&\quad [\![P_2]\!]_{g_0\cup\{z_3:y\}}\,)\big) \\
\parallel !\,\exists y.\big(&\forall z_2, w\big(out(\epsilon;\mathsf{r}(i_2))\,;\, out(\epsilon;\{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_{\rho}(w;\epsilon) \rightarrow \\
&\quad \overline{out(\epsilon;\{\langle a, z_2, y, i_2\rangle\}_{\mathsf{p}(w)})} \\
&\parallel \forall z_3(out(\epsilon;\mathsf{r}(i_2))\,;\, out(\{\epsilon;\langle a, z_2, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow [\![P_2]\!]_{g_0\cup\{z_3:y\}}\,))) \\
= S_1 \qquad&
\end{aligned}$$

Observe that an encrypted tuple containing the service name a, the nonce nc_1, and the location of the requesting client is first exchanged; this transition also relies on Rule (ScL:7) (cf. Def. 2.28), to extend the scope of both the nonce being sent (nc_1) and the soon-to-be-established endpoints (x, y). Notice also that the service has been replicated. The next transition follows:

$$S_1 \longrightarrow_1 \exists nc_1, x, y.\big(\overline{out(\epsilon;\{\langle a, nc_1, x, y\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}y\}} \parallel [\![P_1]\!]_{g_0\cup\{x:y\}}$$

$$\begin{aligned}
&\qquad\qquad \parallel \forall z_3 (out(\epsilon; \mathsf{r}(i_2))\,;\, out(\{\epsilon; \langle a, nc_1, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,))\\
&\quad \parallel\, !\,\exists y. \big(\forall z_2, w \big(out(\epsilon; \mathsf{r}(i_2))\,;\, out(\epsilon; \{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_\rho(w;\epsilon) \rightarrow\\
&\qquad\qquad \overline{out(\epsilon; \{\langle a, z_2, y, i_2\rangle\}_{\mathsf{p}(w)})}\\
&\qquad\qquad \parallel \forall z_3 (out(\epsilon; \mathsf{r}(i_2))\,;\, out(\{\epsilon; \langle a, z_2, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,)))\\
&= S_2
\end{aligned}$$

Above, the requested service a sends to the client an encrypted tuple containing its name, the nonce nc_1, the endpoint name it will use (y), and its location. We now have:

$$\begin{aligned}
S_2 \longrightarrow_1\ & \exists x, y. \big(\,!\,\overline{\{x{:}y\}} \parallel [\![P_1]\!]_{g_0 \cup \{x:y\}} \parallel [\![P_2]\!]_{g_0 \cup \{x:y\}}\big)\\
& \parallel\, !\,\exists y_1. \big(\forall z_2, w \big(out(\epsilon; \mathsf{r}(i_2))\,;\, out(\epsilon; \{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_\rho(w;\epsilon) \rightarrow\\
&\qquad \overline{out(\epsilon; \{\langle a, z_2, y_1, i_2\rangle\}_{\mathsf{p}(w)})}\\
&\qquad \parallel \forall z_3 (out(\epsilon; \mathsf{r}(i_2))\,;\, out(\{\epsilon; \langle a, z_2, y_1, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow\\
&\qquad [\![P_2\{{}^{y_1}\!/_{y}\}]\!]_{g_0 \cup \{z_3:y_1\}}\,)))
\end{aligned}$$

In the final step, the client answers back by sending an encrypted tuple containing: the service name a, the nonce nc_1, and the two newly created endpoints x, y. Note that α-renaming is applied to enforce substitution on the replicated service, as in Rule $\lfloor$SEsTR$\rfloor$. △

5.2 Static Correctness

We establish the correctness of $[\![\cdot]\!]_f : \pi_{\mathsf{E}} \rightarrow \mathtt{lcc}^{\mathsf{p}}$, in the sense of Def. 2.3. We mostly build upon the approach in § 4.2. In this we address name invariance and compositionality criteria.

Theorem 5.7 (Name Invariance for $[\![\cdot]\!]_f$). *Let N be an π_{E} network. Also, let σ and x be a substitution satisfying the renaming policy for $[\![\cdot]\!]_f$ (Def. 5.5(b)) and a variable in* $\mathtt{lcc}$, *resp. Then $[\![N\sigma]\!]_f = [\![N]\!]_f\sigma'$, where $\varphi_{[\![\cdot]\!]_f}(\sigma(x)) = \sigma'(\varphi_{[\![\cdot]\!]_f}(x))$ and $\sigma = \sigma'$.*

Proof. By induction on the structure of N. Both the base case and inductive step are immediate. □

Before stating our compositionality result, we define the evaluation contexts for networks. Notice that there was no need for them to be defined before this point.

Definition 5.8 (Contexts for π_{E}). The syntax of evaluation contexts in π_{E} is given by the following grammar, where N is an π_{E} network and '$-$' represents a hole:

$$D ::= - \;\big|\; D \mid N \;\big|\; N \mid D$$

Notice that we do not consider the restriction operator as an evaluation context. This is because it is not possible to fill the hole with an arbitrary network. We now state compositionality with respect to the parallel composition operator:

Theorem 5.9 (Compositionality for $[\![\cdot]\!]_f$). *Let N be an π_{E} network. Also, let $D[-]$ be an π_{E} evaluation context (cf. Def. 5.8). Then we have: $[\![D[N]]\!]_f = [\![D]\!]_f\left[[\![N]\!]_f\right]$.*

Proof. By induction on $D[-]$ and a case analysis on N. The proof is immediate, since by Def. 5.8 evaluation contexts $D[-]$ can only be parallel contexts and the definition of $[\![\cdot]\!]_f$ is homomorphic with respect to parallel composition. □

5.3 Operational Correspondence

In this section we present the prove that $[\![\cdot]\!]_f$ is both operationally sound and complete. The work in here builds on the methodology developed in § 4.3. In § 5.3.1 we present preliminary observations required for the proof of both soundness and completeness. Finally, in § 5.3.2, and § 5.3.3 we present each proof, respectively.

5.3.1 Preliminaries

The Shape of Closed Networks

Here we state results about the shape of translated closed networks. We formalize the notion of starting and runtime networks, informally introduced earlier.

Definition 5.10 (Starting and Runtime Network). We will say that a network N is *starting* whenever $N = \mathbf{0}$ or $N = N_1 \mid \ldots \mid N_n$, $n \geq 1$ and for every $i \in \{1, \ldots, n\}$, $N_i = [\overline{a}^n\langle x\rangle.P]^m$ or $N_i = [* \, a^\rho(x).Q]^m$. Otherwise, we will call N a *runtime* network.

We now characterize the shape of closed networks (Not. 3.40). We rely on the following statement:

Lemma 5.11 (Shape of a Closed Network). *For every closed network N, we have $N \equiv_{\mathsf{S}} (\boldsymbol{\nu}\widetilde{x}\widetilde{y})P \mid M$, where M is a starting network and P is a process.*

Proof. By induction on the structure of N. The base cases are $N = \mathbf{0}$, $N = [\overline{a}^n\langle x\rangle.P]^m$, $N = [* \, a^\rho(x).Q]^m$ and $N = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})P$ and they are immediate by $\equiv_{\mathsf{S}}$. The inductive step $N = N_1 \mid N_2$ proceeds as follows:

(1) $N_1 = (\boldsymbol{\nu}\widetilde{x_1}\widetilde{y_1})P_1 \mid M_1$ (IH).

(2) $\mathsf{fv}(N_1) = \emptyset$ (Def. 5.10, (1)).

(3) $N_2 = (\boldsymbol{\nu}\widetilde{x_2}\widetilde{y_2})P_2 \mid M_2$ (IH).

(4) $\mathsf{fv}(N_2) = \emptyset$ (Def. 5.10, (3)).

(5) $N \equiv_{\mathsf{S}} ((\boldsymbol{\nu}\widetilde{x_1}\widetilde{x_2}\widetilde{y_1}\widetilde{y_2})(P_1 \mid P_2) \mid M_1 \mid M_2 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})P \mid M$, with $\widetilde{x}\widetilde{y} = \widetilde{x_1}\widetilde{x_2}\widetilde{y_1}\widetilde{y_2}$, $P = P_1 \mid P_2$ and $M = M_1 \mid M_2$ (Applying $\equiv_{\mathsf{S}}$ to N, (3),(4)).

□

The previous statement allows us to observe and work with a general shape for closed networks, which will be handy in the proof of operational soundness. Next, we introduce a notation for characterizing closed networks contextually.

Definition 5.12 (Closed Network Context). We will write $H[-_1, -_2]$ to represent a class of contexts with two holes, one for processes and one for starting networks (denoted $-_1$ and $-_2$, respectively), defined as follows:

$$H[-_1, -_2] = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})\ -_1\ \mid -_2$$

Notice that the two holes in context $H[-_1, -_2]$ represent the two "parts" of a network: one composed only of processes (denoted $-_1$) and another that will be a starting network (denoted $-_2$). We will now introduce a notation analogous to Not. 4.12, to denote the $\mathtt{lcc}^{\mathsf{p}}$ translation of H:

Notation 5.13. Let $H[-_1, -_2] = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(-_1) \mid -_2$ be a context as in Def. 5.12. We will write $[\![H_{\widetilde{x}\widetilde{y}}]\!][-_1, -_2]$ to denote the $\mathtt{lcc}^{\mathsf{p}}$ translation of H:

$$[\![H[-_1, -_2]]\!]_f \overset{\mathsf{def}}{=} \exists \widetilde{x}, \widetilde{y}. \big(\, ! \ \overline{\bigotimes_{\substack{x_i \in \widetilde{x}, \\ y_i \in \widetilde{y}}} \{x_i{:}y_i\}} \parallel -_1\big) \parallel -_2$$

Having defined a general shape for runtime networks, we now concentrate on *initialized* networks, a specific class of runtime networks from which no new sessions can be established. First, we characterize session establishment steps with a series of labeled arrows:

Notation 5.14 (Establishment Steps). Let N be a closed network.

- Whenever $N \longrightarrow_{\mathsf{N}} N'$ with Rule $\lfloor \text{SEstR} \rfloor$ (cf. Fig. 3.3) we will write $N \xrightarrow{\text{est}} N'$, to denote a so-called *establishment steps*. We write $\xrightarrow{\text{est}}{}^{*}$ to denote the reflexive-transitive closure of $\xrightarrow{\text{est}}$.
- Accordingly, we will write $N \not\xrightarrow{\text{est}}$, whenever there is not a reduction with Rule $\lfloor \text{SEstR} \rfloor$.
- Also, we will write $N \xrightarrow{\text{est}}{}^{k} N'$, $k \geq 0$ to say that N evolves into N' by establishing k sessions (i.e., Rule $\lfloor \text{SEstR} \rfloor$ has been applied k times).
- We will also write $N \xrightarrow{\neg\text{est}} N'$ whenever N reduces to N' with a rule different than Rule $\lfloor \text{SEstR} \rfloor$.

We will then prove that every closed network can establish a finite number k of sessions, bounded by the number of requests in a process. First, we define *potentially satisfied requests*:

Definition 5.15 (Potentially Satisfied Requests). Let $N = H[P, M]$ be a well-typed closed network (cf. Not. 3.40). We will say that request $N_0 = [\overline{a}^m\langle x\rangle.P]^n$ in N is *potentially satisfied* if N_0 occurs in M and if there exists a service $[* \, a^{\rho}(y).Q]^m$ in M such that $[\overline{a}^m\langle x\rangle.P]^n \mid [* \, a^{\rho}(y).Q]^m$ is a network redex (Def. 3.39).

Lemma 5.16 (Finite Session Establishments). *For every well-typed closed network N with $k \geq 0$ potentially satisfied requests (cf. Def. 5.15), there exists a sequence $N \xrightarrow{est}{}^{k} N'$, such that $N' \not\xrightarrow{est}$.*

Proof. By induction on the maximum number k of sessions that can be established in N. Notice that k will also be the number of requests in N (cf. Def. 5.15), which will be reduced whenever a session is established.

(1) $N = H[P, M]$ with P a process and M initial (By Lem. 5.11).

(2) $M = N_1 \mid \ldots \mid N_n, n \geq 1$ (By Def. 5.10 to (1)).

(3) $\forall i \in \{1, \ldots, n\}.(N_i = [\overline{a}^n\langle x\rangle.P]^m \vee N_i = [* \, a^\rho(x).Q]^m)$ (By Def. 5.10)

(4) We apply induction on the number k of potentially satisfied requests in N:

Base Case: Whenever $k = 0$. By Def. 5.15, there are no potentially satisfied requests in N and therefore, $N \overset{\text{est}}{\not\longrightarrow}$ and thus, $N' = N$.

Inductive Step: Assume that there are $k \geq 1$ potentially satisfied requests in N. Then, by IH there exists N_0 such that $N \xrightarrow{\text{est}}{}^k N_0$ and $N_0 \overset{\text{est}}{\not\longrightarrow}$. We now prove for $k+1$ potentially satisfied requests. By Def. 5.15 and (3) there exists N_i, N_j $i, j \in \{1, \ldots, n\}$ in M (i.e., $M = N_i \mid N_j \mid M'$ for some M) such that $N_i = [* \, a^\rho(y).P_1]^m$, $N_j = [\overline{a}^m\langle x\rangle.P_2]^n$ and $N_i \mid N_j$ is a network redex. We then proceed as follows:

(i) $H[P, M] = H[P, N_i \mid N_j \mid M']$ (By (1), Def. 5.15).

(ii) $H[P, M] \xrightarrow{\text{est}}{}^1 H[P \mid (\boldsymbol{\nu}xy)(P_1 \mid P_2), N_i \mid M']$ (Rule $\lfloor$SEstR$\rfloor$ to (i)).

(iii) $H[P \mid (\boldsymbol{\nu}xy)(P_1 \mid P_2), N_i \mid M']$ has $k + 1 - 1 = k$ potentially satisfied requests (By (ii)).

(iv) $H[P \mid (\boldsymbol{\nu}xy)(P_1 \mid P_2), N_i \mid M'] \xrightarrow{\text{est}}{}^k N' \overset{\text{est}}{\not\longrightarrow}$ (IH, (3)).

□

The previous lemma establishes that any well-typed closed network is either initialized, or can become initialized. This is relevant because an initialized π_{E} network behaves exactly as an π well-typed program. In fact, the following corollary (which follows from Lem. 5.16) states that the reductions of an initialized network can only originate in its process part (for all establishment steps have been already performed):

Corollary 5.17. *For every well-typed closed initialized network N:*

1. $N \overset{\text{est}}{\not\longrightarrow}$.
2. *If $N \longrightarrow_{\mathsf{N}} N'$ then $N = H[P, M]$ and $N = H[P', M]$.*

We now restrict our attention to well-typed closed networks (Not. 3.40), whose specific shape facilitates reasoning about their behavior. In particular, this will allow us to reuse definitions and results used for $[\![\cdot]\!]$ to prove both completeness and soundness. We will now proceed to establish the first invariant of translated π_{E} networks:

Lemma 5.18 (Translated Form of a Closed Network). *Let N be a well-typed closed network. Then*

$$[\![N]\!]_f = [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![R_1]\!]_{f_1} \parallel \cdots \parallel [\![R_k]\!]_{f_k}, [\![M_1]\!]_{g_1} \parallel \cdots \parallel [\![M_r]\!]_{g_r}]$$

where: (*i*) $k \geq 0$; (*ii*) *each* R_i, $0 \geq i \geq k$ *is a process pre-redex* (*cf. Def.* 3.39) *or* $R_i = \mathbf{0}$; (*iii*) *each* M_j, $1 \geq j \geq r$ *is a network pre-redex* (*cf. Def.* 3.39) *or* $M_j = \mathbf{0}$.

Proof. By induction on the structure of N. There are four base cases corresponding to $N = \mathbf{0}$, $N = (\nu xy)P$, $N = [\overline{a}^n\langle x\rangle.P]^m$ and $N = [a^\rho(x).Q]^m$. All of them are immediate, by the definition of $[\![\cdot]\!]_f$ (Fig. 5.3). The inductive step follows by IH and applying $\equiv_{\mathsf{S}}$ to the sub-processes obtained by IH. □

Junk Processes

The translation presented here, $[\![\cdot]\!]_f$, is similar to $[\![\cdot]\!]$ and so it generates junk (cf. Def. 4.14). For the sake of completeness, next we define junk processes, denoted J: the main difference with respect to the junk generated by $[\![\cdot]\!]$ is the presence of local information in π_{E} abstractions. Nevertheless, this local information is simply $\texttt{tt}$:

Definition 5.19 (Junk in π_{E}). Let P and J be $\texttt{lcc}^{\mathsf{p}}$ processes. Also, let $b \in \{\texttt{tt}, \texttt{ff}\}$ and l_i, l_j be two distinct labels. We say that J is junk, if it belongs to the following grammar:

$$J, J' ::= \forall\epsilon\big(\texttt{tt}\,;\,(b = \neg b) \rightarrow P\big) \mid \forall\epsilon\big(\texttt{tt}\,;\,(l_j = l_i) \rightarrow P\big) \mid \overline{\texttt{tt}} \mid J \parallel J'$$

We now state the main properties of junk processes induced by $[\![\cdot]\!]_f$ (and their corresponding interactions), as we did for $[\![\cdot]\!]$.

Lemma 5.20 (Junk Invariants). *Let J be a junk process. Then the following holds:*

1. $J \stackrel{\tau}{\not\longrightarrow}_1$.
2. *There is no $c \in \mathcal{S}$ (cf. Def. 5.1) such that* $J \parallel \overline{c} \xrightarrow{\tau}_1$.
3. *For every $\mathcal{D}_{\pi_{\mathsf{E}}}\mathcal{S}$-context $C[-]$:*
   (*a*) $\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{E}}}}(J) = \emptyset$ *and*
   (*b*) $\mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{E}}}}(C[J]) = \mathcal{O}^{\mathcal{D}_{\pi_{\mathsf{E}}}}(C[\overline{\texttt{tt}}])$.
4. *For every $\mathcal{D}_{\pi_{\mathsf{E}}}\mathcal{S}$-context $C[-]$, and every process P, we have* $C[P \parallel J] \approx_1^{\pi_{\mathsf{E}}} C[P]$.

Proof. We prove each item.

1. By induction on the structure of J. All cases proceed directly from the semantics of $\texttt{lcc}^{\mathsf{p}}$.
2. By induction on the structure of J. Observe that the constraint system in Def. 5.1 does not introduce any c capable of deducing the required guards for the process to execute a τ-transitions.
3. Both items proceed by induction on the structure of J.
4. By coinduction, using the same relation shown in the proof of Lem. 4.17.

□

The corollary below will allow us to remove junk, via $\cong_1^{\pi_{\mathsf{E}}}$, from both processes and from outside the scope of a restriction, at the level of networks.

Corollary 5.21. *For every junk process J (cf. Def. 5.19) and every $\texttt{lcc}^{\mathsf{p}}$ process P, $P \parallel J \cong_1^{\pi_{\mathsf{E}}} P$.*

Intermediate and Named Transitions

We now introduce intermediate redexes and characterize the kind of reductions that appear in translated networks. First, we start by defining target terms in $\mathtt{lcc}^{\mathsf{p}}$: the main difference with respect to Def. 4.7 is that instead of well-typed π processes, we will consider well-typed closed networks (cf. Not. 3.40).

Definition 5.22 (Target Terms for $[\![\cdot]\!]_f$). *Target terms* is the set of $\mathtt{lcc}^{\mathsf{p}}$ processes that is induced by the translation of well-typed closed π_{E} networks and is closed under τ-transitions:

$$\{S \mid [\![N]\!]_f \overset{\tau}{\Longrightarrow}_1 S \text{ and } \exists\Phi.(\Phi \vdash_{\mathsf{N}} N)\}$$

We shall use $S, S', \ldots$ to range over target terms.

Translating session establishment constructs in $\mathtt{lcc}^{\mathsf{p}}$ introduces further intermediate steps and processes. To capture them, we extend the intermediate redexes defined for $[\![\cdot]\!]$ (cf. Def. 4.25). To do this, we will reuse Def. 4.11 by saying that a closed runtime network $H[P, M]$ is enabled by $\widetilde{x}, \widetilde{y}$ if and only if they enable process P.

Definition 5.23 (Intermediate Redexes for $[\![\cdot]\!]_f$). Let R be a communicating process redex in π_{E} (cf. Def. 3.39) enabled by $\widetilde{x}, \widetilde{y}$ or a communicating network redex. Also, let J be as in Def. 5.19 and f be a set of co-variable constraints. The *set of intermediate* $\mathtt{lcc}^{\mathsf{p}}$ *redexes* of R, denoted $\{\![R]\!\}^{f}_{\widetilde{x}\widetilde{y}}$, is defined in Fig. 5.4.

We now introduce the analogue of Not. 4.26 for $[\![\cdot]\!]_f$. We slightly abuse notation and write $(\!|R, f|\!)^{k}_{\widetilde{x}\widetilde{y}}$ by stating that whenever R is a network redex (cf. Def. 3.39) it is enabled by any pair of vectors $\widetilde{x}, \widetilde{y}$. Notice that a network redex does not need to be enabled, as it can reduce without enclosing restrictions.

Notation 5.24. We will denote the elements of set $\{\![R]\!\}$ as $(\!|R, f|\!)^{k}_{\widetilde{x}\widetilde{y}}$, with $k \in \{1, 2, 3\}$ as in Fig. 5.5.

Finally, we define labeled transitions for $[\![\cdot]\!]_f$, as we did for $[\![\cdot]\!]$ in Def. 4.37. This will be useful to distinguish which kind of behavior is being mimicked by the translated term.

Definition 5.25 (Labeled τ-Transitions for $[\![\cdot]\!]_f$ Target Terms). Let S be a target term (cf. Def. 5.22). Also, let

$$\{\mathsf{IO}, \mathsf{SL}, \mathsf{RP}, \mathsf{CD}, \mathsf{IO}_1, \mathsf{RP}_1, \mathsf{SL}_1, \mathsf{SL}_2, \mathsf{SL}_3, \mathsf{SE}, \mathsf{SE}_1, \mathsf{SE}_2\}$$

be a set of labels, ranged over by $\alpha, \alpha_1, \alpha_2, \alpha', \ldots$. We define the labeled transition $\xrightarrow{\alpha}_1$ by the rules in Fig. 5.6.

The upper part of Fig. 5.6 contains rules which are very similar to the ones in Fig. 4.7; the lower part shows four extra rules, required to account for session establishment. Indeed, since the set of labels given in Def. 5.25 extends the set defined in Def. 4.37 with labels SE, SE_1, and SE_2, we have three corresponding rules for them (Rules $\lfloor$SE$\rfloor$, $\lfloor$SE1$\rfloor$, and $\lfloor$SE2$\rfloor$, respectively). Another new rule in Fig. 5.6 is Rule $\lfloor$COMP$\rfloor$, which enables the parallel composition of closed networks.

Notation 5.26 (Labeled Transitions). We shall use the following convenient notations:

$$
\begin{aligned}
\{\!\![x\langle v\rangle.P \mid y(z).Q]\!\!\}^{f}_{\widetilde{x}\widetilde{y}} &\overset{\mathsf{def}}{=} \{\overline{\mathrm{rcv}(y,v;\epsilon)} \parallel \forall\epsilon\big(ch(x;\epsilon)\,\mathbf{;}\,\{f_x{:}x\}\otimes ch(f_x;\epsilon)\otimes \mathrm{rcv}(f_x,v;\epsilon)\rightarrow \llbracket P\rrbracket_f\big) \parallel \llbracket Q\{v/z\}\rrbracket_f\}\\
\{\!\![x\langle v\rangle.P \mid *\,y(z).Q]\!\!\}^{f}_{\widetilde{x}\widetilde{y}} &\overset{\mathsf{def}}{=} \{\overline{\mathrm{rcv}(y,v;\epsilon)} \parallel \forall\epsilon\big(ch(x;\epsilon)\,\mathbf{;}\,\{f_x{:}x\}\otimes ch(f_x;\epsilon)\otimes \mathrm{rcv}(f_x,v;\epsilon)\rightarrow \llbracket P\rrbracket_f\big) \parallel \llbracket Q\{v/z\}\rrbracket_f \parallel \llbracket *y(w).Q\rrbracket_f\}\\
\{\!\![x\triangleleft l.P \mid y\triangleright\{l_i:Q_i\}_{i\in I}]\!\!\}^{f}_{\widetilde{x}\widetilde{y}} &\overset{\mathsf{def}}{=} \{\overline{\mathsf{bra}(y,l_j;\epsilon)} \parallel \forall\epsilon\big(ch(x;\epsilon)\,\mathbf{;}\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{bra}(f_x,l_j;\epsilon)\rightarrow \llbracket P\rrbracket_f\big) \parallel \forall\epsilon\big(\mathtt{tt}\,\mathbf{;}\,l_j=l_j\rightarrow \llbracket Q_j\rrbracket_f\big) \parallel J,\\
&\qquad \overline{\mathsf{bra}(y,l_j;\epsilon)} \parallel \forall\epsilon\big(ch(x;\epsilon)\,\mathbf{;}\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{bra}(f_x,l_j;\epsilon)\rightarrow \llbracket P\rrbracket_f\big) \parallel \llbracket Q_j\rrbracket_f \parallel J,\\
&\qquad \llbracket P\rrbracket_f \parallel \forall\epsilon(\mathtt{tt}\,\mathbf{;}\,l_j=l_j\rightarrow \llbracket Q_j\rrbracket_f) \parallel J\}\\
\{\!\![\big[\overline{a}^{i_2}\langle x\rangle.P_1\big]^{i_1} \mid \big[*\,a^{\rho}(y).P_2\big]^{i_2}]\!\!\}^{f}_{\widetilde{x}\widetilde{y}} &\overset{\mathsf{def}}{=} \{\exists nc_1,x,y.\big(\forall z_1\big(out(\epsilon;\mathsf{r}(i_1))\,\mathbf{;}\,out(\epsilon;\{\langle a,nc_1,z_1,i_2\rangle\}_{\mathsf{p}(i_1)})\rightarrow \overline{out(\epsilon;\{\langle a,nc_1,x,z_1\rangle\}_{\mathsf{p}(i_2)})} \parallel\\
&\qquad !\,\overline{\{x{:}z_1\}} \parallel \llbracket P_1\rrbracket_{f\cup\{x:z_1\}}\big) \parallel \overline{out(\epsilon;\{\langle a,nc_1,y,i_2\rangle\}_{\mathsf{p}(i_1)})} \parallel\\
&\qquad \forall z_3(out(\epsilon;\mathsf{r}(i_2))\,\mathbf{;}\,out(\{\epsilon;\langle a,nc_1,y,z_3\rangle\}_{\mathsf{p}(i_2)})\rightarrow\\
&\qquad \llbracket P_2\rrbracket_{f\cup\{z_3:y\}}\,)\big) \parallel \llbracket\big[*\,a^{\rho}(z).P_2\{z/y\}\big]^{i_2}\rrbracket_f,\\
&\qquad \exists nc_1,x,y.\big(\overline{out(\epsilon;\{\langle a,nc_1,x,y\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}y\}} \parallel \llbracket P_1\rrbracket_{f\cup\{x:y\}} \parallel\\
&\qquad \forall z_3(out(\epsilon;\mathsf{r}(i_2))\,\mathbf{;}\,out(\{\epsilon;\langle a,nc_1,y,z_3\rangle\}_{\mathsf{p}(i_2)})\rightarrow \llbracket P_2\rrbracket_{f\cup\{z_3:y\}})\big) \parallel \llbracket\big[*\,a^{\rho}(z).P_2\{z/y\}\big]^{i_2}\rrbracket_f\}
\end{aligned}
$$

Figure 5.4: Intermediate redexes for $\llbracket\cdot\rrbracket_f$ (cf. Def. 5.23)

$$
(\!|R, f|\!)^{1}_{\widetilde{x}\widetilde{y}} \stackrel{\mathsf{def}}{=}
\begin{cases}
\begin{array}{l}
\overline{\mathsf{rcv}(y, v; \epsilon)} \parallel \forall \epsilon\big(ch(x;\epsilon)\,\mathbf{;}\,\{f_x{:}x\} \otimes ch(f_x;\epsilon) \otimes \mathsf{rcv}(f_x, v; \epsilon) \to [\![P]\!]_f\big) \parallel \\
[\![Q\{v/z\}]\!]_f
\end{array} & \text{if } R = x\langle v\rangle.P \mid y(z).Q \\
\begin{array}{l}
\overline{\mathsf{rcv}(y, v; \epsilon)} \parallel \forall \epsilon\big(ch(x;\epsilon)\,\mathbf{;}\,\{f_x{:}x\} \otimes ch(f_x;\epsilon) \otimes \mathsf{rcv}(f_x, v; \epsilon) \to [\![P]\!]_f\big) \parallel \\
[\![Q\{v/z\}]\!]_f \parallel [\![*y(w).Q]\!]_f
\end{array} & \text{if } R = x\langle v\rangle.P \mid\ *y(z).Q \\
\begin{array}{l}
\overline{\mathsf{bra}(y, l_j; \epsilon)} \parallel \forall \epsilon\big(ch(x;\epsilon)\,\mathbf{;}\,\{x{:}f_x\} \otimes ch(f_x;\epsilon) \otimes \mathsf{bra}(f_x, l_j; \epsilon) \to [\![P]\!]_f\big) \parallel \\
\forall \epsilon\big(\mathtt{tt}\,\mathbf{;}\, l_j = l_j \to [\![Q_j]\!]_f\big) \parallel J
\end{array} & \text{if } R = x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i \in I} \\
\begin{array}{l}
\exists nc_1, x, y.\big(\forall z_1\big(out(\epsilon; \mathsf{r}(i_1))\,\mathbf{;}\, out(\epsilon; \{\langle a, nc_1, z_1, i_2\rangle\}_{\mathsf{p}(i_1)}) \to \\
\qquad \overline{out(\epsilon; \{\langle a, nc_1, x, z_1\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}z_1\}} \parallel [\![P_1]\!]_{f \cup \{x:z_1\}}\big) \parallel \\
\qquad \overline{out(\epsilon; \{\langle a, nc_1, y, i_2\rangle\}_{\mathsf{p}(i_1)})} \parallel \\
\qquad \forall z_3(out(\epsilon; \mathsf{r}(i_2))\,\mathbf{;}\, out(\{\epsilon; \langle a, nc_1 y, z_3\rangle\}_{\mathsf{p}(i_2)}) \to \\
\qquad\qquad [\![P_2]\!]_{f \cup \{z_3:y\}}\,)\big) \parallel [\![\,[*\,a^{\rho}(z).P_2\{z/y\}]^{i_2}]\!]_f
\end{array} & \text{if } R = [\overline{a}^{i_2}\langle x\rangle.P_1]^{i_1} \mid [*\,a^{\rho}(y).P_2]^{i_2} \\
\text{undefined,} & \text{otherwise}
\end{cases}
$$

$$
(\!|R, f|\!)^{2}_{\widetilde{x}\widetilde{y}} \stackrel{\mathsf{def}}{=}
\begin{cases}
\begin{array}{l}
\forall \epsilon\big(\underline{ch(x;\epsilon)}\,\mathbf{;}\,\{x{:}f_x\} \otimes ch(f_x;\epsilon) \otimes \mathsf{bra}(f_x, l_j; \epsilon) \to [\![P]\!]_f\big) \parallel \\
\overline{\mathsf{bra}(y, l_j; \epsilon)} \parallel [\![Q_j]\!]_f \parallel J
\end{array} & \text{if } R = x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i \in I} \\
\begin{array}{l}
\exists nc_1, x, y.\big(\overline{out(\epsilon; \{\langle a, nc_1, x, y\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}y\}} \parallel [\![P_1]\!]_{f \cup \{x:y\}} \parallel \\
\qquad \forall z_3(out(\epsilon; \mathsf{r}(i_2))\,\mathbf{;}\, out(\{\epsilon; \langle a, nc_1, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \to \\
\qquad\qquad [\![P_2]\!]_{f \cup \{z_3:y\}})\big) \parallel [\![\,[*\,a^{\rho}(z).P_2\{z/y\}]^{i_2}]\!]_f
\end{array} & \text{if } R = [\overline{a}^{i_2}\langle x\rangle.P_1]^{i_1} \mid [*\,a^{\rho}(y).P_2]^{i_2} \\
\text{undefined,} & \text{otherwise}
\end{cases}
$$

$$
(\!|R, f|\!)^{3}_{\widetilde{x}\widetilde{y}} \stackrel{\mathsf{def}}{=}
\begin{cases}
[\![P]\!]_f \parallel \forall \epsilon(\mathtt{tt}\,\mathbf{;}\, l_j = l_j \to [\![Q_j]\!]_f) \parallel J & \text{if } R = x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i \in I} \\
\text{undefined,} & \text{otherwise}
\end{cases}
$$

Figure 5.5: Notation for the intermediate redexes of $[\![\cdot]\!]_f$ (cf. Not. 5.24).

$$\lfloor\text{IO}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket x\langle v\rangle.P_1\rrbracket_f \parallel \llbracket y(z).P_2\rrbracket_f, M] \xrightarrow{\mathtt{IO}(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x\langle v\rangle.P_1 \mid y(z).P_2, f\rrparenthesis^1_{xy}, M]$$

$$\lfloor\text{RE}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket x\langle v\rangle.P_1\rrbracket_f \parallel \llbracket * y(z).P_2\rrbracket_f, M] \xrightarrow{\mathsf{RP}(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x\langle v\rangle.P_1 \mid * y(z).P_2, f\rrparenthesis^1_{xy}, M]$$

$$\lfloor\text{SL}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket x \triangleleft l_j.P_1\rrbracket_f \parallel \llbracket y \triangleright \{l_i : P_i\}_{i\in I}\rrbracket_f, M] \xrightarrow{\mathsf{SL}(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x \triangleleft l_j.P_1 \mid y \triangleright \{l_i : P_i\}_{i\in I}, f\rrparenthesis^1_{xy}, M]$$

$$\lfloor\text{CDT}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket \mathtt{tt}?\,(P_1):(P_2)\rrbracket_f, M] \xrightarrow{\mathsf{CD}(-)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket P_1\rrbracket_f \parallel \forall\epsilon\big(\mathtt{tt}\,;\, \mathtt{tt} = \mathtt{ff} \rightarrow \llbracket P_2\rrbracket_f\big), M]$$

$$\lfloor\text{CDF}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket \mathtt{ff}?\,(P_1):(P_2)\rrbracket_f, M] \xrightarrow{\mathsf{CD}(-)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket P_2\rrbracket_f \parallel \forall\epsilon\big(\mathtt{tt}\,;\, \mathtt{ff} = \mathtt{tt} \rightarrow \llbracket P_1\rrbracket_f\big), M]$$

$$\lfloor\text{IO1}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x\langle v\rangle.P_1 \mid y(z).P_2, f\rrparenthesis^1_{xy}, M] \xrightarrow{\mathtt{IO}_1(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket P_1\rrbracket_f \parallel \llbracket P_2\rrbracket_f\{v/z\}, M]$$

$$\lfloor\text{RE1}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x\langle v\rangle.P_1 \mid * y(z).P_2, f\rrparenthesis^1_{xy}, M] \xrightarrow{\mathsf{RP}_1(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket P_1\rrbracket_f \parallel \llbracket P_2\rrbracket_f\{v/z\} \parallel \llbracket * y(z).P_2\rrbracket_f, M]$$

$$\lfloor\text{SL1}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}, f\rrparenthesis^1_{xy}, M] \xrightarrow{\mathsf{SL}_1(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}, f\rrparenthesis^2_{xy}, M]$$

$$\lfloor\text{SL2}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}, f\rrparenthesis^1_{xy}, M] \xrightarrow{\mathsf{SL}_1(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}, f\rrparenthesis^3_{xy}, M]$$

$$\lfloor\text{SL3}\rfloor \; \frac{J = \prod_{i\in I\setminus\{j\}} \forall\epsilon\big(\mathtt{tt}\,;\, l_j = l_i \rightarrow \overline{\mathsf{bra}(y, l_j)} \parallel \llbracket P_i\rrbracket_f\big)}{\llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}, f\rrparenthesis^2_{xy}, M] \xrightarrow{\mathsf{SL}_2(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket P\rrbracket_f \parallel \llbracket P_j\rrbracket_f \parallel J, M]}$$

$$\lfloor\text{SL4}\rfloor \; \frac{J = \prod_{i\in I\setminus\{j\}} \forall\epsilon\big(\mathtt{tt}\,;\, l_j = l_i \rightarrow \overline{\mathsf{bra}(y, l_j)} \parallel \llbracket P_i\rrbracket_f\big)}{\llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llparenthesis x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}, f\rrparenthesis^3_{xy}, M] \xrightarrow{\mathsf{SL}_3(x,y)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[\llbracket P\rrbracket_f \parallel \llbracket P_j\rrbracket_f \parallel J, M]}$$

$$\lfloor\text{COMP}\rfloor \; \frac{\llbracket H_{\widetilde{w_1}\widetilde{z_1}}\rrbracket[P, N] \xrightarrow{\alpha}_1 \llbracket H_{\widetilde{w_2}\widetilde{z_2}}\rrbracket[P', N']}{\llbracket H_{\widetilde{x}\widetilde{w_1}\widetilde{y}\widetilde{z_1}}\rrbracket[P \parallel Q, M \parallel N] \xrightarrow{\alpha}_1 \llbracket H_{\widetilde{x}\widetilde{w_2}\widetilde{y}\widetilde{z_2}}\rrbracket[P' \parallel Q, M \parallel N']}$$

$$\lfloor\text{SE}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[P, \llbracket[\overline{a}^n\langle x\rangle.P_1]^m\rrbracket_f \parallel \llbracket[* a^\rho(y).P_2]^n\rrbracket_f] \xrightarrow{\mathsf{SE}(a)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[P, \llparenthesis[\overline{a}^n\langle x\rangle.P_1]^m \mid [* a^\rho(y).P_2]^n, f\rrparenthesis^1_{xy}]$$

$$\lfloor\text{SE1}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[P, \llparenthesis[\overline{a}^n\langle x\rangle.P_1]^m \mid [* a^\rho(y).P_2]^n, f\rrparenthesis^1_{xy}] \xrightarrow{\mathsf{SE}_1(a)}_1 \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[P, \llparenthesis[\overline{a}^n\langle x\rangle.P_1]^m \mid [* a^\rho(y).P_2]^n, f\rrparenthesis^2_{xy}]$$

$$\lfloor\text{SE2}\rfloor \; \llbracket H_{\widetilde{x}\widetilde{y}}\rrbracket[P, \llparenthesis[\overline{a}^n\langle x\rangle.P_1]^m \mid [* a^\rho(y).P_2]^n, f\rrparenthesis^2_{xy}] \xrightarrow{\mathsf{SE}_2(a)}_1 \llbracket H_{\widetilde{x}x\widetilde{y}y}\rrbracket[P \mid \llbracket P_1 \mid P_2\rrbracket_{f\cup\{x:y\}}, \llbracket[* a^\rho(y).P_2]^n\rrbracket_f]$$

Figure 5.6: Labeled transitions for translated $\mathtt{lcc}^{\mathsf{p}}$ processes (cf. Def. 5.25).

1. Given variables x, y, we will write $\alpha(x, y)$ to stand for any $\alpha \notin \{\mathsf{SE}, \mathsf{SE}_1, \mathsf{SE}_2\}$. Whenever $\alpha \in \{\mathsf{SE}, \mathsf{SE}_1, \mathsf{SE}_2\}$, we will write $\alpha(a)$, for some service name a.

2. We will write $\mathtt{CD}(-)$ to denote a conditional transition, as no variables nor service names are needed.

3. Extending Not. 4.42, we will write $\gamma(\widetilde{xy}, \widetilde{a})$ to denote a sequence of labels where $\widetilde{xy}$ is a vector of variables (representing labeled transitions related to intra-session communication) and $\widetilde{a}$ is a vector of service names (representing labeled transitions related to session establishment).

4. We will write $\gamma_1(\widetilde{x_1y_1}, \widetilde{a_1})\gamma_2(\widetilde{x_2y_2}, \widetilde{a_2})$ to denote the concatenation of two sequences of labels.

5. In writing $\gamma(\widetilde{xy}, \widetilde{a})$, vectors $\widetilde{xy}$ and $\widetilde{a}$ can be empty (denoted "$-$"): we will write $\gamma(\widetilde{xy}, -)$ to denote a sequence of labeled transitions that *does not include* actions of session establishment (SE, SE_1, SE_2). Similarly, we will write $\gamma(-, \widetilde{a})$ to denote a sequence of labeled transitions that *only includes* actions of session establishment.

The following lemma asserts that whenever the translation of an initialized network executes an action different from session establishment, then only the process part of the translated network changes.

Lemma 5.27. *For every well-typed closed initialized network $N = H[P, M]$, if*

$$[\![N]\!]_f \xRightarrow{\gamma(\widetilde{x}, \widetilde{y}, -)}_1 S$$

then $S = [\![H_{\widetilde{xy}}]\!][S', [\![M]\!]_{f \cup \{\widetilde{x}:\widetilde{y}\}}]$.

Proof. By induction on $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{x}, \widetilde{y}, -)}_1 S$ and a case analysis on label $\alpha(x, y)$ in the last transition. Since we consider a sequence without session initiation, then α is not a session establishment label (i.e., SE, SE_1, SE_2), and therefore, $[\![M]\!]_{f \cup \widetilde{xy}}$ remains the same, which follows from Def. 5.25. □

The translation of an initialized network cannot perform a labeled transition that corresponds to session establishment (i.e., $\mathsf{SE}(a)$, $\mathsf{SE}_1(a)$, $\mathsf{SE}_2(a)$).

Lemma 5.28. *If N is a well-typed, initialized and closed network then $[\![N]\!]_f \not\xrightarrow{\alpha(a)}_1$ for any service name a.*

Proof. By contradiction, we proceed as follows:

(1) N is well-typed, initialized and closed (Assumption).

(2) $[\![N]\!]_f \xrightarrow{\alpha(a)}_1$ for some service name a (Assumption).

(3) We distinguish cases depending on whether $\alpha(a) = \mathsf{SE}(a)$, $\alpha(a) = \mathsf{SE}_1(a)$ or $\alpha(a) = \mathsf{SE}_2(a)$. The latter two are vacuously true, as the translation of a network cannot yield an intermediate process. The former follows:

(i) $[\![N]\!]_f \xrightarrow{\mathsf{SE}(a)}_1$ for some service name a (Assumption)

(ii) $N = H[P, M]$ (Lem. 5.11, Def. 5.12, (1)).

(iii) $N \not\xrightarrow{\mathsf{est}}$ (Cor. 5.17, (1)).

(iv) $N \not\longrightarrow_{\mathsf{N}}$ with Rule $\lfloor \text{SE}\textsc{st}\text{R} \rceil$ (Not. 5.14, (iii)).

(v) $M \not\equiv_{\mathsf{S}} [\overline{a}^n\langle x\rangle.Q_1]^m \mid [* \, a^\rho(x).Q_2]^n \mid M'$ for some M' such that

$$[\overline{a}^n\langle x\rangle.Q_1]^m \mid [* \, a^\rho(y).Q_2]^n$$

is a network redex (Fig. 3.3, (iv)).

(vi) $[\![N]\!]_f = [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_f, [\![[\overline{a}^n\langle x\rangle.Q_1]^m]\!]_f \parallel [\![[* \, a^\rho(y).Q_2]^n]\!]_f]$ (Def. 5.25, (ii)).

(vii) $M \equiv_{\mathsf{S}} [\overline{a}^n\langle x\rangle.Q_1]^m \mid [* \, a^\rho(y).Q_2]^n$ with $[\overline{a}^n\langle x\rangle.Q_1]^m \mid [* \, a^\rho(y).Q_2]^n$ a network redex (By (vi), Fig. 5.3).

(viii) We have reached a contradiction in (vii), (iii).

□

Transforming Translated Terms Into `lcc` *via Erasure*

As hinted before, translations $[\![\cdot]\!]$ (cf. Fig. 4.2) and $[\![\cdot]\!]_f$ (cf. Fig. 5.3) are very similar, in particular when it comes to initialized networks, which can only perform actions of intra-session communication (Lem. 5.28). The *erasure function* below connects these translations: it target terms induced by $[\![\cdot]\!]_f$ into target terms induced by $[\![\cdot]\!]$.

Definition 5.29 (Erasure). The *erasure function* $\delta(\cdot) : \mathtt{lcc}^{\mathsf{p}} \to \mathtt{lcc}$ is defined inductively in Fig. 5.7.

Notice that erasure is a partial function; we will now give an example of how it works.

Example 5.30. Let us consider runtime network $N = (\boldsymbol{\nu}xy)(x\langle v\rangle.P \mid y(z).Q)$. We have:

$$\begin{aligned}[\![N]\!]_f = \exists x, y.\big(\, ! \, \overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x; v)} \parallel \\ &\forall \epsilon(ch(x;\epsilon) \,;\, \{x{:}f_x\} \otimes ch(f_x;\epsilon) \otimes \mathsf{rcv}(f_x, v; \epsilon) \to [\![P]\!]_f) \parallel \\ &\forall z\big(ch(y;\epsilon) \,;\, \{y{:}f_y\} \otimes ch(f_y;\epsilon) \otimes \mathsf{snd}(f_y; z) \to \overline{\mathsf{rcv}(y, z; \epsilon)} \parallel [\![Q]\!]_f\big)\big)\end{aligned}$$

Using the erasure function $\delta(\cdot)$ in Def. 5.29, we would obtain:

$$\begin{aligned}\delta([\![N]\!]_f) = \exists x, y.\big(\, ! \, \overline{\{x{:}y\}} \parallel \overline{\mathsf{snd}(x, v)} \parallel \forall w_1(\{x{:}f_x\} \otimes \mathsf{rcv}(w_1, v) \to \delta([\![P]\!]\{^{w_1}/_{f_x}\})) \parallel \\ \forall z, w_2\big(\{y{:}f_y\} \otimes \mathsf{snd}(w_2; z) \to \overline{\mathsf{rcv}(y, z)} \parallel \delta([\![Q]\!]\{^{w_2}/_{f_y}\})\big)\big)\end{aligned}$$

which corresponds to $[\![(\boldsymbol{\nu}xy)(x\langle v\rangle.P \mid y(z).Q)]\!]$, up to α-conversion. △

As a sanity check, we prove the correctness of $\delta(\cdot)$, in the form of an operational correspondence result:

$$\delta(\overline{c}) \overset{\mathsf{def}}{=} \begin{cases} !\,\delta(d), & \text{if } \overline{c} = !\,\overline{d} \\ \overline{\mathtt{tt}}, & \text{if } \overline{c} = \overline{\mathtt{tt}} \\ \overline{\{x{:}y\}}, & \text{if } \overline{c} = \overline{\{x{:}y\}} \\ \overline{\mathsf{snd}(x,v)}, & \text{if } \overline{c} = \overline{\mathsf{snd}(x;v)} \\ \overline{\mathsf{rcv}(x,v)}, & \text{if } \overline{c} = \overline{\mathsf{rcv}(x,v;\epsilon)} \\ \overline{\mathsf{sel}(x,l)}, & \text{if } \overline{c} = \overline{\mathsf{sel}(x;l)} \\ \overline{\mathsf{bra}(x,l)}, & \text{if } \overline{c} = \overline{\mathsf{bra}(x,l;\epsilon)} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\delta(\forall\widetilde{z}(c\,;\,d{\to}P)) \overset{\mathsf{def}}{=} \begin{cases} \forall\epsilon\big(d \to \delta(P)\big) & \text{if } c = \mathtt{tt} \\ \forall w\big(\mathsf{rcv}(w,v) \otimes \{w{:}x\} \to \\ \quad \delta(P\{{}^{w}\!/\!{}_{z}\})\big) & \text{if } c{=}ch(x;\epsilon) \otimes \{z{:}x\} \otimes \mathsf{rcv}(z,v;\epsilon) \\ \forall y,w\big(\mathsf{snd}(w,y) \otimes \{w{:}x\} \to \\ \quad \delta(P\{{}^{w}\!/\!{}_{z}\})\big) & \text{if } c{=}ch(x;\epsilon) \otimes \{z{:}x\} \otimes \mathsf{snd}(z;y) \\ \forall w\big(\mathsf{bra}(w,l) \otimes \{w{:}x\} \to \\ \quad \delta(P\{{}^{w}\!/\!{}_{z}\})\big) & \text{if } c{=}ch(x;\epsilon) \otimes \{z{:}x\} \otimes \mathsf{bra}(z,l;\epsilon) \\ \forall l,w\big(\mathsf{sel}(w,l) \otimes \{w{:}x\} \to \\ \quad \delta(P\{{}^{w}\!/\!{}_{z}\})\big) & \text{if } c{=}ch(x;\epsilon) \otimes \{z{:}x\} \otimes \mathsf{sel}(z;l) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\delta(P \parallel Q) \overset{\mathsf{def}}{=} \delta(P) \parallel \delta(Q) \quad \delta(\exists x.\big(P\big)) \overset{\mathsf{def}}{=} \exists x.\big(\delta(P)\big) \quad \delta(!\,P) \overset{\mathsf{def}}{=} !\,\delta(P)$$

Figure 5.7: Erasure function for $\mathtt{lcc}^{\mathsf{p}}$ processes (cf. Def. 5.29).

Lemma 5.31. *For every well-typed initialized closed network $N = H[P, M]$ such that*

$$P = (\boldsymbol{\nu} x_1, y_1)P_1 \mid \ldots \mid (\boldsymbol{\nu} x_n y_n)P_n$$

the following holds ($k \geq 0$):

1. *If $[\![P]\!] \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}^{k}_{1} S_1$ then $[\![P]\!]_f \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}^{k}_{1} S_2$ and $\delta(S_2) = S_1$.*
2. *If $[\![P]\!]_f \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}^{k}_{1} S$ then $[\![P]\!] \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}^{k}_{1} \delta(S)$.*

Proof. We prove each statement individually. Both cases proceed by induction on the length of the transition. For details see App. C.1. □

Finally, we show that it is possible to reduce $\xrightarrow{\neg\mathsf{est}}$ reductions (cf. Not. 5.14) to $\pi^{\lightning}_{\mathsf{OR}}$ reductions:

Lemma 5.32. *Let N be a well-typed closed network such that $N = H[P, M]$, where H is as in Def. 5.12. Then, the following holds:*

1. *If $N \xrightarrow{\neg est} M$ then $M = H[P', M]$, for some P such that $P \longrightarrow P'$.*
2. *If $P \longrightarrow P'$ then $H[P, M] \xrightarrow{\neg est} H[P', M]$.*

Proof. Immediate, since by Def. 5.12 $P = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})P'$ and by Def. 3.36, P' contains only π processes. $\square$

5.3.2 *Operational Completeness*

Having established a formal relation between $[\![\cdot]\!]_f$ and $[\![\cdot]\!]$, we now prove completeness for $[\![\cdot]\!]_f$. Intuitively, the proof proceeds by induction on the π_{E} reduction and a case analysis on the last taken step. By leveraging on Lem. 5.31, cases not related to session establishment can be reduced to Thm. 4.20.

Theorem 5.33 (Completeness for $[\![\cdot]\!]_f$). *Let $[\![\cdot]\!]_f$ be the translation in Def. 5.5. Also, let N be a well-typed closed π_{E} network . Then, if $N \longrightarrow_{\mathsf{N}}^* M$ then $[\![N]\!]_f \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{E}}} [\![M]\!]_g$, with $f \subseteq g$.*

Proof. By induction on the reduction $N \longrightarrow_{\mathsf{N}}^* M$ with a case analysis on the applied rule. The base case is immediate; for the inductive step, we assume, by IH, $N \longrightarrow_{\mathsf{N}}^* N_0 \longrightarrow_{\mathsf{N}} M$, for some N_0 and that $[\![N]\!]_f \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{E}}} [\![N_0]\!]_h$, with $h \subseteq f$. By assumption, Lem. 5.11 and Def. 5.12, $N_0 = H[P, N_0']$ for some process P and a starting network N_0'. We will then analyze two cases depending on the reduction:

$N_0 \xrightarrow{\neg\mathbf{est}} M$: We consider all the possibilities that do not correspond to session establishment. Since $[\![N_0]\!]_f = [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_{f'}, N_0']$, and since $N_0 \xrightarrow{\neg\mathbf{est}} M$, $M = H[P', N_0']$. Then, we must prove that $[\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_{f'}, N_0'] \overset{\tau}{\Longrightarrow}_1 [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P']\!]_{f'}, N_0']$. By Lem. 5.31(2) and Lem. 5.32(2), this reduces to prove that if $P \longrightarrow P'$ then $\delta([\![P]\!]_f) \overset{\tau}{\Longrightarrow}_1 \delta([\![P']\!]_f)$, which follows from Thm. 4.20.

$N_0 \xrightarrow{\mathbf{est}} M$: As follows:

Case $\lfloor\text{SE}\textsc{st}\text{R}\rfloor$: We have:

(1) $[\![N]\!]_f \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{E}}} [\![N_0]\!]_{g_0}$, $f \subseteq g_0$ (IH).

(2) $N_0 \longrightarrow_{\mathsf{N}} M$ with Rule $\lfloor\text{SE}\textsc{st}\text{R}\rfloor$ (Assumption).

(3) $N_0 = \left[\overline{a}^{i_2}\langle x\rangle.P_1\right]^{i_1} \mid \left[* \, a^{\rho}(y).P_2\right]^{i_2}$ (By (2)).

(4) $M = (\boldsymbol{\nu}xy)(P_1 \mid P_2) \mid \left[* \, a^{\rho}(y).P_2\right]^{i_2}$ (Rule $\lfloor\text{SE}\textsc{st}\text{R}\rfloor$ to (3)).

(5) By definition of $[\![\cdot]\!]$ (Fig. 5.3) and the semantics of $\mathtt{lcc}^{\mathsf{p}}$:

$$\begin{aligned}
[\![N_0]\!]_{g_0} = \exists nc_1, x.\big(&\overline{out(\epsilon; \{\langle a, nc_1, i_1\rangle\}_{\mathsf{p}(i_2)})} \parallel \\
&\forall z_1\big(out(\epsilon; \mathsf{r}(i_1))\,;\, out(\epsilon; \{\langle a, nc_1, z_1, i_2\rangle\}_{\mathsf{p}(i_1)}) \rightarrow \\
&\quad \overline{out(\epsilon; \{\langle a, nc_1, x, z_1\rangle\}_{\mathsf{p}(i_2)})} \parallel !\, \overline{\{x{:}z_1\}} \parallel \\
&\quad [\![P_1]\!]_{g_0 \cup \{x:z_1\}}\big)\big) \parallel \\
!\exists y.\big(&\forall z_2, w\big(out(\epsilon;\mathsf{r}(i_2)); out(\epsilon;\{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_{\rho}(w;\epsilon) \rightarrow \\
&\overline{out(\epsilon; \{\langle a, z_2, y, i_2\rangle\}_{\mathsf{p}(w)})} \parallel \\
&\forall z_3(out(\epsilon; \mathsf{r}(i_2))\,;\, out(\{\epsilon; \langle a, z_2, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow \\
&\quad [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,)))\\
\xrightarrow{\tau}_1 \exists nc_1, x, y.\big(&\forall z_1\big(out(\epsilon; \mathsf{r}(i_1)); out(\epsilon; \{\langle a, nc_1, z_1, i_2\rangle\}_{\mathsf{p}(i_1)}) \rightarrow
\end{aligned}$$

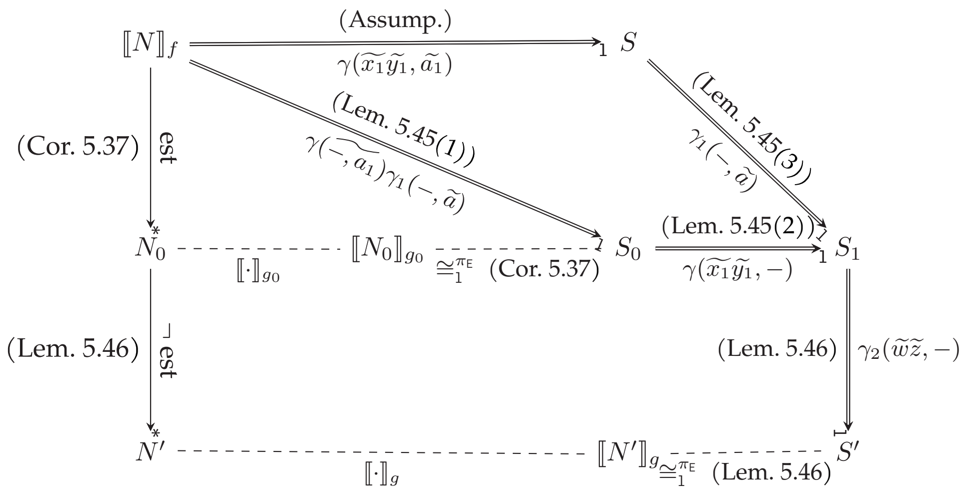
$$
\begin{aligned}
&\qquad\qquad \overline{out(\epsilon; \{\langle a, nc_1, x, z_1\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}z_1\}} \parallel \\
&\qquad\qquad [\![P_1]\!]_{g_0 \cup \{x:z_1\}}\big) \parallel \\
&\qquad\quad \overline{out(\epsilon; \{\langle a, nc_1, y, i_2\rangle\}_{\mathsf{p}(i_1)})} \parallel \\
&\qquad\quad \forall z_3(out(\epsilon; \mathsf{r}(i_2))\,;\, out(\{\epsilon; \langle a, nc_1, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow \\
&\qquad\qquad [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,)\big) \parallel \\
&\ !\,\exists y.\big(\forall z_2, w\big(out(\epsilon; \mathsf{r}(i_2))\,; \\
&\qquad\qquad out(\epsilon; \{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_\rho(w;\epsilon) \rightarrow \\
&\qquad\qquad \overline{out(\epsilon; \{\langle a, z_2, y, i_2\rangle\}_{\mathsf{p}(w)})} \parallel \\
&\qquad \forall z_3(out(\epsilon; \mathsf{r}(i_2))\,;\, out(\{\epsilon; \langle a, z_2, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow \\
&\qquad\qquad\ [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,)\big)\big) \\
\xrightarrow{\tau}_1\ & \exists nc_1, x, y.\big(\overline{out(\epsilon; \{\langle a, nc_1, x, y\rangle\}_{\mathsf{p}(i_2)})} \parallel !\,\overline{\{x{:}y\}} \parallel \\
&\qquad\quad [\![P_1]\!]_{g_0 \cup \{x:y\}} \parallel \\
&\qquad\quad \forall z_3(out(\epsilon; \mathsf{r}(i_2)); out(\{\epsilon; \langle a, nc_1, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow \\
&\qquad\qquad\ [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,)\big) \parallel \\
&\ !\,\exists y.\big(\forall z_2, w\big(out(\epsilon; \mathsf{r}(i_2))\,; \\
&\qquad\qquad out(\epsilon; \{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_\rho(w;\epsilon) \rightarrow \\
&\qquad\qquad \overline{out(\epsilon; \{\langle a, z_2, y, i_2\rangle\}_{\mathsf{p}(w)})} \parallel \\
&\qquad \forall z_3(out(\epsilon; \mathsf{r}(i_2))\,;\, out(\{\epsilon; \langle a, z_2, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow \\
&\qquad\qquad\ [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,)\big)\big) \\
\xrightarrow{\tau}_1\ & \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![P_1]\!]_{g_0 \cup \{x:y\}} \parallel [\![P_2]\!]_{g_0 \cup \{x:y\}}\big) \parallel \\
&\ !\,\exists y.\big(\forall z_2, w\big(out(\epsilon; \mathsf{r}(i_2))\,; \\
&\qquad\qquad out(\epsilon; \{\langle a, z_2, w\rangle\}_{\mathsf{p}(i_2)}) \otimes \mathrm{loc}_\rho(w;\epsilon) \rightarrow \\
&\qquad\qquad \overline{out(\epsilon; \{\langle a, z_2, y, i_2\rangle\}_{\mathsf{p}(w)})} \\
&\qquad\quad \parallel \forall z_3(out(\epsilon; \mathsf{r}(i_2)); out(\{\epsilon; \langle a, z_2, y, z_3\rangle\}_{\mathsf{p}(i_2)}) \rightarrow \\
&\qquad\qquad [\![P_2]\!]_{g_0 \cup \{z_3:y\}}\,)\big)\big) \\
\cong_1^{\pi_{\mathsf{E}}}\ & [\![(\boldsymbol{\nu} xy)(P_1 \mid P_2) \mid [* \, a^\rho(y).P_2]^{i_2}]\!]_{g_0 \cup \{x:y\}} = [\![M]\!]_{g_0 \cup \{x:y\}}
\end{aligned}
$$

(6) $g_0 \subseteq g_0 \cup \{x{:}y\}$ (Def. of $\subseteq$).

$\square$

5.3.3 *Operational Soundness*

In this section we prove that translation $[\![\cdot]\!]_f$ is operationally sound. Following the same scheme that in § 4.3.3, we first state the property, give a sketch of the proof, and then provide the details. It is important to notice that some of the results developed for proving the soundness of $[\![\cdot]\!]$ will help into simplifying some aspects of the proof for $[\![\cdot]\!]_f$. The following *labeled* soundness statement, defined in terms of the transitions introduced in Def. 5.25, will serve as an important stepping stone towards the main soundness result, given in Thm. 5.49.



$$\begin{tikzcd}
\llbracket N \rrbracket_f \arrow[rr, Rightarrow, "(\text{Assump.})", "\gamma(\widetilde{x_1 y_1}, \widetilde{a_1})"' , "1" near end] \arrow[d, "*", "(\text{Cor. } 5.37)"', "\mathsf{est}"] \arrow[rrd, Rightarrow, "(\text{Lem. } 5.45(1))", "\gamma(-,\widetilde{a_1})\gamma_1(-,\widetilde{a})"'] & & S \arrow[rd, Rightarrow, "(\text{Lem. } 5.45(3))", "\gamma_1(-,\widetilde{a})"'] & \\
N_0 \arrow[r, dash, dashed, "\llbracket \cdot \rrbracket_{g_0}"'] \arrow[d, "*", "(\text{Lem. } 5.46)"', "\neg\,\mathsf{est}"] & \llbracket N_0 \rrbracket_{g_0} \arrow[r, dash, dashed, "\cong_1^{\pi_{\mathsf{E}}}\ (\text{Cor. } 5.37)"'] & S_0 \arrow[r, Rightarrow, "(\text{Lem. } 5.45(2))", "\gamma(\widetilde{x_1 y_1}, -)"', "1" near end] & S_1 \arrow[d, Rightarrow, "(\text{Lem. } 5.46)"', "\gamma_2(\widetilde{wz}, -)"] \\
N' \arrow[rr, dash, dashed, "\llbracket \cdot \rrbracket_g"'] & & \llbracket N' \rrbracket_g \arrow[r, dash, dashed, "\cong_1^{\pi_{\mathsf{E}}}\ (\text{Lem. } 5.46)"'] & S'
\end{tikzcd}$$

Figure 5.8: Diagram of the proof of labeled soundness for $\llbracket \cdot \rrbracket_f$ (cf. Lem. 5.34).

Lemma 5.34 (Labeled Soundness for $\llbracket \cdot \rrbracket_f$). *Let $\llbracket \cdot \rrbracket_f$ be the translation in Def. 5.5. Also, let N be a well-typed closed π_{E} network. For every S such that $\llbracket N \rrbracket_f \xRightarrow{\gamma(\widetilde{x_1 y_1}, \widetilde{a_1})}_1 S$ there is an N' such that $N \longrightarrow^* N'$ and $S \xRightarrow{\gamma'(\widetilde{x_2 y_2}, \widetilde{a_2})}_1 \cong_1^{\pi_{\mathsf{E}}} \llbracket N' \rrbracket_g$, with $f \subseteq g$.*

Proof Outline for Lem. 5.34

We give a high-level description of our proof for Lem. 5.34, pointing to results to be introduced later on. First, we will show that the translation of any closed network N can reach the translation of an initialized network (cf. Def. 5.10). This is the content of Lem. 5.39 and Lem. 5.45. Then, since an initialized network is, in essence, an π process together with zero or more services that cannot be invoked anymore, we may use Lem. 5.46 to show that an initialized network satisfies operational correspondence. More in details, the proof is as depicted in Fig. 5.8, with the following steps:

(1) By assumption, $\llbracket N \rrbracket_f \xRightarrow{\gamma(\widetilde{x_1 y_1}, \widetilde{a_1})}_1 S$.

(2) Lem. 5.45(1) will ensure there is an S_0 and a transition $\llbracket N \rrbracket_f \xRightarrow{\gamma(-,\widetilde{a_1})\gamma_1(-,\widetilde{a})}_1 S_0$ such that S_0 cannot transition anymore (i.e., $S_0 \overset{\alpha(a)}{\not\longrightarrow}_1$, for any a).

(3) Cor. 5.37 ensures there is an initialized network N_0 such that $N \xrightarrow{\mathsf{est}}{}^k N_0 \overset{\mathsf{est}}{\not\longrightarrow}$, for some $k \geq 0$, and $\llbracket N_0 \rrbracket_{g_0} = S_0$, for some g_0. Notice that $\llbracket N_0 \rrbracket_{g_0} = S_0$ implies $\llbracket N_0 \rrbracket_{g_0} \cong_1^{\pi_{\mathsf{E}}} S_0$.

(4) Lem. 5.45(2) ensures there is an S_1 such that $S_0 \xRightarrow{\gamma(\widetilde{x_1 y_1}, -)}_1 S_1$ for some S_1. This sequence of transitions contains all actions in γ not related to session establishment.

(5) Given that S_0 is the translation of an initialized network, Lem. 5.46 will ensure the existence of S' and N' such that $S_1 \xRightarrow{\gamma_2(\widetilde{w}\widetilde{z},-)}_1 S'$, $N \longrightarrow^* N'$, and $[\![N']\!]_g \cong_1^{\pi_{\mathsf{E}}} S'$.

(6) Finally, we will use Lem. 5.45(3) to prove that S can reach S_1 by performing actions in $\gamma_1(-,\widetilde{a})$. This refers to all the session establishment actions that have not been yet done in S and are not contained in $\gamma(\widetilde{x_1}\widetilde{y_1},\widetilde{a_1})$.

Auxiliary Results

In this section we introduce some auxiliary results, useful for proving soundness. First, we prove that the translation of every well-typed closed network can reach a process S in which no more session establishment actions can be mimicked.

Lemma 5.35. *For every well-typed closed network N there exists S such that $[\![N]\!]_f \xRightarrow{\gamma(-,\widetilde{a})}_1 S$ and $S \not\xrightarrow{\alpha(a)}_1$ for any service name a.*

Proof. Follows from completeness (Thm. 5.33). Since N is a well-typed closed network, by Lem. 5.16 there is an N' such that $N \xrightarrow{\mathsf{est}}^* N' \not\xrightarrow{\mathsf{est}}$. Thus, by Thm. 5.33, there is an S such that $[\![N]\!]_f \xRightarrow{\gamma(-,\widetilde{a})}_1 S$ and $S \cong_1^{\pi_{\mathsf{E}}} [\![N']\!]_g$. Thus, by Def. 5.25, S will not be able to mimic further session establishment actions. □

We now prove that the translation of every closed network N can evolve into the translation of an initialized network N' reachable from N.

Lemma 5.36. *Let N be a well-typed closed network. If $[\![N]\!]_f \xRightarrow{\gamma(-,-,\widetilde{a})}_1 S$ and $S \not\xrightarrow{\alpha(a)}_1$ for any service name a, then there exists N' such that $N \xrightarrow{est}^* N' \not\xrightarrow{est}$ and $[\![N']\!]_g = S$ with $f \subseteq g$.*

Proof. We proceed by induction on the maximum number of sessions k that can be established from N. For details see App. C.2.

□

From Lem. 5.35 and Lem. 5.36, the following corollary asserts that for the translation of every well-typed closed network N there exists a source term S that will correspond to the encoding of the initialized network N' obtained from $N \xrightarrow{\mathsf{est}}^* N' \not\xrightarrow{\mathsf{est}}$.

Corollary 5.37. *For every well-typed closed network N there exist S and N' such that $[\![N]\!]_f \xRightarrow{\gamma(-,\widetilde{a})}_1 S$, $S \not\xrightarrow{\alpha(a)}_1$ (for any service name a) and $N \xrightarrow{est}^* N' \not\xrightarrow{est}$ and $[\![N']\!]_g = S$ for some g, $f \subseteq g$.*

The following results say that target terms can mimic session establishment actions independently from other actions, that is, they are not precluded by the representation of other kinds of actions.

Lemma 5.38. *Let S be a target term (Def. 5.22). If $S \xrightarrow{\alpha(a)}_1 S_1$ and $S \xRightarrow{\gamma(\widetilde{x}\widetilde{y},-)}_1 S_2$ then there exists an S_3 such that $S_1 \xRightarrow{\gamma(\widetilde{x}\widetilde{y},-)}_1 S_3$ and $S_2 \xrightarrow{\alpha(a)}_1 S_3$.*

Proof. Recalling Not. 5.26, there are three cases: (1) $\alpha(a) = \mathsf{SE}(a)$, (2) $\alpha(a) = \mathsf{SE}_1(a)$ and (3) $\alpha(a) = \mathsf{SE}_2(a)$. We proceed by induction on k, defined as the length of transition $S \xRightarrow{\gamma(\widetilde{xy},-)}_1 S_2$, together with a case analysis on the label $\alpha'(x,y)$ of the last action in the transition. For details see App. C.2. $\square$

Lemma 5.39. *Let S be a target term. If $S \xRightarrow{\gamma_1(-,\widetilde{a})}_1 S_1$ and $S \xRightarrow{\gamma_2(\widetilde{xy},-)}_1 S_2$ then there exists an S_3 such that $S_1 \xRightarrow{\gamma_2(\widetilde{xy},-)}_1 S_3$ and $S_2 \xRightarrow{\gamma_1(-,\widetilde{a})}_1 S_3$.*

Proof. By induction on $S \xRightarrow{\gamma_1(-,\widetilde{a})}_1 S_1$. The base case is immediate from Lem. 5.38. Similarly, the inductive step proceeds by applying the IH and applying Lem. 5.38 to the result. $\square$

The following two properties establish that session establishment actions can be "swapped" in the context of a sequence of labeled transitions. As a result, any sequence of labeled transitions can be rearranged so as to *promote* session establishment transitions at the beginning of the sequence.

Lemma 5.40. *Let S be a target term. If $S \xRightarrow{\gamma(\widetilde{xy},-)\alpha(a)\gamma_0(\widetilde{x_0y_0},\widetilde{a_0})}_1 S_1$ then $S \xRightarrow{\gamma_1(\widetilde{xy},-)\alpha(a)\gamma_2(\widetilde{xy},-)\gamma_0(\widetilde{x_0y_0},\widetilde{a_0})}_1 S_1$, where $\gamma_1(\widetilde{xy},-)\gamma_2(\widetilde{xy},-) = \gamma(\widetilde{xy},-)$.*

Proof. By induction on k, the length of sequence $\gamma(\widetilde{xy},-)$, coupled with a case analysis on the last label $\alpha'(x,y)$ in $\gamma(\widetilde{xy},-)$. For details see App. C.2. $\square$

Lemma 5.41. *Let N be a well-typed closed network. If $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{xy},\widetilde{a})}_1 S$ then*

$$[\![N]\!]_f \xRightarrow{\gamma(-,\widetilde{a})\gamma(\widetilde{xy},-)}_1 S$$

Proof. By induction on the length k of sequence $\widetilde{a}$.

Base Case: $k = 1$. The proof will follow directly from Lem. 5.40.

By assumption, $[\![N]\!]_f \xRightarrow{\gamma_1(\widetilde{x_1y_1},-)\alpha(a)\gamma_2(\widetilde{x_2y_2},-)}_1 S$, with

$$\gamma(\widetilde{x},\widetilde{y},-) = \gamma_1(\widetilde{x_1y_1},-)\gamma_2(\widetilde{x_2y_2},-)$$

Then, by Lem. 5.40, $[\![N]\!]_f \xRightarrow{\alpha(a)\gamma_1(\widetilde{x_1y_1},-)\gamma_2(\widetilde{x_2y_2},-)}_1 S$.

Inductive Step: $k > 1$. The IH lets us assume that if the length of $\widetilde{a}$ is $k > 1$ then we can promote session establishment actions at the beginning of the sequence (i.e., $|\widetilde{a}| = k \wedge [\![N]\!]_f \xRightarrow{\gamma(\widetilde{xy},\widetilde{a})}_1 S_0$ implies $[\![N]\!]_f \xRightarrow{\gamma(-,\widetilde{a})\gamma(\widetilde{xy},-)}_1 S_0$). We need to prove that if a new transition with $\alpha(a) = \mathsf{SE}(a)$ or $\alpha(a) = \mathsf{SE}_1(a)$ is executed, then the property holds (i.e., $[\![N]\!]_f \xRightarrow{\gamma_1(\widetilde{x_1y_1},\widetilde{a_1})\alpha(a)\gamma_2(\widetilde{x_2y_2},\widetilde{a_2})}_1 S$, $\gamma(\widetilde{xy},\widetilde{a}) = \gamma_1(\widetilde{x_1y_1},\widetilde{a_1})\gamma_2(\widetilde{x_2y_2},\widetilde{a_2})$). By IH, we can promote the sequence $\gamma_1(-,\widetilde{a_1})$, obtaining $[\![N]\!]_f \xRightarrow{\gamma_1(-,\widetilde{a_1})\gamma_1(\widetilde{x_1y_1},-)\alpha(a)\gamma_2(\widetilde{x_2y_2},\widetilde{a_2})}_1 S$. By Lem. 5.40, we can promote $\alpha(a)$, obtaining $[\![N]\!]_f \xRightarrow{\gamma_1(-,\widetilde{a_1})\alpha(a)\gamma_1(\widetilde{x_1y_1},-)\gamma_2(\widetilde{x_2y_2},\widetilde{a_2})}_1 S$ and finally, by IH, we can promote $\gamma_2(-,\widetilde{a_2})$, obtaining

$$[\![N]\!]_f \xRightarrow{\gamma_1(-,\widetilde{a_1})\alpha(a)\gamma_2(-,\widetilde{a})\gamma_1(\widetilde{x_1y_1},-)\gamma_2(\widetilde{x_2y_2},-)}_1 S$$

which is what we wanted to prove.

□

The following lemma states that if a target term S (cf. Def. 5.22) executes a sequence of session establishment actions $\gamma(-,\widetilde{a})$ that ends in an $\mathtt{lcc}^{\mathsf{p}}$ process S_1 that cannot mimic any other session establishment actions, then all the session establishment actions possible from S are already included in $\gamma(-,\widetilde{a})$.

Lemma 5.42. *Let S be a target term. If $S \xRightarrow{\gamma(-,\widetilde{a})}_1 S_1$, $S_1 \not\xrightarrow{\alpha(a)}_1$ for any a, and $S \xrightarrow{\alpha_1(a_1)}_1 S_2$, for some $S_2, \alpha_1(a_1)$, then $\alpha_1(a_1) \in \gamma(-,\widetilde{a})$.*

Proof. By contradiction. We distinguish three cases: (1) $\alpha_1(a_1) = \mathsf{SE}(a_1)$, (2) $\alpha_1(a_1) = \mathsf{SE}_1(a_1)$, and (3) $\alpha_1(a_1) = \mathsf{SE}_2(a_1)$. We only show the proof for (1), as (2) and (3) are similar:

(1) $S \xRightarrow{\gamma(-,\widetilde{a})}_1 S_1 \wedge S_1 \not\xrightarrow{\alpha(a)}_1$ (Assumption).

(2) $\exists S_2, \alpha_1(a_1).(S \xrightarrow{\mathsf{SE}_1(a_1)}_1 S_2)$ (Assumption).

(3) $\alpha_1(a_1) \notin \gamma(-,\widetilde{a})$ (Assumption).

(4) $S = [\![H_{\widetilde{x}\widetilde{y}}]\!][S_0, [\![[\overline{a_1}^n\langle x\rangle.Q_1]^m \mid [* \, a_1{}^{\rho}(x).Q_2]^n]\!]_f \parallel S_0']$ (Def. 5.25, (2)).

(5) $S_1 = [\![H_{\widetilde{x}\widetilde{y}}]\!][S_0, [\![[\overline{a_1}^n\langle x\rangle.Q_1]^m \mid [* \, a_1{}^{\rho}(x).Q_2]^n]\!]_f \parallel S_0'']$ (Def. 5.25, (1),(3),(4)).

(6) $S_1 \xrightarrow{\mathsf{SE}_1(a_1)}_1 [\![H_{\widetilde{x}\widetilde{y}}]\!][S_0, (\![[\overline{a_1}^n\langle x\rangle.Q_1]^m \mid [* \, a_1{}^{\rho}(y).Q_2]^n, f]\!)^1_{xy} \parallel S_0'']$ (Def. 5.25, (5)).

(7) We have a reached a contradiction: (6) contradicts (1).

□

The following two result imply a kind of "diamond lemma" for session establishment actions with respect to other actions. Recall that by extending Not. 4.42 we will write $\gamma(\widetilde{x}\widetilde{y},-)\setminus\alpha_i(x_j,y_j)$ to denote the sequence obtained from $\gamma(\widetilde{x}\widetilde{y},-)$ by removing $\alpha_i(x_j,y_j)$.

Lemma 5.43. *Let S be a target term. If $S \xrightarrow{\alpha(a)}_1 S_1$ and $S \xRightarrow{\gamma(-,\widetilde{a})}_1 S_2$ such that $S_2 \not\xrightarrow{\alpha(a_1)}_1$ for any service name a_1, then $S_1 \xRightarrow{\gamma(-,\widetilde{a})\setminus\alpha(a)}_1 S_2$.*

Proof. By induction on the length k of transition $S \xRightarrow{\gamma(-,\widetilde{a})}_1 S_2$. The base case is $k = 1$, which is vacuously true. In this case, the only possible $\alpha(a)$ is $\mathsf{SE}_2(a)$. This occurs because otherwise (i.e., $\alpha(a) = \mathsf{SE}(a)$ or $\alpha(a) = \mathsf{SE}_1(a)$) there would exist a sequence of transitions that allow to finish the session establishment. For the inductive step, assume $k > 1$. Thus, we distinguish cases between $\alpha(a) = \mathsf{SE}(a)$, $\alpha(a) = \mathsf{SE}_1(a)$ and $\alpha(a) = \mathsf{SE}_1(a)$. Each case concludes using Lem. 5.42 to show that $\alpha(a)$ is included in the sequence. □

Lemma 5.44. *Let S be a target term. If $S \xRightarrow{\gamma_1(-,\widetilde{a_1})}_1 S_1$ and $S \xRightarrow{\gamma_2(-,\widetilde{a_2})}_1 S_2$ such that $S_2 \not\xrightarrow{\alpha(a')}_1$ for any service name a', then $S_1 \xRightarrow{\gamma_2(-,\widetilde{a_2})\backslash\gamma_1(-,\widetilde{a_1})}_1 S_2$.*

Proof. By induction on k, the length of transition $S \xRightarrow{\gamma_1(-,\widetilde{a_1})}_1 S_1$. The base case is $k = 0$ and follows immediately. For the inductive step, we use both the IH and Lem. 5.43. □

The following lemma allows us to rearrange the transitions emanating from the translation of a well-typed closed network: we may execute first all actions concerning session establishment. This leads to the translation of an initialized network, and does not interferes with any other action.

Lemma 5.45. *Let N be a well-typed closed network. If $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{xy},\widetilde{a})}_1 S$ then there exist S_0, S_1, $\widetilde{a_1}$, and γ_1 such that*

1. *$[\![N]\!]_f \xRightarrow{\gamma(-,\widetilde{a})\gamma_1(-,\widetilde{a_1})}_1 S_0$ and $S_0 \not\xrightarrow{\alpha(a)}_1$ for any service name a.*

2. *$S_0 \xRightarrow{\gamma(\widetilde{xy},-)}_1 S_1$; and*

3. *$S \xRightarrow{\gamma_1(-,\widetilde{a_1})}_1 S_1$.*

Proof. Directly from the definitions:

(1) By Assumption, $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{xy},\widetilde{a})}_1 S$.

(2) By Lem. 5.41 and (1), $\exists S'.([\![N]\!]_f \xRightarrow{\gamma(-,a)}_1 S' \xRightarrow{\gamma(\widetilde{xy})}_1 S)$.

(3) By Cor. 5.37 and (1), $[\![N]\!]_f \xRightarrow{\gamma_0(-,\widetilde{a_0})}_1 S_0$ for some $S_0, \gamma_0(-,\widetilde{a_0})$ and $S_0 \not\xrightarrow{\alpha(a_0)}_1$ for any a_0.

(4) Applying Lem. 5.44 with $S = [\![N]\!]_f, S_1 = S', S_2 = S_0$ to (3), $[\![N]\!]_f \xRightarrow{\gamma(-,\widetilde{a})}_1 S' \xRightarrow{\gamma_1(-,\widetilde{a_1})}_1 S_0$, where $\gamma_1(-,\widetilde{a_1}) = \gamma_0(-,\widetilde{a_0}) \setminus \gamma(-,\widetilde{a})$.

(5) Applying Lem. 5.39 with $S = S'$, $S_1 = S_0$, $S_2 = S$, and $S_3 = S_1$, we have that $\exists S_1.S_0 \xRightarrow{\gamma(\widetilde{xy},-)}_1 S_1 \wedge S \xRightarrow{\gamma_1(-,\widetilde{a_1})}_1 S_1$.

□

We now prove that soundness holds for initialized networks; this follows from Lem. 5.31 and Lem. 5.32, which uses erasure (cf. Def. 5.29) to reduce the proof to the soundness property of $[\![\cdot]\!]$ (cf. Thm. 4.21):

Lemma 5.46. *Let N be a well-typed closed initialized network. If $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{x_1y_1},-)}_1 S$ then there exist N_1, S_1, and $\gamma'(\widetilde{x_2y_2},-)$ such that $N \xrightarrow{\neg est} N_1$ and $S \xRightarrow{\gamma'(\widetilde{x_2y_2},-)}_1 S_1 \cong_1^{\pi_{\mathsf{E}}} [\![N_1]\!]_g$ with $f \subseteq g$.*

Proof. Directly from the definitions, using the labels in Def. 5.25:

(1) $N \xrightarrow{\neg\mathrm{est}}$ (Cor. 5.17).

(2) $N = H[P, M], \vdash P$ (Def. 5.12, assumption).

(3) $\mathsf{SE}(a), \mathsf{SE}_1(a) \notin \gamma(\widetilde{x_1 y_1}, -)$ for any service name a (Lem. 5.28, (1)).

(4) $[\![N]\!]_f = [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_f, [\![M]\!]_f]$ ((2), Fig. 5.3).

(5) $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{x_1}\widetilde{y_1}, -)}_1 [\![H_{\widetilde{x}\widetilde{y}}]\!][S', [\![M]\!]_f]$ (Lem. 5.27).

(6) $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})P]\!] \xRightarrow{\gamma(\widetilde{x_1}\widetilde{y_1})}_1 \delta(S')$ (By Lem. 5.31(2), (5)).

(7) $\exists R, S_0.(P \longrightarrow^* R \wedge \delta(S') \xRightarrow{\gamma'(\widetilde{x_2}\widetilde{y_2})}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} S_0 \wedge R \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} \delta(S'))$ (Thm. 4.21, (2)).

(8) $\exists N_1.(N \xrightarrow{\neg\mathrm{est}} N_1 = H[R, M])$ (Lem. 5.32(2)).

(9) $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})P]\!]_f \xRightarrow{\gamma'(\widetilde{x_2}\widetilde{y_2}, -)}_1 \cong_1^{\pi_{\mathsf{E}}} S_0' \wedge \delta(S_0') = S_0$ (By Lem. 5.31(1), (7)) .

(10) $[\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_f, [\![M]\!]_f] \xRightarrow{\gamma'(\widetilde{x_2}\widetilde{y_2}, -)}_1 [\![H_{\widetilde{x}\widetilde{y}}]\!][S_0', [\![M]\!]_f] = S_1$ (Thm. 5.9, (9)).

(11) $[\![H[R, M]]\!]_f \cong_1^{\pi_{\mathsf{E}}} S_1$ (Congruence property of $\cong_1^{\pi_{\mathsf{E}}}$, (8), (10)).

□

Proof of Operational Soundness

Finally, we repeat the labeled soundness statement of Lem. 5.34 and present its proof:

Lemma 5.34 (Labeled Soundness for $[\![\cdot]\!]_f$). *Let $[\![\cdot]\!]_f$ be the translation in Def. 5.5. Also, let N be a well-typed closed π_{E} network. For every S such that $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{x_1}\widetilde{y_1}, \widetilde{a_1})}_1 S$ there is an N' such that $N \longrightarrow^* N'$ and $S \xRightarrow{\gamma'(\widetilde{x_2}\widetilde{y_2}, \widetilde{a_2})}_1 \cong_1^{\pi_{\mathsf{E}}} [\![N']\!]_g$, with $f \subseteq g$.*

Proof. We derive the proof as follows:

(1) N is a well-typed closed network (Assumption)

(2) $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{x_1}\widetilde{y_1}, \widetilde{a_1})}_1 S$ (Assumption).

(3) $\exists \gamma_1(-, \widetilde{a}), S_0.([\![N]\!]_f \xRightarrow{\gamma(-, \widetilde{a_1})\gamma_1(-, \widetilde{a})}_1 S_0)$ and $S_0 \not\xrightarrow{\mathsf{SE}(a)}_1$ nor $S_0 \not\xrightarrow{\mathsf{SE}_1(a)}_1$, for any a (Lem. 5.45 to (1), (2)).

(4) $S_0 \xRightarrow{\gamma(\widetilde{x_1}\widetilde{y_1}, -)}_1 S_1$, for some S_1 (Lem. 5.45, (1), (2)).

(5) $S \xRightarrow{\gamma_1(-, \widetilde{a})}_1 S_1$ (Lem. 5.45 to (1), (2)).

(6) There exists N_0 such that $N \xrightarrow{\mathrm{est}}^k N_0 \not\xrightarrow{\mathrm{est}}$ and $S_0 = [\![N_0]\!]_{g_0}$, for some k and $f \subseteq g_0$ (Cor. 5.37 to (1)).

(7) N_0 is a well-typed closed network (Lem. 3.38 to (6)).

(8) There exist S' and N' such that $S_0 \xRightarrow{\gamma(\widetilde{x_1}\widetilde{y_1},-)}_1 S_1$ implies that $S_1 \xRightarrow{\gamma_2(\widetilde{w}\widetilde{z},-)}_1 S'$ for some $\gamma_2(\widetilde{w}\widetilde{z},-)$, such that $N_0 \longrightarrow^* N'$ and $[\![N']\!]_g \cong_1^{\pi_{\mathsf{E}}} S'$ with $g_0 \subseteq g$ (Lem. 5.46 to (7), (4) $S_0 = [\![N_0]\!]_{g_0}$).

(9) From (9) we can conclude that $\gamma'(\widetilde{x_2}\widetilde{y_2},\widetilde{a_2}) = \gamma_1(-,\widetilde{a})\gamma_2(\widetilde{w}\widetilde{z},-)$, $S \xRightarrow{\gamma'(\widetilde{x_2}\widetilde{y_2},\widetilde{a_2})}_1 S'$ and $N \longrightarrow_{\mathsf{N}}^* N'$ where $[\![N']\!]_g \cong_1^{\pi_{\mathsf{E}}} S'$ and $f \subseteq g$ ((5),(6)).

□

While informative, labeled soundness is not yet enough to prove operational correspondence in the sense of Def. 2.6. This is because Lem. 5.34 does not use the standard semantics we assume for $\mathtt{lcc}^{\mathsf{p}}$ (i.e., τ-transitions). We now develop some auxiliary results to obtain an unlabeled version of Lem. 5.34.

The following lemma ensures that a transition in the labeled semantics $\xrightarrow{\alpha}_1$ implies a transition in the standard semantics $\xrightarrow{\tau}_1$.

Lemma 5.47. *Let S be a target term such that one of the following hold: $S \xrightarrow{\alpha(x,y)}_1 S'$, or $S \xrightarrow{\alpha(a)}_1 S'$, or $S \xrightarrow{\alpha(-)}_1 S'$. Then $S \xrightarrow{\tau}_1 S'$*

Proof. By applying a case analysis on label α. Each case proceeds by applying Rule $\lfloor$C:SyncLoc$\rfloor$, presented in § 3.3 and showing that the transition yields the correct process. □

Next, we state the converse of the previous lemma, which will allow us to prove the desired operational correspondence statement:

Lemma 5.48. *Let S be a target term such that $S \xrightarrow{\tau}_1 S'$. Then one of the following holds: (1) $S \xrightarrow{\alpha(x,y)}_1 S'$, or (2) $S \xrightarrow{\alpha(a)}_1 S'$, or (3) $S \xrightarrow{\alpha(-)}_1 S'$.*

Proof (Sketch). The proof follows closely the argument given in the one for Lem. 4.40. First, by Def. 5.22, there must exist a well-typed closed network N such that $[\![N]\!]_f \xRightarrow{\tau}_1 S$. Thus, we must show that for every $S = [\![H_{\widetilde{x}\widetilde{y}}]\!][U_1 \parallel \cdots \parallel U_n \parallel J, W_1 \parallel \cdots \parallel W_m]$ (with $n, m \geq 1$), every U_i and W_j (with $1 \leq i \leq n$ and $1 \leq j \leq m$) satisfy one of the following and J is some junk (cf. Def. 5.19):

(1) $U_i = [\![R_k]\!]_f$, where R_k is a conditional process redex reachable from N;

(2) $U_i = [\![R_k]\!]_f$, where R_k is a process pre-redex reachable from N;

(3) $U_i \in \{\![R_k \mid R_j]\!\}^f_{\widetilde{x}\widetilde{y}}$, where process redex $R_k \mid R_j$ is reachable from N;

(4) $W_i = [\![N_k]\!]_f$, where N_k is a network pre-redex reachable from N;

(5) $W_i \in \{\![N_k \mid N_r]\!\}^f_{\widetilde{x}\widetilde{y}}$, where network redex $N_k \mid N_r$ is reachable from N.

(Recall that the terminology for redexes is given in Def. 3.39.) The proof of the previous statement, similar to the one presented for Lem. 4.30, follows by induction on the length of the transition $[\![M]\!]_f \xRightarrow{\tau}_1 S$ (i.e., the transition that originated target term S from some π_{E} well-typed closed network M). Next, we prove the lemma by applying a case analysis on the arbitrary U_i and W_j that originated the transition in the inductive step. We finish the proof by showing that the transitions taken from U_i and W_j must necessarily correspond to some labeled transition. □

With the previous statement and using the labeled soundness result in Lem. 5.34, we may finally state our soundness property:

Theorem 5.49 (Soundness for $[\![\cdot]\!]_f$). *Let $[\![\cdot]\!]_f$ be the translation in Def. 5.5. Also, let N be a well-typed closed π_{E} network. For every target term S such that $[\![N]\!]_f \overset{\tau}{\Longrightarrow}_1 S$ there is an N' such that $N \longrightarrow^* N'$ and $S \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{E}}} [\![N']\!]_g$, with $f \subseteq g$.*

Proof. Since S is a target term, by Lem. 5.48, it must be the case that $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{x}\widetilde{y},\widetilde{a})}_1 S$, for some γ, $\widetilde{x}$, $\widetilde{y}$, $\widetilde{a}$. Then, by Lem. 5.34, there exists N' such that $N \longrightarrow^* N'$ and $S \xRightarrow{\gamma(\widetilde{x_1}\widetilde{x_2},\widetilde{a})}_1 \cong_1^{\pi_{\mathsf{E}}} [\![N']\!]_g$ with $f \subseteq g$. Finally, since by Lem. 5.47 every labeled sequence corresponds to a τ-transition, $S \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{E}}} [\![N']\!]_g$. □

As a consequence of the presented properties, we can conclude that $[\![\cdot]\!]_g$ is name invariant (cf.Thm. 5.7), compositional (cf. Thm. 5.9), sound (cf. Thm. 5.49), and complete (cf. Thm. 5.33). Therefore, we may state that the translation is a valid encoding (cf. Def. 2.3).

Corollary 5.50. *Translation $\langle [\![\cdot]\!]_f, \varphi_{[\![\cdot]\!]_f} \rangle$ is an encoding (Def. 2.3).*

5.4 Secure Types and the Translation

Our final correctness property for the translation is *typability* with respect to the type system in § 3.3. The goal of this property is to ensure that our translation does not allow abstractions on restricted variables, ensuring the absence of information leaks.

Theorem 5.51 (Typability of $[\![\cdot]\!]_f$). *For every well-typed closed network N, the derivation $\vdash_\diamond [\![N]\!]_f$ holds.*

Proof. By induction on the structure of N. See App. C.3 for details. □

The above theorem attests that our encoding adheres to a robust interpretation of restriction and scope extrusion, provided a disciplined used of patterns (following the signature in Fig. 5.1). By using secure patterns in our encoding $[\![\cdot]\!]_f$, we effectively limit the power of linear abstractions with local information, so as to avoid careless or malicious information leaks related to non-abstractable variables. Furthermore, we have shown that our translation does not allow information leaks related to non-abstractable variables (cf. Thm. 5.51).

We believe that $\mathtt{lcc}^{\mathsf{p}}$ and our encoding allow us to isolate and analyze the security protocols required to implement the session establishment stage. Indeed, it is possible to think of variants of the translation in Fig. 5.3 where different protocols for secure session establishment are used. This would enable us to study the different properties of these security protocols, using $\mathtt{lcc}^{\mathsf{p}}$.

6 Conclusions and Related Work

In this chapter we present the overall conclusions and related work for the two translations presented in Part II. In § 6.1 we give some concluding remarks and discuss some of the highlights of our translation; in § 6.2 we give some related work.

6.1 Concluding Remarks

We have presented two *encodings* of session π-calculi into lcc, a declarative process model based on the ccp paradigm. Our first encoding, in Ch. 4, concerns $\pi^{\lightning}_{\mathsf{OR}}$, the session π-calculus in § 3.1.1. The second encoding, given in Ch. 5, considers as source language π_{E}, a conservative extension of $\pi^{\lightning}_{\mathsf{OR}}$ with constructs for session establishment (cf. § 3.1.3); the target language is lcc$^{\mathsf{p}}$ (cf. § 3.3), the extension of lcc with abstractions with local information. Our encodings are insightful because lcc and $\pi^{\lightning}_{\mathsf{OR}}$ are very different: lcc is declarative, whereas $\pi^{\lightning}_{\mathsf{OR}}$ is operational; communication in lcc is asynchronous, based on a shared memory, whereas communication in $\pi^{\lightning}_{\mathsf{OR}}$ is point-to-point, based on message passing. Our encodings reconcile these differences, and explain precisely how to simulate the operational behavior of $\pi^{\lightning}_{\mathsf{OR}}$ (and π_{E}) using declarative features in lcc (and lcc$^{\mathsf{p}}$). Both of the encodings presented here use the same principle: they "decouple" point-to-point communication in $\pi^{\lightning}_{\mathsf{OR}}$ (and π_{E}) by exploiting synchronization on two constraints. Remarkably, because lcc treats constraints as linear, consumable resources we can correctly represent well-typed $\pi^{\lightning}_{\mathsf{OR}}$ (and π_{E}) processes that should feature linear behavior—communication actions governed by session types must occur exactly once. Thus, linearity sharply arises as the common trait in our expressiveness results. Besides the aforementioned advantages, our second encoding exhibits another contribution of our work, namely a type system for lcc$^{\mathsf{p}}$ processes that enforces secure abstractions, thus addressing an anomaly of known abstraction-based representations of scope extrusion in the π-calculus. We address the correctness of both translations via an abstract notion of encoding that follows [Gor10].

The strong correctness properties that we establish for our encodings demonstrate that lcc and $\texttt{lcc}^{\mathsf{p}}$ can both provide a unified account of operational and declarative requirements in message-passing programs. Because of the differences between lcc and $\pi^{\lightning}_{\mathsf{OR}}$, we have adopted some of the encodability criteria by Gorla in [Gor10], namely *name invariance*, *compositionality*, and *operational correspondence*. In particular, our encoding enjoys the exact same formulation of operational correspondence defined in [Gor10]. These correctness properties guarantee that the behavior of source terms is preserved and reflected appropriately by target terms.

The correctness properties of our encoding hold for $\pi^{\lightning}_{\mathsf{OR}}$ (and π_{E}) processes that are well-typed. Types not only allow us to concentrate on a meaningful class of source processes; they also allow us to address the differences between $\pi^{\lightning}_{\mathsf{OR}}$ and lcc, already mentioned. In fact, well-typed $\pi^{\lightning}_{\mathsf{OR}}$ processes have a syntactic structure that can be precisely characterized and is stable under reductions. Moreover, the compositional nature of our encoding ensures that this structure is retained by translated lcc processes (target terms) and turns out to be essential in analyzing their behavior. In this analysis, we reconstructed the behavior of source processes via the constraints that their corresponding target terms consume or add to the store during reductions. As such, this reconstruction is enabled by observables induced by the semantics of lcc. By combining information about the syntactic structure and the observable behavior of target terms, we were able to establish several invariant properties which are in turn central to prove operational correspondence, in particular soundness.

It is worth emphasizing that well-typed session π-calculus processes can contain rather liberal forms of non-deterministic behavior. In π, this is enabled by unrestricted types—see § 2.2. In contrast, lcc (and by extension $\texttt{lcc}^{\mathsf{p}}$) can represent more limited forms of non-determinism. To cope with these differences the type system for $\pi^{\lightning}_{\mathsf{OR}}$ was designed to correctly capture the kind of processes that lcc can faithfully represent. At the heart of our type system is the notion of *output race*, enabled by allowing output actions on the same channel in parallel. These actions are allowed in π (cf. § 2.2), but disallowed for source $\pi^{\lightning}_{\mathsf{OR}}$ processes. As such, our type system is a critical ingredient when proving both operational soundness and completeness. We conjecture that the machinery we have introduced here can be adapted to prove operational soundness when source processes are well-typed using the type system in § 2.2. We leave this interesting open question for follow-up work.

As application of our results and approach, we have shown how to use the first encoding (cf. Ch. 4) to represent relevant timed patterns in communication protocols, as identified by Neykova et al. [NBY14] (cf. § 1.6). Such timed patterns are commonly found in several practical applications. Hence, they serve as a valuable validation for our approach. Indeed, thanks to operational correspondence and compositionality, encodings of $\pi^{\lightning}_{\mathsf{OR}}$ processes can be used as "black boxes" whose behavior correctly mimics the source terms. These boxes can be plugged inside lcc contexts to obtain specifications that exhibit features that are not easily representable in $\pi^{\lightning}_{\mathsf{OR}}$. This way, we can analyze message-passing programs in the presence of partial and contextual information.

Finally, our second encoding features a low-level implementation of the session establishment phase of π_{E}. Although this implementation relies on the NSL authentication protocol, thanks to the way in which our completeness and soundness results were proven, it would not be difficult to "factor out" the session establishment phase so as to extend our results to translations that rely on different authentication protocols. In the same vein, we do not foresee any difficulty to extend our correctness results to a variant of π_{E} with nested sessions, which we have left out for the sake for simplicity.

6.2 Related Work

Our developments build upon the spirit of previous works by the current authors [LOP09, HL09]. However, because of the substantial technical differences (notably, the presence of linearity) our results cannot be derived from those in [LOP09] (which developed encodings of session π-calculi in utcc) nor in [HL09] (which presented a type system for utcc).

A key difference with respect to [LOP09] is the ccp language considered (lcc here, utcc in [LOP09]): this is crucial because, as already discussed, thanks to the linear abstractions in lcc, our encodings of $\pi^{\lightning}_{\mathsf{OR}}$ and π_{E}, presented in Ch. 4 and Ch. 5, are rather compact and satisfy tight operational correspondences. We also improve on expressiveness: since utcc is a deterministic calculus, the encoding in [LOP09] cannot capture non-deterministic behavior (as required for session establishment). In contrast, exploiting linearity, our encoding captures non-deterministic session establishment and the non-determinism derivable using unrestricted types in $\pi^{\lightning}_{\mathsf{OR}}$ (cf. § 3.1.1). Fig. 4.3 gives a process encodable in our approach but not in [LOP09]. We have shown that the linearity of lcc naturally matches the linear communication in $\pi^{\lightning}_{\mathsf{OR}}$. In utcc abstractions are persistent, and so the encoding in [LOP09] is more involved and its operational correspondence is harder to establish. Intuitively, representing *linear* input prefixes with *persistent* abstractions causes difficulties at several levels. Neither the anomaly of abstraction-based interpretations of scope extrusion/restriction or the use of type system for secure abstractions to limit abstraction expressiveness are addressed in [LOP09]. The type system in [HL09] (defined for utcc) and the one in § 3.3 are similar in spirit, but not in details: moving to lcc and considering linearity requires non-trivial modifications.

Haemmerlé [Hae11] gives an encoding of an asynchronous π-calculus into lcc, and establishes operational correspondence for it. Since his encoding concerns two asynchronous models, this operational correspondence is rather direct. Monjaraz and Mariño [MM12] encode the asynchronous π-calculus into Flat Guarded Horn Clauses. They consider compositionality and operational correspondence issues, as we do here. In contrast to [Hae11, MM12], here we consider a session π-calculus with synchronous communication, which adds challenges in the encoding and its associated correctness proofs. The developments in [Hae11, MM12] are not concerned with the analysis of message-passing systems in general, nor with session-based concurrency in particular.

The relationship between linear logic and session types has been recently clarified. Caires and Pfenning gave an interpretation of intuitionistic linear logic as session types, in the style of Curry-Howard [CP10]. Wadler developed this interpretation for classical linear logic [Wad14]. Giunti and Vasconcelos gave a linear reconstruction of session types [GV10]; their system is further developed in [Vas12].

Loosely related to our work are [BHTY10, CGHL10]. Bocchi et al. [BHTY10] integrate declarative requirements into *multiparty* session types by enriching (type-based) protocol descriptions with *logical assertions* which are globally specified within multiparty protocols and potentially projected onto specifications for local participants. Rather than a declarative process model based on constraints, the target process language in [BHTY10] is a π-calculus with predicates for checking (both outgoing and incoming) communications. It should be interesting to see if such an extended session π-calculus can be encoded in lcc by adapting our encoding in Ch. 4. Also in the context of choreographies, although in a different vein, Carbone et al. [CGHL10] explore reasoning via a variant of Hennessy-Milner logic for global specifications.

Several works have aimed at combining declarative and operational descriptions of services. Works on Web service contracts have been particularly successful at combining operational descriptions (akin to CCS specifications) and constraints, where the entailment of a constraint represents the possibility for a service to comply with the requirements of a requester. In [BM07b, BM11], Buscemi and Montanari develop CC-pi, a constraint language that combines the message-passing communication model from the π-calculus with operations over a constraint store as in ccp languages. Analysis techniques for CC-pi processes exploit behavioral equivalences (open bisimulation [BM08]); logical characterizations of process behavior have not been studied. A challenge for obtaining such characterizations is CC-pi's *retract* construct, which breaks the monotonicity requirements imposed for constraint stores in the ccp model. We do not know of any attempts on applying session-type analysis for specifications in CC-pi.

In a similar line of work, Coppo and Dezani-Ciancaglini [CD09] present an extension of the session π-calculus in [HVK98] with constraint handling operators, such as tells, asks and constraint checks. Session initiation is then bound to the satisfaction of constraint in the store. The merge of constraints and a session type system guarantees *bilinearity*, i.e., channels in use remain private, and that the communications proceed according to the order prescribed by the session type. It is worth noticing that the underlying constraint store in [CD09] is not linear, which can create potential races among different service providers. A linear treatment of constraints (or a process construct similar to CC-pi's retract) is left for future work.

The interplay of constraints and service contracts has been also studied by Buscemi et al. [BCDM11]. In their model, service interactions follow three phases: service negotiation, commitment and service execution. In a service negotiation phase, processes agree in fulfilling certain desired behaviors, without guarantee of success. Once committed, it is guaranteed that the execution of processes will honor promised behaviors, and forbidding a service to get stuck (deadlock-freedom). The model in [BCDM11] uses two languages: a variant of CCS is used as a source language, where the behavior of services and clients is specified; these specifications are later compiled to a target language based on CC-pi with no retraction operator, where constraints guarantee that interactions between clients and services do not deadlock. We believe that this two-level model could be enriched by the use of linear constraints similar to the ones studied in lcc and presented in this Ch. 4, thus refining the consumption of resources in the environment.

The work of Bartoletti et al. [BZ10, BTZ12] promotes contract-oriented computing as a novel vision for the runtime enforcement of service behaviors. The premise is that in scenarios where third-party components can be used but not inspected, verification based on (session) types becomes a challenge. Contracts exhibit promises about the expected runtime behavior of each component; they can be used to establish new sessions (contract negotiation) and to enforce that components abide to their promised behavior (honesty). The calculus for contracting processes is based on PCL, a propositional contract logic that includes a contractual form of implication [BZ10]; this enables to express multiparty assume-guarantee specifications where services only engage in a communication once there are enough guarantees that their requirements will be fulfilled. PCL is used as the underlying constraint system for the contract language used in [BZ10], a variant of ccp with name-passing primitives. In its accompanying Technical Report [BZ09], the authors analyze the expressive power of the contract calculus with respect to the synchronous π-calculus, establishing name-invariance, compositionality and operational correspondence, as we do here. In later developments [BTZ12] the authors introduce CO_2, a generic framework for contract-oriented computing. A characterization of contracts as processes and as formulas in PCL has been developed.

PART III

SESSION-BASED CONCURRENCY AND SYNCHRONOUS REACTIVE PROGRAMMING

7
Encoding $\pi_{\mathsf{R}}^{\lightning}$ in ReactiveML

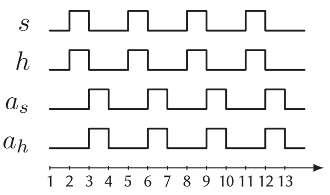
In this chapter we present a translation from $\pi_{\mathsf{R}}^{\lightning}$ (§ 3.1.2) into ReactiveML (cf. § 2.4). In § 7.1 we present the translation, denoted $[\![\cdot]\!]_f^g$. In § 7.2, show that the translation satisfies name invariance (cf. Def. 2.3(2)) and compositionality (cf. Def. 2.3(2)). In § 7.3, we show that the translation satisfies operational correspondence (cf. Def. 2.3(3,4) and Def. 2.6(3′)) and summarize our correctness results by demonstrating that the translation is both a valid and refined encoding (cf. Def. 2.3, and Def. 2.6). A highlight from this translation is the fact that it yields runnable ReactiveML code. For instance, the translation $[\![P]\!]_f^g$ of some $\pi_{\mathsf{R}}^{\lightning}$ process P can be run as a ReactiveML program by compiling the following ReactiveML expression:

$$\texttt{let process } \mathit{program} = [\![P]\!]_f^g \texttt{ in run } \mathit{program}$$

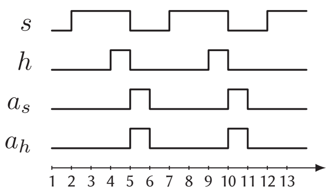
and running it in a computer. Furthermore, thanks to the correctness properties of the translation, we can guarantee that the execution of the program above preserves the behavior of the $\pi_{\mathsf{R}}^{\lightning}$ process. This chapter concludes in § 7.4 by showing examples of the timed patterns in § 1.6 represented with the translation.

7.1 The Translation

The translation from $\pi_{\mathsf{R}}^{\lightning}$ into ReactiveML uses valued signals as channels carrying messages. We also borrow inspiration from [DGS12] and use a *continuation-passing style* to preserve both the linearity and value polymorphism characteristics of π-calculus channels. In this sense, $\pi_{\mathsf{R}}^{\lightning}$ channels are translated as signals that can only be used once. Thus, these signals also carry *fresh* signals which will be used in ensuing interactions. A key difference between the translation presented here and the one in [CAP17] is that we introduce a new 'kind' of signal for synchronization purposes. In this sense, valued signals that represent channels will be called *channel signals* and signals that do not carry a value (i.e., *pure* signals) will be called *handshake signals*.



(**a**) Program *handshake*$_1$.



(**b**) Program *handshake*$_2$.

Figure 7.1: Behavior of the programs in Ex. 7.1.

Handshake signals induce a synchronization mechanism that allows for outputs to be preserved throughout several instants, until the message is received by some input. This is useful to correctly translate processes such as:

$$(\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(x\langle v_1\rangle.P_1 \mid w\langle v_2\rangle.P_2 \mid z(u_1).y(u_2).P_3)$$

Above, subprocess $w\langle v_2\rangle.P_2$ must "wait" until the outermost synchronization has occurred for its interaction to be enable. Since signal emission in RML is asynchronous, we need a mechanism that allows processes to wait until their signals have been acknowledged before executing their continuation. This is done by defining two RML programs, called *handshake processes*, that mediate the synchronization between the translated output and input processes:

$$\mathcal{H}_o(s,t,h,Q) \stackrel{\mathsf{def}}{=} \texttt{do loop (emit } s\ t\texttt{; pause) until } h \rightarrow Q \tag{7.1}$$

$$\mathcal{H}_i(s,h,t_x,R) \stackrel{\mathsf{def}}{=} \texttt{do emit } h \texttt{ when } s\texttt{; await } s(t_x) \texttt{ in } R \tag{7.2}$$

In expression $\mathcal{H}_o(s,t,h,Q)$, signal s with a tuple of values t is continuously emitted until it is detected and signal h is emitted by some other process. The emission of h triggers the continuation Q in the next instant. Analogously, process $\mathcal{H}_i(s,h,t_x,R)$ emits handshake signal h whenever s has been emitted. Notice that sub-expression `do emit` h `when` s is used to detect the signal and emit h, whereas `await` $s(t_x)$ `in` R is used to extract the tuple of values contained in s by storing it in a tuple of variables t_x. Continuation R is executed at the next instant.

Example 7.1. To better understand the intuitive behavior of expressions $\mathcal{H}_o(\cdot)$ and $\mathcal{H}_i(\cdot)$, let us consider the following two ReactiveML programs:

```
let rec process $handshake_1$ =
     (pause ; $\mathcal{H}_o(s,t,h,$emit $a_h)$) || (pause ; $\mathcal{H}_i(s,h,t_x,$emit $a_s)$);
     run $handshake_1$
 in run $handshake_1$
let rec process $handshake_2$ =
     (pause ; $\mathcal{H}_o(s,t,h,$emit $a_h)$) || (pause ; pause ; pause ; $\mathcal{H}_i(s,h,t_x,$emit $a_s)$);
     run $handshake_2$
 in run $handshake_2$
```

Whose intuitive behavior are described by the timing diagrams in Fig. 7.1. Program *handshake*$_1$ illustrates how the handshake delays the continuations until the next instant. In Fig. 7.1a, we observe that no signals are emitted in the first instant. Then, once signal s is emitted, signal h is also emitted in response. In the next time instant both a_s and a_h are emitted. The recursion continues in an infinite loop.

Program *handshake*$_2$ illustrates how $\mathcal{H}_o(\cdot)$ continues emitting s until h is present (see Fig. 7.1b). In the figure, s is emitted in instant 2 and continues to be emitted until instant 4 when signal h is detected. As with *handshake*$_1$, the execution of the continuations is delayed until the next instant. $\triangle$

The translation in [CAP17] uses a *renaming function* (ranged over by $f, f', \ldots$) to mimic the structure of session protocols. Modeling a handshaking mechanism in the translation requires adding a second renaming function (ranged over by $g, g', \ldots$). We write f_x and g_x as shorthand notations for $f(x)$ and $g(x)$.

Renaming functions must satisfy some well-formedness conditions, described below.

Definition 7.2 (Well-defined Renaming Functions). We will say that the renaming function f is *well-defined* if it satisfies the following properties:

1. $f : \mathcal{U}_s \to \mathcal{U}_r$;
2. $f(\mathtt{tt}) = \mathtt{tt}$ and $f(\mathtt{ff}) = \mathtt{ff}$.

We shall assume that renaming functions are always well-defined functions, unless otherwise stated.

Remark 7.3. Our results apply to well-typed programs (cf. Not. 2.21). This influences the definition of well-formed renaming functions: if we were to consider any process P, instead of a program, we would require both f and g to be defined for all the free variables of P. Otherwise, the translation of P would be ill-defined. Whenever we do not consider programs, we explicitly require this condition.

We now define the translation from $\pi_{\mathsf{R}}^{\lightning}$ into RML. For this we consider the formal languages defined in Def. 3.100(3) and Def. 3.102(3).

Definition 7.4 (Translating $\pi_{\mathsf{R}}^{\lightning}$ into RML). Given two well-defined renaming functions f and g, we define a translation $\mathcal{L}_{\pi_{\mathsf{R}}^{\lightning}}$ into $\mathcal{L}_{\mathsf{RML}}$, written $\langle [\![\cdot]\!]_f^g, \psi_{[\![\cdot]\!]_f^g} \rangle$, where:

1. $[\![\cdot]\!]_f^g : \pi_{\mathsf{R}}^{\lightning} \to \mathsf{RML}$ is as in Fig. 7.2.
2. $\psi_{[\![\cdot]\!]_f^g}(x) = x$, i.e., every variable in $\pi_{\mathsf{R}}^{\lightning}$ is mapped to the same variable in RML.

In writing $[\![P]\!]_f^g$ we shall assume that the renamings of f occur first than those of g.

We discuss the most interesting cases in Fig. 7.2:

- The output process $x\langle v\rangle.P$ is translated using the handshake process $\mathcal{H}_o(\cdot)$ in (7.1). Expanding its definition, we obtain:

$$\mathtt{signal}\ x', h_{x'}\ \mathtt{in}$$
$$(\mathtt{do}\ \mathtt{loop}\ (\mathtt{emit}\ f_x\ (f_v, g_v, x', h_{x'}); \mathtt{pause})\ \mathtt{until}\ g_x \to [\![P]\!]_{f,\{x \leftarrow x'\}}^{g,\{x' \leftarrow h_{x'}\}})$$

$$
\begin{aligned}
[\![x\langle v\rangle.P]\!]^g_f &\overset{\mathsf{def}}{=} \texttt{signal}\ x', h_{x'}\ \texttt{in}\ \mathcal{H}_o\big(f_x, (f_v, g_v, x', h_{x'}), g_x, [\![P]\!]^{g,\{x'\leftarrow h_{x'}\}}_{f,\{x\leftarrow x'\}}\big)\\
[\![x(y).P]\!]^g_f &\overset{\mathsf{def}}{=} \mathcal{H}_i\big(f_x, g_x, (y, h_y, w, h_w), [\![P]\!]^{g,\{w\leftarrow h_w, y\leftarrow h_y\}}_{f,\{x\leftarrow w, y\leftarrow y\}}\big)\\
[\![*\,x(y).P]\!]^g_f &\overset{\mathsf{def}}{=} \texttt{let rec process}\ \mathit{repl}\ \alpha\ \beta^g_f =\\
&\qquad\quad \texttt{do emit}\ g_\alpha\ \texttt{when}\ f_\alpha;\\
&\qquad\quad \texttt{await}\ f_\alpha(y, h_y, w, h_w)\\
&\qquad\quad \texttt{in run}\ \beta^{g,\{w\leftarrow h_w, y\leftarrow h_y\}}_{f,\{\alpha\leftarrow w, y\leftarrow y\}} \parallel \texttt{run}\ (\mathit{repl}\ \alpha\ \beta^{g,\{w\leftarrow h_w, y\leftarrow h_y\}}_{f,\{\alpha\leftarrow w, y\leftarrow y\}})\\
&\quad\ \texttt{in run}\ (\mathit{repl}\ x\ \texttt{process}\ [\![P]\!]^g_f)\\
[\![x \triangleleft l.P]\!]^g_f &\overset{\mathsf{def}}{=} \texttt{signal}\ x', h_{x'}\ \texttt{in}\ \mathcal{H}_o\big(f_x, (l, x', h_{x'}), g_x, [\![P]\!]^{g,\{x'\leftarrow h_{x'}\}}_{f,\{x\leftarrow x'\}}\big)\\
[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]^g_f &\overset{\mathsf{def}}{=} \mathcal{H}_i\big(f_x, g_x, (l, w, h_w), \texttt{match}\ l\ \texttt{with}\ l_i \to [\![P_i]\!]^{g,\{w\leftarrow h_w\}}_{f,\{x\leftarrow w\}}\big)\\
[\![v?\,(P):(Q)]\!]^g_f &\overset{\mathsf{def}}{=} \texttt{if}\ v\ \texttt{then}\ (\texttt{pause}\,;[\![P]\!]^g_f)\ \texttt{else}\ (\texttt{pause}\,;[\![Q]\!]^g_f)\\
[\![(\boldsymbol{\nu} xy)P]\!]^g_f &\overset{\mathsf{def}}{=} \texttt{signal}\ w, h_w\ \texttt{in}\ [\![P]\!]^{g,\{x\leftarrow h_w, y\leftarrow h_w\}}_{f,\{x\leftarrow w, y\leftarrow w\}}\\
[\![P \mid Q]\!]^g_f &\overset{\mathsf{def}}{=} [\![P]\!]^g_f \parallel [\![Q]\!]^g_f\\
[\![\mathbf{0}]\!]^g_f &\overset{\mathsf{def}}{=} ()
\end{aligned}
$$

Figure 7.2: Translation from $\pi_{\mathsf{R}}^{\lightning}$ to RML (Def. 7.4).

There are two fresh signals: x' is used for ensuing communications and $h_{x'}$ is used for ensuing handshakes. The translation uses a loop that constantly emits signal f_x with tuple $(f_v, g_v, x', h_{x'})$ as a value. In it, f_v and g_v denote the image of value v in each renaming function. Notice that, by Def. 7.2, f_v and g_v only become relevant when v is a channel or a variable (not a ground value). The loop keeps emitting f_x until handshake signal g_x is emitted by a corresponding input on x. At that point, the translation of P is executed by first renaming x as x' and then by assigning $h_{x'}$ as the handshake signal for x'.

- The translation of the input process $x(y).P$ goes hand-by-hand with the translation of output, using handshake process $\mathcal{H}_i(\cdot)$ (cf. (7.2)). Expanding its definition:

$$\texttt{do emit}\ g_x\ \texttt{when}\ f_x; \texttt{await}\ f_x(y, h_y, w, h_w)\ \texttt{in}\ [\![P]\!]^{g,\{w\leftarrow h_w, y\leftarrow h_y\}}_{f,\{x\leftarrow w, y\leftarrow y\}}$$

The translation waits until signal f_x is emitted by a corresponding output. Once that occurs, signal g_x is emitted to acknowledge that an input will occur in the current instant. Subsequently, the tuple (y, h_y, w, h_w) is instantiated with the parameters extracted from the tuple contained f_x: variables w and h_w bind the signals to be used in ensuing communications, whereas y and h_y bind the variables that will be substituted by f_v and g_v, respectively. The translation of the continuation is executed in the next instant using the received parameters.

- Another interesting translation is that of the replicated input $*\,x(y).P$. Observe that we use the same handshake mechanism for input. For readability purposes we expand this handshake mechanism.

```
let rec process $repl\ \alpha\ \beta_f^g =$
    do emit $g_\alpha$ when $f_\alpha$;
    await $f_\alpha(y, h_y, w, h_w)$
    in run $\beta_{f,\{\alpha\leftarrow w, y\leftarrow y\}}^{g,\{w\leftarrow h_w, y\leftarrow h_y\}}$ $\|$ run $(repl\ \alpha\ \beta_{f,\{\alpha\leftarrow w, y\leftarrow y\}}^{g,\{w\leftarrow h_w, y\leftarrow h_y\}})$
in run $(repl\ x$ process $[\![P]\!]_f^g)$
```

Recursive process *repl* receives a variable α and a process β as parameters. Variable α denotes the channel endpoint x, and β_f^g denotes the translation of the continuation P (with its respective renaming functions f and g). Intuitively, *repl* generates a copy of itself that is executed in parallel only *at the next instant*. Whenever signal f_α is detected, signal g_α is emitted, and the continuations are activated in the next instant with the necessary renamings updated in g and f.

- The translations of selection and branching are similar to the ones of output and input. The only differences are self-explanatory in Fig. 7.2. The conditional $v?\,(P):(Q)$ is translated by adding a pause before processes P and Q, aiming to preserve the main invariant of our translation: an action will be executed in a single instant. The translation of the restriction operator unifies channel endpoints on a single signal by renaming both endpoints with the same signal w. Moreover, when translating $(\boldsymbol{\nu} xy)P$, we create a handshake signal h_w to enforce the synchronization of the translations of communicating processes. Finally, the parallel composition operator is translated homomorphically. For clarity, we silently expand $\mathcal{H}_i(\cdot)$ and $\mathcal{H}_o(\cdot)$ in all the examples below.

We provide a detailed example of the application of Fig. 7.2:

Example 7.5. Let us consider the following well-typed program:

$$P_1 = (\boldsymbol{\nu} x_1 x_2)(\boldsymbol{\nu} y_1 y_2)(x_1\langle y_2\rangle.y_1\langle \mathtt{tt}\rangle.\mathbf{0} \mid x_2(u).u(u').\mathbf{0})$$

whose step-by-step translation is:

$$\begin{aligned}
[\![P_1]\!]_f^g &= [\![(\boldsymbol{\nu} x_1 x_2)(\boldsymbol{\nu} y_1 y_2)(x_1\langle y_2\rangle.y_1\langle \mathtt{tt}\rangle.\mathbf{0} \mid x_2(u).u(u').\mathbf{0})]\!]_f^g \\
&= \mathtt{signal}\ x, y, h_x, h_y\ \mathtt{in}\ [\![x_1\langle y_2\rangle.y_1\langle \mathtt{tt}\rangle.\mathbf{0} \mid x_2(u).u(u').\mathbf{0}]\!]_{f'}^{g'} && (7.3)\\
&= \mathtt{signal}\ x, y, h_x, h_y\ \mathtt{in}\ [\![x_1\langle y_2\rangle.y_1\langle \mathtt{tt}\rangle.\mathbf{0}]\!]_{f'}^{g'} \parallel [\![x_2(u).u(u').\mathbf{0}]\!]_{f'}^{g'} && (7.4)\\
&= \mathtt{signal}\ x, y, h_x, h_y\ \mathtt{in} \\
&\quad \mathtt{signal}\ x', h_{x'}\ \mathtt{in} \\
&\quad \mathtt{do}\ (\mathtt{loop}\ (\mathtt{emit}\ f'_{x_1}\ (f'_{y_2}, g'_{y_2}, x', h'_{x'}); \mathtt{pause}\,)) \\
&\quad\ \mathtt{until}\ g'_{x_1} \rightarrow [\![y_1\langle \mathtt{tt}\rangle.\mathbf{0}]\!]_{f''}^{g''} \parallel && (7.5)\\
&\quad \mathtt{do\ emit}\ g'_{x_2}\ \mathtt{when}\ f'_{x_2}; \mathtt{await}\ f'_{x_2}(u, h_u, z, h_z)\ \mathtt{in}\ [\![u(u').\mathbf{0})]\!]_{\hat{f}}^{\hat{g}} \\
&= \mathtt{signal}\ x, y, h_x, h_y\ \mathtt{in} \\
&\quad \mathtt{signal}\ x', h_{x'}\ \mathtt{in} \\
&\quad \mathtt{do\ loop}\ (\mathtt{emit}\ f'_{x_1}\ (f'_{y_2}, g'_{y_2}, x', h_{x'}); \mathtt{pause}\,)
\end{aligned}$$

$$\begin{aligned}
&\texttt{until } g'_{x_1} \rightarrow \texttt{signal } y', h_{y'} \\
&\qquad\qquad \texttt{in do loop (emit } f''_{y_1}\ (f''_{\texttt{tt}}, g''_{\texttt{tt}}, y', h_{y'}); \texttt{pause}) \\
&\texttt{until } g''_{y_1} \rightarrow () \parallel \\
&\texttt{do emit } g'_{x_2} \texttt{ when } f'_{x_2}; \texttt{await } f'_{x_2}(u, h_u, z, h_z) \texttt{ in} \\
&\texttt{do emit } \hat{g}_u \texttt{ when } \hat{f}_u; \texttt{await } \hat{f}_u(u', h_{u'}, z', h_{z'}) \texttt{ in } ()
\end{aligned} \tag{7.6}$$

We now show how the renaming functions are initialized and updated in each translation step:

(1) Since $\mathsf{fv}_\pi(P_1) = \emptyset$, we let $f = \{(\texttt{tt} \mapsto \texttt{tt}), (\texttt{ff} \mapsto \texttt{ff})\}$ and $g = \{(\texttt{tt} \mapsto \texttt{tt}), (\texttt{ff} \mapsto \texttt{ff})\}$.

(2) We declare fresh signals x, y, h_x, h_y. Briefly, signal x "unifies" endpoints x_1 and x_2, and signal y unifies y_1 and y_2. Signals h_x and h_y are the respective handshake signals for x and y. Then, we update functions f and g accordingly: $f' = f, \{x_1 \leftarrow x, x_2 \leftarrow x, y_1 \leftarrow y, y_2 \leftarrow y\}$ and $g' = \{x_1 \leftarrow h_x, x_2 \leftarrow h_x, y_1 \leftarrow h_y, y_2 \leftarrow h_y\}$.

(3) We translate the parallel composition operator '$\mid$'. Notice that f' and g' are carried unchanged in both parallel processes.

(4) Each parallel process updates the functions individually. Thus, we will have that:
$$f'' = f', \{x \leftarrow x'\} \qquad g'' = g', \{x' \leftarrow h_{x'}\}$$
$$\hat{f} = f', \{y \leftarrow z, u \leftarrow u\} \qquad \hat{g} = g', \{z \leftarrow h_z, u \leftarrow h_u\}$$

Finally, we show the execution of the translated process, using the semantics of RML:

$$\begin{aligned}
[\![P_1]\!]^g_f \longmapsto\ &\texttt{signal } y', h_{y'} \texttt{ in do (loop (emit } y\ (\texttt{tt}, \texttt{tt}, y', h_{y'}); \texttt{pause})) \texttt{ until } h_y \rightarrow () \parallel \\
&\texttt{do emit } h_y \texttt{ when } y; \texttt{await } y(u', h_{u'}, z', h_{z'}) \texttt{ in } () \\
\longmapsto\ &() \parallel ()
\end{aligned}$$

△

Remark 7.6. Notice that the well-definedness condition on renaming functions (cf. Def. 7.2) is preserved during the translation. Indeed, by definition, the translation only adds new pairs or updates the image of a variable in the domain of the renaming function.

In the following example, we analyze the semantics of the RML expressions generated by $[\![\cdot]\!]^g_f$ (cf. Fig. 7.2). We also use this example to show how signal declarations disappear during a RML execution.

Example 7.7. Consider the following well-typed process:

$$P_2 = (\boldsymbol{\nu} x_1 x_2)(\boldsymbol{\nu} y_1 y_2)(x_1\langle \texttt{tt}\rangle.\mathbf{0} \mid x_2(z_1).y_2(z_2).\mathbf{0} \mid y_1\langle \texttt{tt}\rangle.\mathbf{0}) \tag{7.7}$$

and consider the translation in Fig. 7.2:

$$[\![P_2]\!]^g_f = \texttt{signal } x, y, h_x, h_y \texttt{ in}$$

$$\begin{aligned}
&\texttt{signal}\ x', h_{x'}\ \texttt{in do loop}\ (\texttt{emit}\ x\ (\texttt{tt}, \texttt{tt}, x', h_{x'}); \texttt{pause})\ \texttt{until}\ h_x \to ()\ \|\\
&[\![x_2(z_1).y_2(z_2).\mathbf{0}]\!]^{g'}_{f'}\ \|\\
&\texttt{signal}\ y', h_{y'}\ \texttt{in do loop}\ (\texttt{emit}\ y\ (\texttt{tt}, \texttt{tt}, y', h_{y'}); \texttt{pause})\ \texttt{until}\ h_y \to ()\\
\longmapsto\ &()\ \|\ [\![y_2(z_2).\mathbf{0}]\!]^{g''}_{f''}\ \|\ \texttt{do loop}\ (\texttt{emit}\ y\ (\texttt{tt}, \texttt{tt}, y', h_{y'}); \texttt{pause})\ \texttt{until}\ h_y \to ()\\
\longmapsto\ &()\ \|\ ()\ \|\ ()
\end{aligned}$$

where f and g only contain the identity mapping for tt and ff and where $f' = f, \{x_1 \leftarrow x, x_2 \leftarrow x, y_1 \leftarrow y, y_2 \leftarrow y\}$, and $g' = g, \{x_1 \leftarrow h_x, x_2 \leftarrow h_x, y_1 \leftarrow h_y, y_2 \leftarrow h_y\}$. Notice that we do not expand f'' and g'': they can be derived by following the definitions.

Notice that every big-step reduction of RML in the translation reaches a state which has the same 'capabilities' as the source process (i.e., two synchronizations can be executed). This invariant is preserved due to the handshake signals, which disallow signal emissions from output translations to be lost. Observe that the one of the parallel subprocesses in the big-step reduction sequence above corresponds to:

$$S = \texttt{do loop}\ (\texttt{emit}\ y_1\ (\texttt{tt}, \texttt{tt}, y_1', h_{y_1'}); \texttt{pause})\ \texttt{until}\ h_{y_1} \to ()$$

By inspection, it is possible to see that S corresponds to the translation of an output operator (sans the required signal declarations). Indeed, although S has the same capabilities as the translation of an output, it is not syntactically equal to such translation. This syntactic difference is introduced by the semantics of RML. This is a key insight for proving operational correspondence, as we will need to account for the missing signal declarations. $\triangle$

7.2 Static Correctness

Name invariance (cf. Def. 2.3(2)) states that an encoding should preserve substitutions. To prove this property in $[\![\cdot]\!]^f_g$, we consider the fact that substitutions in $\pi_{\mathsf{R}}^{\lightning}$ operate over variables. Therefore, both channels and values may be substituted. Since renaming function also substitutes variables in the translation, they should somehow be reflected in the name invariance statement. We formalize this by introducing a function composition $\odot$ that merges renaming functions and substitutions. Intuitively, $\odot$ merges a renaming function and a substitution by adding the pairs corresponding to the variables to be substituted, mapped to their intended image to the renaming functions. After presenting this definition, we state our name invariance property.

Definition 7.8 (Composition of Renaming Functions). Let $f : \mathcal{U}_s \to \mathcal{U}_r$ and $\sigma : \mathcal{V}_s \to \mathcal{U}_s$ be a substitution function in $\pi_{\mathsf{R}}^{\lightning}$. We define a composition operator $\odot$ such that

$$\begin{aligned}
f \odot \sigma =& \{(u \mapsto u') \mid \forall x \in (dom(f) \cap dom(\sigma)).(u = \sigma(x) \wedge u' = f(x))\} \cup \\
& \{(u \mapsto u') \mid \forall (u \mapsto u') \in f.(u \notin dom(\sigma))\}
\end{aligned}$$

We extend this composition operator to renaming function g as expected.

Theorem 7.9 (Name Invariance for $[\![\cdot]\!]^g_f$). *For every $\pi_{\mathsf{R}}^{\lightning}$ process P, substitution σ, and any f and g such that $\mathsf{fv}_\pi(P) \subseteq dom(f)$ and $\mathsf{fv}_\pi(P) \subseteq dom(g)$, it holds that $[\![P\sigma]\!]^{g\odot\sigma}_{f\odot\sigma} = [\![P]\!]^g_f\sigma'$ for some σ'.*

Proof. By induction on the structure of P. The proof has one base case and eight inductive cases. The base case is whenever $P = \mathbf{0}$ and is immediate. Moreover, since all the inductive cases are similar, we only show $P = x\langle v\rangle Q$. We distinguish four sub-cases that depend on $x, v \in dom(\sigma)$: (1) $x, v \in dom(\sigma)$, (2) $x, v \notin dom(\sigma)$, (3) $x \in dom(\sigma), v \notin dom(\sigma)$, (4) $x \notin dom(\sigma), v \in dom(\sigma)$. We prove the first one as the other are similar. First, assume that $\sigma(x) = \dot{x}$ and $\sigma(v) = \dot{v}$:

(1)	$dom(f) = dom(g) \supseteq \mathsf{fv}_\pi(P)$	(Assumption)
(2)	$f(\mathtt{tt}) = \mathtt{tt}$ and $f(\mathtt{ff}) = \mathtt{ff}$	(Assumption)
(3)	$x\langle v\rangle.Q\sigma = \dot{x}\langle\dot{v}\rangle.(Q\sigma)$	(Def. of substitution)
(4)	$(\dot{x} \mapsto f_x), (\dot{v} \mapsto f_v) \in f \odot \sigma$	(Def. 7.8)
(5)	$(\dot{x} \mapsto g_x), (\dot{v} \mapsto g_v) \in g \odot \sigma$	(Def. 7.8)
(6)	$[\![P\sigma]\!]^{f\odot\sigma}_{g\odot\sigma} = \mathtt{signal}\ x', c'\ \mathtt{in}$ $\mathtt{do\ loop}\ (\mathtt{emit}\ f_{\dot{x}}\ (f_{\dot{v}}, g_{\dot{v}}, x', c'); \mathtt{pause})$ $\mathtt{until}\ g_{\dot{x}} \to [\![Q\sigma]\!]^{g\odot\sigma,\{x'\leftarrow c'\}}_{f\odot\sigma,\{\dot{x}\leftarrow x'\}}$	(Fig. 7.2 to (3), (1))
(7)	$[\![P]\!]^f_g = \mathtt{signal}\ x', y'\ \mathtt{in}$ $\mathtt{do\ loop}\ (\mathtt{emit}\ f_x\ (f_v, g_v, x_1, y_1); \mathtt{pause})\ \mathtt{until}\ g_x$ $\to [\![Q]\!]^{g,\{x\leftarrow x'\}}_{f,\{x\leftarrow x'\}}$	(Fig. 7.2, (3), (1))
(8)	$f \odot \sigma, \{x \leftarrow x'\} = f, \{x \leftarrow x'\} \odot \sigma$	(Def. 7.8, (4))
(9)	$g \odot \sigma, \{x' \leftarrow c'\} = g, \{x' \leftarrow c'\} \odot \sigma$	(Def. 7.8, (5))
(10)	$\exists\sigma'.([\![Q]\!]^{g,\{x'\leftarrow c'\}}_{f,\{x\leftarrow x'\}}\sigma' = [\![Q\sigma]\!]^{g\odot\sigma,\{x'\leftarrow c'\}}_{f\odot\sigma,\{\dot{x}\leftarrow x'\}}$	(By IH and (8), (9))
(11)	$[\![P\sigma]\!]^{f\odot\sigma}_{g\odot\sigma} = [\![P]\!]^f_g\sigma'$	(By (10), (6), (7))

□

Compositionality for $[\![\cdot]\!]^g_f$ is rather straightforward, as our translation is homomorphic to both the parallel composition operator and restriction (which is translated to signal declaration in RML).

Theorem 7.10 (Compositionality for $[\![\cdot]\!]^g_f$). *Let P and $E[\cdot]$ be a $\pi_{\mathsf{R}}^{\lightning}$ process and an evaluation context (cf. Def. 2.12), respectively. Then, for every f and g such that $\mathsf{fv}_\pi(P) \subseteq dom(f)$ and $\mathsf{fv}_\pi(P) \subseteq dom(g)$, it holds that $[\![E[P]]\!]^g_f = [\![E]\!]^g_f\big[[\![P]\!]^{g'}_{f'}\big]$, for some f' and g'.*

Proof. By induction on the structure of P and a case analysis on $E[\cdot]$. There is one base case and eight inductive cases. The proofs are straightforward, therefore, we only show two:

Base Case: Whenever $P = \mathbf{0}$. There are three Sub-cases depending on the grammar in Def. 2.12, we only show whenever $E = [\cdot] \mid R$, for any R.

(1)	$E[P] = \mathbf{0} \mid R$	(Assumption)
(2)	$[\![E[P]]\!]^g_f = () \parallel [\![R]\!]^g_f$ for any f and g	(Fig. 7.2)
(3)	$[\![E]\!]^g_f\big[[\![\mathbf{0}]\!]^g_f\big] = () \parallel [\![R]\!]^g_f$ for any f and g	(Fig. 7.2, (2))
(4)	$[\![E]\!]^g_f\big[[\![P]\!]^g_f\big] = [\![E[P]]\!]^g_f$	(By (2),(3))

Inductive Step: There are eight inductive cases, corresponding to each possible process P, each one with three sub-cases corresponding to each possible evaluation context E. We only show whenever $P = x\langle v\rangle.P$ and $E = [\cdot] \mid R$, any R:

(1)	$E[P] = x\langle v\rangle.P \mid R$	(Assumption)
(2)	$[\![E[P]]\!]^g_f = [\![x\langle v\rangle.P]\!]^g_f \parallel [\![R]\!]^g_f$ for any f and g	(Fig. 7.2, (1))
(3)	$[\![E]\!]^g_f\big[[\![P]\!]^g_f\big] = [\![x\langle v\rangle.P]\!]^g_f \parallel [\![R]\!]^g_f$ for any f and g	(Fig. 7.2)
(4)	$[\![E]\!]^g_f\big[[\![P]\!]^g_f\big] = [\![E[P]]\!]^g_f$	(By (2),(3))

□

7.3 Operational Correspondence

7.3.1 *Considerations*

As mentioned in § 3.1.2, big-step semantics such as the one given to RML are not compatible with the reduction semantics of $\pi_{\mathsf{R}}^{\lightning}$.

1. *Signal declaration against restriction:* Signal declaration in RML is not a static entity, as it is the case for restriction $\pi_{\mathsf{R}}^{\lightning}$. In RML for example, RML expression `signal` x `in pause`$;e$ evolves to e in a single big-step reduction. On the other hand, in $\pi_{\mathsf{R}}^{\lightning}$, restrictions are static entities that are carried throughout the execution. For example, if $(\boldsymbol{\nu} xy)P$ has any possible reduction, using Rule $\lfloor\text{Res}\rfloor$ (cf. Fig. 2.1), it can be seen that $(\boldsymbol{\nu} xy)P \longrightarrow (\boldsymbol{\nu} xy)Q$. Nonetheless, this issue can be tackled in RML by showing that signal declarations can be 'inserted' after a big-step reduction without affecting the process behavior.

2. *Big-steps are not reduction steps:* In RML, a big-step reduction does not correspond to single synchronization. Indeed, a single big-step reduction for a translated process executes several synchronizations (as it corresponds to a time instant). This situation is different from $\pi_{\mathsf{R}}^{\lightning}$ where a single reduction step corresponds to a single synchronization step. We tackle this issue by using the big-step semantics for $\pi_{\mathsf{R}}^{\lightning}$ (cf. § 3.1.2).

Signal Declaration Against Restriction

In translations it is not uncommon to find *intermediate processes*, which indicate an execution state in which the translation has not yet completely simulated the source process. In a correct translation, it must be proven that intermediate processes converge to a correctly translated source process.

In our setting, there are no proper intermediate processes. Rather, there are intermediate states in which all the channel and handshake signals have been declared. This fact can be observed by recalling P_2 in Ex. 7.7:

$$P_2 = (\boldsymbol{\nu} x_1x_2)(\boldsymbol{\nu} y_1y_2)(x_1\langle \mathtt{tt}\rangle.\mathbf{0} \mid x_2(z_1).y_2(z_2).\mathbf{0} \mid y_1\langle \mathtt{tt}\rangle.\mathbf{0})$$

we showed that the big-step semantics of RML allow for the following transitions (assuming that $f_{x_1} = x_1$, $g_{x_1} = c_1$, and $f_{y_1} = y_1$, $g_{y_2} = c_2$):

$[\![P_2]\!]^g_f = \texttt{signal}\ x_1, y_1, c_1, c_2\ \texttt{in}$

$$
\begin{aligned}
&\texttt{signal } x_1', c_1' \texttt{ in do loop (emit } x_1 \ (\texttt{tt}, g_{\texttt{tt}}, x_1', c_1'); \texttt{pause) until } c_1 \rightarrow () \parallel \\
&[\![x_2(z_1).y_2(z_2).\mathbf{0}]\!]^{g_1}_{f_1} \parallel \\
&\texttt{signal } y_1', c_2' \texttt{ in do loop (emit } y_1 \ (\texttt{tt}, g_{\texttt{tt}}, y_1', c_2'); \texttt{pause) until } c_2 \rightarrow () \\
\longmapsto\, & () \parallel [\![y_2(z_2).\mathbf{0}]\!]^{g_1'}_{f_1'} \parallel \texttt{do loop (emit } y_1 \ (\texttt{tt}, g_{\texttt{tt}}, y_1', c_2'); \texttt{pause) until } c_2 \rightarrow () \\
=\, & R_1 \\
\longmapsto\, & () \parallel () \parallel () = R_2
\end{aligned}
$$

From here, observe that $R_1 \not\hookrightarrow_{\mathsf{R}} [\![(\boldsymbol{\nu} x_1 x_2)(\boldsymbol{\nu} y_1 y_2)(y_2(z_2).\mathbf{0} \mid y_1\langle \texttt{tt}\rangle.\mathbf{0})]\!]^{g}_{f}$, which is what we expected. In fact, $R_1 \hookrightarrow_{\mathsf{R}} [\![y_2(z_2).\mathbf{0}]\!] \parallel S$, where:

$$S = \texttt{do loop (emit } y_1 \ (\texttt{tt}, \texttt{tt}, y_1', c_2'); \texttt{pause) until } c_2 \rightarrow ()$$

Nonetheless, observing the RML processes, we can see that

$$\texttt{signal } y_1', c_2' \texttt{ in } S = [\![y_1\langle \texttt{tt}\rangle.\mathbf{0}]\!]^{g_1}_{f_1}$$

for some g_1 and some f_1. Furthermore, it is also possible to observe that:

$$
\begin{aligned}
&\texttt{signal } y_1, c_2 \texttt{ in } [\![y_2(z_2).\mathbf{0}]\!]^{g'}_{f'} \parallel \texttt{signal } y_1', c_2' \texttt{ in } S \\
&= [\![(\boldsymbol{\nu} x_1 x_2)(\boldsymbol{\nu} y_1 y_2)(y_2(z_2).\mathbf{0} \mid y_1\langle \texttt{tt}\rangle.\mathbf{0})]\!]^{g'}_{f'}
\end{aligned}
$$

for some f' and g'. Notice that it is possible to obtain the desired translation by adding appropriate signal declarations to replace the ones that were executed. In particular, since our results are restricted to well-typed programs, we can recognize which signals are missing by using the free variables of the translated process. For example, $\mathsf{fv}(R_1) = \{y_1, y_1', c_2', c_2\}$, which means that we need to declare the signals appropriately in the process, as done above. We will say that any target term whose signals have been initialized is called an *initialized translation*.

Big-Steps are Not Reduction Steps

A single big-step reduction in $[\![\cdot]\!]^{g}_{f}$ simulates several reduction steps in $\pi_{\mathsf{R}}^{\lightning}$. To address this discrepancy we shall use the big-step semantics for $\pi_{\mathsf{R}}^{\lightning}$ given in § 3.1.2. This semantics correctly matches the execution of a big-step reduction in RML. Let us start by presenting an example:

Example 7.11. Consider the process

$$P_3 = (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(\boldsymbol{\nu} x_1 y_1)(x\langle z\rangle.w\langle \texttt{tt}\rangle.\mathbf{0} \mid y(w').w'(w'').\mathbf{0} \mid x_1\langle v_1\rangle.\mathbf{0} \mid y_1(z_1).\mathbf{0}) \quad (7.8)$$

A possible reduction is:

$$P_3 \longrightarrow (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(\boldsymbol{\nu} x_1 y_1)(x\langle z\rangle.w\langle \texttt{tt}\rangle.\mathbf{0} \mid y(w').w'(w'').\mathbf{0} \mid \mathbf{0} \mid \mathbf{0})$$

Using the big-step semantics of RML, it can be shown that the translation exhibits the following behavior in a single big-step reduction:

$$[\![P_3]\!]^{g}_{f} \longmapsto [\![w\langle \texttt{tt}\rangle.\mathbf{0}]\!]^{g_1}_{f_1} \parallel [\![z(w'').\mathbf{0}]\!]^{g_1}_{f_1} \parallel () \parallel ()$$

Which corresponds to more than a single reduction in $\pi_{\mathsf{R}}^{\lightning}$:

$$P_3 \longrightarrow^2 (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(\boldsymbol{\nu} x_1 y_1)(w\langle \texttt{tt}\rangle.\mathbf{0} \mid z(w'').\mathbf{0} \mid \mathbf{0} \mid \mathbf{0})$$

△

Proof Outline

Below we state two operational correspondence statements that must be proven to ensure this translation is indeed an encoding. The first one encompasses the operational soundness and completeness of valid encodings (cf. Def. 2.3), and uses the big-step semantics of $\pi_{\mathsf{R}}^{\lightning}$ (cf. § 3.1.2). The second one corresponds to the operational correspondence statement of refined encodings (cf. Def. 2.6), and uses the reduction semantics in Fig. 2.1 for $\pi_{\mathsf{R}}^{\lightning}$. Notice that it will be shown later that Thm. 7.13 can be derived from the semantic correspondence (cf. Lem. 3.35) and Thm. 7.12.

Theorem 7.12 (Valid Operational Correspondence). *Let P be a well-typed $\pi_{\mathsf{R}}^{\lightning}$ program (cf. Not. 2.21). Also, let f and g be renaming functions. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \hookrightarrow\!\!\!\!\rightarrow^{*} Q$, it holds that $[\![P]\!]_f^g \longmapsto^{*} \lesssim [\![Q]\!]_{f'}^{g'}$, for some f' and g'.*

2. **Soundness:** *For every* RML *process S such that $[\![P]\!]_f^g \longmapsto^{*} S$, there exists Q such that $P \hookrightarrow\!\!\!\!\rightarrow^{*} Q$ and $S \lesssim [\![Q]\!]_{f'}^{g'}$, for some f' and g'.*

Theorem 7.13 (Refined Operational Correspondence). *Let P be a well-typed $\pi_{\mathsf{R}}^{\lightning}$ program (cf. Not. 2.21). Also, let f and g be renaming functions. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \longrightarrow^{*} Q$, then there exists Q' such that $[\![P]\!]_f^g \longmapsto^{*} \lesssim [\![Q']\!]_{f'}^{g'}$ and $Q \longrightarrow^{*} Q'$, for some f' and g'.*

2. **Soundness:** *For every* RML *process S such that $[\![P]\!]_f^g \longmapsto^{*} S$, there exists Q such that $P \longrightarrow^{*} Q$ and $S \lesssim [\![Q]\!]_{f'}^{g'}$, for some f' and g'.*

Before presenting a proof outline, we briefly discuss about the two key insights that make up the proof. They are enumerated below.

(1) It can be shown that for every target term S there exists a signal declaration context $D_{\widetilde{x}}$ such that $S \lesssim [\![P]\!]_f^g$ (cf. Def. 2.42) for some well-typed program P.

(2) It can be shown that for every well-typed program P such that $P \hookrightarrow\!\!\!\!\rightarrow P'$ (cf. Def. 3.31), then $[\![P]\!]_f^g \longmapsto \lesssim [\![P']\!]_f^g$ and vice versa.

We now give an outline of the proof. First, we sketch the necessary auxiliary lemmas; then, we discuss how these lemmas are used to prove Thm. 7.13. We first focus on syntactically characterizing target terms since they exhibit concrete syntactical characteristics. Using these characterizations we proceed to prove Thm. 7.13(1) and then Thm. 7.13(2). Details follow:

(1) First, Lem. 7.15, Def. 7.16 and Cor. 7.17 are used to characterize the syntactic structure of translated pre-redexes and characterizing the target terms that can be obtained from a single RML big-step reduction (so-called *initialized translations*).

(2) Lem. 7.19 shows that it is possible to add the missing signal declarations on initialized translations of pre-redexes and conditionals, obtaining a RML expression that corresponds exactly to the translation of the source $\pi_{\mathsf{R}}^{\lightning}$ process.

(3) Lem. 7.20 is used to characterize the shape of the translation of well-typed programs, which is useful when dealing with target terms.

(4) Lem. 7.23 shows that the translations of enabled redexes (cf. Def. 7.21) evolves, in a single RML big-step reduction, to the translation of a process that can be obtained by reducing source process; i.e., if R is a redex and $(\boldsymbol{\nu}xy)R \longrightarrow$, then $[\![(\boldsymbol{\nu}xy)R]\!]_f^g \longmapsto\!\hookrightarrow_{\mathsf{R}} [\![R']\!]_{f'}^{g'}$ for some f', g' and $(\boldsymbol{\nu}xy)R \longrightarrow (\boldsymbol{\nu}xy)R'$.

(5) Lem. 7.27 generalizes the results in Item (4) to the translation of well-typed programs.

Using these results we prove the operational completeness and operational soundness results to demonstrate Thm. 7.13:

Completeness: The proof proceeds by induction on the length of the big-step reduction $P \hookrightarrow\!\!\!\rightarrow^* P'$. The base case is immediate. The inductive step follows from the conjunction of Lem. 7.28 and the IH. In Lem. 7.28 we show that a single $\pi_{\mathsf{R}}^{\lightning}$ big-step reduction (cf. Def. 3.31) can be matched by a single RML big-step reduction.

Soundness: The proof of operational soundness has the following structure: first, Thm. 7.30 shows that every RML big-step reduction of a translated process can be matched by multiple $\pi_{\mathsf{R}}^{\lightning}$ reductions (i.e., $\longrightarrow^*$). Then, we prove that if a single RML big-step reduction is matched by multiple $\pi_{\mathsf{R}}^{\lightning}$ reductions, then it must be the case that the same single RML big-step reduction is matched by a $\pi_{\mathsf{R}}^{\lightning}$ big-step reduction (cf. Lem. 7.31). Finally, we prove completeness by applying induction on the number of RML steps (cf. Cor. 7.32).

Auxiliary Results

Some auxiliary results and definitions follow. We first formalize the notion of target term for $[\![\cdot]\!]_f^g$:

Definition 7.14 (Target Terms). We define *target terms* as the set of RML expressions that are induced by the translation $[\![\cdot]\!]_f^g$ of well-typed $\pi_{\mathsf{R}}^{\lightning}$ programs and is closed under $\longmapsto$: $\{S \mid [\![P]\!]_f^g \longmapsto S \text{ and } \vdash P\}$. We shall use $S, S', \ldots$ to range over target terms.

We then show that the translation of a pre-redex or inaction process either executes a RML big-step reduction to the encoding of itself or to some RML expression that can only reduce to itself:

Lemma 7.15. *For every pre-redex or inaction $\pi_{\mathsf{R}}^{\lightning}$ process P (cf. Def. 2.19) it holds that either (1) $[\![P]\!]_f^g \longmapsto [\![P]\!]_f^g$ or (2) there exists some target term S such that $[\![P]\!]_f^g \longmapsto S \longmapsto^* S$.*

Proof (Sketch). We prove each item individually. In each item we apply Fig. 7.2 to P and we conclude by using the rules in the semantics of RML in Fig. 2.8 for the resulting target term $[\![P]\!]_f^g$. □

We now introduce the initialized translation of a pre-redex and the inaction process. Notice that conditionals such as $v?\,(P_1):(P_2)$ do not have an initialized translation.

$$
init(\llbracket P \rrbracket_f^g) \overset{\mathsf{def}}{=} \begin{cases}
\llbracket P \rrbracket_f^g & \text{if } P = \mathbf{0}, P = x(y).Q \text{ or} \\
& P = x \triangleright \{l_i : Q_i\}_{i \in I} \\
\texttt{do (loop (emit } f_x\ (m, g_m, x', c')\texttt{; pause))} & \\
\ \texttt{until } g_x \to \llbracket Q \rrbracket_{f,\{x \to x'\}}^{g,\{x' \to c'\}} & \text{if } P = x\langle m \rangle.Q \\
\texttt{do (loop (emit } f_x\ (m, x', c')\texttt{; pause))} & \\
\ \texttt{until } g_x \to \llbracket Q \rrbracket_{f,\{x \to x'\}}^{g,\{x' \to c'\}} & \text{if } P = x \triangleleft m.Q \\
\texttt{do emit } g_x \texttt{ when } f_x\texttt{;} & \\
\texttt{await } f_x(y, h_y, w, h_w) \texttt{ in} & \\
\quad \texttt{run process } \llbracket Q \rrbracket_{f,\{\alpha \leftarrow w, y \leftarrow y\}}^{g,\{w \leftarrow h_w, y \leftarrow h_y\}} \ \| & \text{if } P = * \, x(y).Q \\
\quad \texttt{run } \llbracket * \, x(y).Q \rrbracket_{f,\{\alpha \leftarrow w, y \leftarrow y\}}^{g,\{w \leftarrow h_w, y \leftarrow h_y\}} &
\end{cases}
$$

Figure 7.3: Initialized Translations.

Definition 7.16 (Initialized Translations). Let P be a $\pi_{\mathsf{R}}^{\lightning}$ pre-redex (cf. Def. 2.19) or the inaction process. The initialized translation of P, written $init(\llbracket P \rrbracket_f^g)$, is given in Fig. 7.3.

Using Lem. 7.15 and Def. 7.16, we can prove the following corollary by applying a case analysis on P:

Corollary 7.17. *For every $\pi_{\mathsf{R}}^{\lightning}$ pre-redex P, $\llbracket P \rrbracket_f^g \longmapsto init(\llbracket P \rrbracket_f^g) \longmapsto^* init(\llbracket P \rrbracket_f^g)$.*

We now introduce some useful notation for renaming functions whenever they are applied to sequences of variables:

Notation 7.18. Let f be a renaming function and x be a sequence of variables $x_1 \ldots x_n$ with $n \geq 1$. The following notation will be used throughout Chapters 7 and 8.

- We write $f_{\widetilde{x}}$ to denote the sequence formed by $f_{x_1} \ldots f_{x_n}$.
- Assuming another renaming function f', we write $f_{\widetilde{x}} f'_{\widetilde{x}}$ to denote the concatenation of the sequences $f_{\widetilde{x}}$ and $f'_{\widetilde{x}}$.
- Assuming another sequence of variables $\widetilde{x}'$ such that $|\widetilde{x'}| = |\widetilde{x}|$, we write $f, \{\widetilde{x} \leftarrow \widetilde{x}'\}$ to refer to the update $f, \{x_1 \leftarrow x'_1, \ldots, x_n \leftarrow x'_n\}$.
- We use the same notation for renaming function g.

Using the previous notation and lemmas we can prove the existence of signal declaration contexts (cf. Def. 2.40) for the initialized translations of pre-redexes:

Lemma 7.19. *For every $\pi_{\mathsf{R}}^{\lightning}$ pre-redex or inaction process P and every target term S, it holds that $S = init(\llbracket P \rrbracket_f^g)$ implies that either:*

1. *$S = \llbracket P \rrbracket_f^g$ (or)*

2. *there exists a declaration context $D_{\widetilde{x}}$ such that $D_{\widetilde{x}}[S] = [\![P]\!]^g_f$, for some g and f and $\widetilde{x} = (\mathsf{fv}(S) \cap \mathsf{fv}([\![P]\!]^f_g)) \setminus (ran(f) \cup ran(g))$.*

Proof. By applying a case analysis on S. For details see App. D.1. □

We now show that the syntactic shape of well-typed $\pi_{\mathsf{R}}^{\lightning}$ programs induces a concrete structure in the syntax of translated processes:

Lemma 7.20. *For every well-typed $\pi_{\mathsf{R}}^{\lightning}$ program $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})P$ (cf. Not. 2.21), and any well-defined renaming functions f and g (cf. Def. 7.2), it holds that:*

$$[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})P]\!]^g_f = \texttt{signal}\ \tilde{z}\tilde{c}\ \texttt{in}\ ([\![P_1]\!]^{g_1}_{f_1} \parallel [\![P_2]\!]^{g_1}_{f_1} \parallel \cdots \parallel [\![P_n]\!]^{g_1}_{f_1})$$

where $n \geq 1$, every P_i, $1 \leq i \leq n$ is either a pre-redex (cf. Def. 2.19) or $P_i = v?\,(Q_1)\!:\!(Q_2)$, and $f_1 = f, \{\widetilde{x} \leftarrow \widetilde{z}, \widetilde{y} \leftarrow \widetilde{z}\}$ and $g_1 = g, \{\widetilde{x} \leftarrow \widetilde{c}, \widetilde{y} \leftarrow \widetilde{c}\}$, with $|\widetilde{z}| = |\widetilde{c}| = |\widetilde{x}| = |\widetilde{y}| \geq 1$.

Proof. Assume $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})P$ is a well-typed $\pi_{\mathsf{R}}^{\lightning}$ program. Then, by Cor. 3.34 applied to P, we obtain $P \equiv_{\mathsf{S}} (\boldsymbol{\nu}\tilde{x}\tilde{y})(P_1 \mid P_2 \mid \ldots \mid P_n)$. Finally, apply Fig. 7.2:

$$[\![(\boldsymbol{\nu}\tilde{x}\tilde{y})(P_1 \mid P_2 \mid \ldots \mid P_n)]\!]^g_f = \texttt{signal}\ \tilde{z}\tilde{c}\ \texttt{in}\ ([\![P_1]\!]^{g_1}_{f_1} \parallel [\![P_2]\!]^{g_1}_{f_1} \parallel \cdots \parallel [\![P_n]\!]^{g_1}_{f_1})$$

which is what we wanted to prove. □

Up until this point we have characterized the dynamic behavior of translated pre-redexes. We have shown that the translation of pre-redexes have big-step reductions that cannot be precisely matched by source $\pi_{\mathsf{R}}^{\lightning}$ processes (as pre-redexes cannot reduce). These big-step reductions correspond to initialized translations. We have also shown that it is possible to recover the initial translation by adding a harmless context that rebinds the signals being declared. In the sequel, we focus on analyzing the behavior of translated $\pi_{\mathsf{R}}^{\lightning}$ programs, rather than pre-redexes. We first introduce some auxiliary definitions:

Definition 7.21 (Enablers for $\pi_{\mathsf{R}}^{\lightning}$ Processes). Let P be an $\pi_{\mathsf{R}}^{\lightning}$ process. We say that the sequence of variables $\widetilde{x}, \widetilde{y}$ *enable* P if there is some P' such that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})P \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})P'$.

Enablers allow us to refer to the specific sequence of variables and co-variables that enable $\pi_{\mathsf{R}}^{\lightning}$ processes to reduce. Following our previous results, it is expected that enablers are 'lost' in a big-step reduction from a translated $\pi_{\mathsf{R}}^{\lightning}$ process.

Example 7.22. Let $R = x\langle v\rangle.P \mid y(z).Q$. Assuming that $v \in \{\texttt{tt}, \texttt{ff}\}$, then R is enabled by x, y. Thus, $(\boldsymbol{\nu}xy)R \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q\{v/z\})$. Therefore,

$$[\![(\boldsymbol{\nu}xy)R]\!]^g_f = \texttt{signal}\ w, h_w\ \texttt{in}\ [\![x\langle v\rangle.P]\!]^{g,\{x\leftarrow h_w, y\leftarrow h_w\}}_{f,\{x\leftarrow w, y\leftarrow w\}} \parallel [\![y(z).Q]\!]^{g,\{x\leftarrow h_w, y\leftarrow h_w\}}_{f,\{x\leftarrow w, y\leftarrow w\}}$$

Using the big-step semantics of RML it is possible to show that:

$$[\![(\boldsymbol{\nu}xy)R]\!]^g_f \longmapsto [\![P]\!]^{g'}_{f'} \parallel [\![Q\{v/z\}]\!]^{g'}_{f'}$$

for some unimportant f', g'. Notice that the declaration $\texttt{signal}\ w, h_w\ \texttt{in}\ \ldots$ disappears in the transition. Therefore, to recover the translation it is necessary to insert a signal declaration context $D_{\tilde{z}}$ which rebinds the lost signal declarations. △

The following lemma proves that whenever a redex reduces, it is possible to find the necessary signals to define a signal declaration context (cf. Def. 2.40) that recovers the translation of the enablers of said redex.

Lemma 7.23. *Let R be a well-typed $\pi_{\mathsf{R}}^{\sharp}$ redex enabled by $\widetilde{x}, \widetilde{y}$. If $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R]\!]_f^g \longmapsto S$ then there exists R' such that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'$ and the following holds:*

1. *$S \hookrightarrow_{\mathsf{R}} [\![R']\!]_{f'}^{g'}$ for some f', g' such that $\widetilde{x}\widetilde{y} \in dom(f')$ and $\widetilde{x}\widetilde{y} \in dom(g')$ and*

2. *there exists $D_{\widetilde{z}}$ such that $D_{\widetilde{z}}[S] \equiv_\alpha [\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R']\!]_{f'}^{g'}$ for some $\widetilde{z}$.*

Proof. By a case analysis on the possible redexes. For every case we first show Item (1) and then Item (2) (for details see App. D.1). □

We now fully characterize target terms. The following lemma shows that a translated $\pi_{\mathsf{R}}^{\sharp}$ program either reduces to an initialized translation or to the translation of some $\pi_{\mathsf{R}}^{\sharp}$ program.

Lemma 7.24. *Let $P = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_n)$, $n \geq 1$ be a well-typed $\pi_{\mathsf{R}}^{\sharp}$ program. Then, the following holds: for every S such that $[\![P]\!]_f^g \longmapsto^* S$, it holds that (1) $S = S_1 \parallel \cdots \parallel S_m$, $m \geq n$, (2) there exists $P' = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid P'_m)$ such that $P \longrightarrow^* P'$ and (3) for every $1 \leq i \leq m$ there exist f', g' such that either:*

1. *$S_i \hookrightarrow_{\mathsf{R}} init([\![P'_i]\!]_{f'}^{g'})$, for some pre-redex P'_i or*

2. *$S_i \hookrightarrow_{\mathsf{R}} [\![P'_i]\!]_{f'}^{g'}$, such that P'_i is a pre-redex, $P'_i = v?\,(Q_1) : (Q_2)$, or $P'_i = \mathbf{0}$.*

Proof. By Cor. 3.34, for every $1 \leq i \leq n$, P_i is a pre-redex or $P_i = v?\,(P'_i) : (P''_i)$. We apply induction on the length r of transition $[\![P]\!]_f^g \longmapsto^* S$. The base case is $[\![P]\!]_f^g \longmapsto^* [\![P]\!]_{f}^{g}$, which is immediate. For details on the inductive step see App. D.1. □

As a corollary from Lem. 7.19 and Lem. 7.24 we have that:

Corollary 7.25. *Let $P = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_n)$, $n \geq 1$ be a well-typed $\pi_{\mathsf{R}}^{\sharp}$ process. Then, for every S such that $[\![P]\!]_g^f \longmapsto^* S$, it holds that (1) $S = S_1 \parallel \cdots \parallel S_m$, $m \geq n$, (2) there exists $P' = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid P'_m)$ such that $P \longrightarrow^* P'$ and (3) for every $1 \leq i \leq m$, there exist a, possible empty, declaration context $D_{\widetilde{z_i}}$ and some renaming functions f' and g' such that:*

$$D_{\widetilde{z_1}}[S_1] \parallel \cdots \parallel D_{\widetilde{z_m}}[S_m] \hookrightarrow_{\mathsf{R}} [\![P'_1]\!]_{f'}^{g'} \parallel \cdots \parallel [\![P_m]\!]_{f'}^{g'}$$

From Lem. 7.23(2), we know that there exists a declaration context for the process obtained by the big-step reduction of a translated redex. This context contains all the necessary signal declarations to recover the translation of any process reachable from the source process. We can generalize this result to well-typed programs, as shown below. We first introduce an auxiliary lemma and then state this fact.

Lemma 7.26. *Let P be a well-typed program. Then, $\mathsf{fv}_\pi(P) = \mathsf{fv}([\![P]\!]_f^g) = \emptyset$.*

Proof. By induction on the structure of P. All the cases are immediate. □

Lemma 7.27. *Let $P = (\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \ldots \mid P_n)$, $n \geq 1$ be a well-typed program. For every S such that $[\![P]\!]_f^g \longmapsto^* S$, it holds that (1) $S = S_1 \parallel \cdots \parallel S_m$, $m \geq n$, (2) there exists $P' = (\boldsymbol{\nu}\widetilde{xy})(P_1' \mid \ldots \mid P_m')$ such that $P \longrightarrow^* P'$ and (3) there exist signal declaration contexts (cf. Def. 2.40) $D_{\widetilde{z}}, D_{\widetilde{z_1}}, \ldots, D_{\widetilde{z_m}}$ and renaming functions f' and g' such that:*

$$D_{\widetilde{z}}[D_{\widetilde{z_1}}[S_1] \parallel \cdots \parallel D_{\widetilde{z_m}}[S_m]] \hookrightarrow_{\mathsf{R}} [\![(\boldsymbol{\nu}\widetilde{xy})(P_1' \mid \ldots \mid P_m')]\!]_{f'}^{g'}$$

Proof. From Cor. 7.25 it follows that:

$$\exists D_{\widetilde{z_1}}, \ldots, D_{\widetilde{z_m}}.(D_{\widetilde{z_1}}[S_1] \parallel \cdots \parallel D_{\widetilde{z_m}}[S_m] \hookrightarrow_{\mathsf{R}} [\![P_1']\!]_{f'}^{g'} \parallel \cdots \parallel [\![P_m']\!]_{f'}^{g'}) \tag{7.9}$$
$$P \longrightarrow^* (\boldsymbol{\nu}\widetilde{xy})(P_1' \mid \ldots \mid P_m') \tag{7.10}$$

To prove the existence of $D_{\widetilde{z}}$, we proceed by induction on the number of enabled redexes. Notice that typability and Thm. 3.28 will ensure that it is not possible for multiple S_i reacting to the same signal being emitted:

Base Case: There are $r = 0$ enabled redexes in P. If there are no enabled redexes, then $P \not\longrightarrow$. Then, by Lem. 7.24, $[\![P]\!]_f^g \longmapsto S_1 \parallel \cdots \parallel S_m$, $m \geq n$, where for every $1 \leq i \leq m$, process S_i is either the encoding of an initialized pre-redex (cf. Def. 7.16), a pre-redex, a conditional process or the inaction process. Furthermore, by (1) and (2):

$$D_{\widetilde{z_1}}[S_1] \parallel \cdots \parallel D_{\widetilde{z_m}}[S_m] \hookrightarrow_{\mathsf{R}} [\![P_1]\!]_{f'}^{g'} \parallel \cdots \parallel [\![P_m]\!]_{f'}^{g'}$$

and since $P \not\longrightarrow$, then $m = n$. Therefore from Lem. 7.26, we conclude that to obtain $D_{\widetilde{z}}$, we need to close all the remaining free variables appropriately. For this, we take the sequence $\widetilde{z} = f_{\widetilde{x}} g_{\widetilde{x}}$ and conclude the proof.

Inductive Step: Assume $r > 1$ enabled redexes in P. We prove that when adding a new enabled redex, it is possible to find the necessary contexts. This follows from Lem. 7.23 and IH.

□

7.3.2 *Operational Completeness*

For proving operational completeness we must first show that a single $\pi_{\mathsf{R}}^{\lightning}$ big-step reduction is matched by a single RML big-step reduction.

Lemma 7.28. *Let P be a well-typed $\pi_{\mathsf{R}}^{\lightning}$ program. Then, for every P' such that $P \hookrightarrow\!\!\!\!\rightarrow P'$, it holds that $[\![P]\!]_f^g \longmapsto\lesssim [\![P']\!]_{f'}^{g'}$ for some f' and some g'.*

Proof. By Cor. 3.34, $P = (\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \ldots \mid P_n)$, $n \geq 1$ and for every $1 \leq i \leq n$, P_i is either a pre-redex (cf. Def. 2.19) or a conditional process. We proceed by induction on the number n of processes in P:

Base Case: Whenever $n = 1$. There are seven cases corresponding to all five pre-redexes, inaction and conditional process. The former two are immediate, since by Fig. 2.1, there are no reductions for individual pre-redexes or inaction. The case for $P_1 = b?\,(Q_1) : (Q_2)$ is also immediate since by Fig. 2.8, $[\![(\boldsymbol{\nu}\widetilde{xy})(b?\,(Q_1) :$

$(Q_2))]\!]^g_f \longmapsto\lesssim [\![Q_i]\!]^g_f$, $i \in \{1,2\}$, depending on $b \in \{\mathtt{tt},\mathtt{ff}\}$. Similarly, we have that $(\boldsymbol{\nu}\widetilde{xy})(b?\,(Q_1):(Q_2)) \longrightarrow Q_i$, depending on b (Rules $\lfloor$IFT$\rfloor$ or $\lfloor$IFF$\rfloor$ in Fig. 2.1).

Inductive Step: Whenever $n > 1$. Assume that $P = (\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \ldots \mid P_{n-1})$ and we need to show that the property holds whenever we add a new pre-redex, inaction or conditional process P_n. In all these cases $f' = f$ and $g' = g$.

IH: If $(\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \ldots \mid P_{n-1}) \hookrightarrow\!\!\!\!\!\rightarrow (\boldsymbol{\nu}\widetilde{xy})(P'_1 \mid \ldots \mid P'_{n-1})$ then

$$[\![(\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \ldots \mid P_{n-1})]\!]^g_f \longmapsto\lesssim [\![(\boldsymbol{\nu}\widetilde{xy})(P'_1 \mid \ldots \mid P'_{n-1})]\!]^{g'}_{f'}$$

for some f', g'.

We proceed by a case analysis on P_n. There are seven cases. Cases $P_n = \mathbf{0}$ and $P_n = b?\,(Q_1):(Q_2)$ are immediate as there is no interaction with other P_i, $1 \leq i \leq n-1$. Whenever P_n is pre-redex, there are five cases, which are similar. We will only show one $P_n = x\langle v\rangle.Q$ as all the other cases are similar:

Case $P_n = x\langle v\rangle.Q$: There are two cases: Whenever there exists $1 \leq i \leq n-1$ such that $P_i = y(z).R$ and $(\boldsymbol{\nu}\widetilde{xy})(P_n \mid P_i)$ is an enabled redex (cf. Def. 7.21) or whenever such P_i does not exist. Notice that by typability and Thm. 3.28 if such P_i exists, it is unique (cf. Def. 3.27).

Case $\exists P_i = y(z).R$: Let $M = (\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \ldots \mid y(z).R \mid \ldots \mid P_{n-1})$.

(1)	$\vdash P$	(Assumption)
(2)	$P = (\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \ldots \mid y(z).R \mid \ldots \mid P_{n-1} \mid x\langle v\rangle.Q)$	(Assumption)
(3)	$P \hookrightarrow\!\!\!\!\!\rightarrow P'$ $P' = (\boldsymbol{\nu}\widetilde{xy})(P'_1 \mid \ldots \mid R\{^v\!/z\} \mid \ldots \mid P'_{n-1} \mid Q)$	(By Def. 3.31, (2))
(4)	$M \hookrightarrow\!\!\!\!\!\rightarrow M'$ $M' = (\boldsymbol{\nu}\widetilde{xy})(P'_1 \mid \ldots \mid y(z).R \mid \ldots \mid P'_{n-1})$	(By (1), Def. 3.31, (2))
(5)	$[\![M]\!]^g_f \longmapsto\lesssim [\![M']\!]^{g'}_{f'}$, for some f', g'	(IH, (4))
(6)	$[\![P]\!]^g_f \longmapsto\lesssim [\![P']\!]^{g'}_{f'}$ for some f', g'	((4), Fig. 2.8, Lem. 7.27)

Case $\neg\exists P_i = y(z).R$: The proof follows by IH and Lem. 7.27. Notice that there is no process to await the signal emitted by P_n. Therefore, there is no synchronization.

□

Having proving this correspondence, we can now prove that the translation is operationally complete for $\hookrightarrow\!\!\!\!\!\rightarrow$ (Thm. 7.12(2)).

Theorem 7.29 (Refined Completeness for $[\![\cdot]\!]^g_f$). *Let P be a well-typed $\pi_{\mathsf{R}}^{\natural}$ program. Then, for every Q such that $P \hookrightarrow\!\!\!\!\!\rightarrow^* Q$, it holds that $[\![P]\!]^g_f \longmapsto^*\lesssim [\![Q]\!]^{g'}_{f'}$, for some f' and some g'.*

Proof. By induction on the length of $P \hookrightarrow\!\!\!\!\!\rightarrow^* P'$. The base case is immediate. The inductive step follows from the conjunction of the IH and Lem. 7.28. □

7.3.3 *Operational Soundness*

For proving the soundness property, we first require proving Thm. 7.13(2).

Theorem 7.30 (Valid Soundness). *For every well-typed $\pi_{\mathsf{R}}^{\lightning}$ program P and every* RML *process S such that $[\![P]\!]_f^g \longmapsto^* T$, there exists P' such that $P \longrightarrow^* P'$ and $S \lesssim [\![P']\!]_{f'}^{g'}$, for some f' and some g'.*

Proof. Follows directly from Lem. 7.27. □

We now prove that whenever there is a correspondence between a single RML big-step reduction of a translated $\pi_{\mathsf{R}}^{\lightning}$ process and multiple reduction steps in $\pi_{\mathsf{R}}^{\lightning}$, then the sequence of steps must necessarily be a single $\pi_{\mathsf{R}}^{\lightning}$ big-step reduction (cf. Def. 3.31).

Lemma 7.31. *Let P, P' be well-typed programs and f and g, be well-defined renaming functions (cf. Def. 7.2). Then, if $[\![P]\!]_f^g \longmapsto\lesssim [\![P']\!]_{f'}^{g'}$, for some f' and some g', and $P \longrightarrow^* P'$, it holds that $P \hookrightarrow\!\!\!\!\!\rightarrow P'$.*

Proof. By Cor. 3.34, $P = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_n)$, $n \geq 1$ and for every $1 \leq i \leq n$, P_i is either a pre-redex (cf. Def. 2.19) or a conditional process. We proceed by induction on the number n of processes in P:

Base Case: Whenever $n = 1$. There are seven cases corresponding to all five pre-redexes, inaction and conditional process. The former two are immediate, since by Fig. 2.1, there are no reductions for individual pre-redexes or inaction. The case for $P_1 = b?\,(Q_1)\!:\!(Q_2)$ is also immediate since, by Fig. 2.8, $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(b?\,(Q_1)\!:\!(Q_2))]\!]_f^g \longmapsto\lesssim [\![Q_i]\!]_f^g$, $i \in \{1,2\}$, depending on $b \in \{\mathtt{tt}, \mathtt{ff}\}$. Similarly, we have that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(b?\,(Q_1)\!:\!(Q_2)) \longrightarrow Q_i$, depending on b (Rules $\lfloor \textsc{IfT} \rfloor$ or $\lfloor \textsc{IfF} \rfloor$), which implies, by Def. 3.31, that $P \hookrightarrow\!\!\!\!\!\rightarrow P'$.

Inductive Case: Assume $n \geq 1$.

IH: If $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_{n-1})]\!]_f^g \longmapsto\lesssim [\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1' \mid \ldots \mid P_m')]\!]_{f'}^{g'}$, for some f' and some g', with $m \geq n$ and

$$(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_{n-1}) \longrightarrow^* (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1' \mid \ldots \mid P_m')$$

then $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_{n-1}) \hookrightarrow\!\!\!\!\!\rightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1' \mid \ldots \mid P_m')$

We show that whenever we add a new process P_n in P, the property still holds. The proof proceeds by a case analysis on P_n. There are seven cases corresponding to all five pre-redexes, inaction and conditional process. The inaction process is immediate, as well as the conditional, which proceed by IH and using the semantics in Fig. 2.8 and Rule $\lfloor \textsc{Big-Step} \rfloor$ in Def. 3.31.

Whenever we add a pre-redex, there will be two cases: (1) whenever P_n can synchronize with another P_i, $1 \leq i \leq n-1$, and (2) whenever such synchronization cannot occur. Case (2) follows immediately by IH, as we do not introduce any reduction. In Case (1), the proof is straightforward by identifying the synchronization Rule (i.e., $\lfloor \textsc{Com} \rfloor$, $\lfloor \textsc{Sel} \rfloor$, $\lfloor \textsc{Rep} \rfloor$) that can occur with the added P_n. Then, by well-formedness (cf. Def. 3.27), adding P_n does only induces a unique redex. Thus, we can show that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_{n-1} \mid P_n) \hookrightarrow\!\!\!\!\!\rightarrow$

$(\boldsymbol{\nu}\widetilde{xy})(P_1' \mid \dots \mid P_{n-1}' \mid P_n')$ is a valid application of Rule $\lfloor$Big-Step$\rfloor$. Finally, it can be shown that

$$[\![(\boldsymbol{\nu}\widetilde{xy})(P_1 \mid \dots \mid P_{n-1} \mid P_n)]\!]_f^g \longmapsto \lesssim [\![(\boldsymbol{\nu}\widetilde{xy})(P_1' \mid \dots \mid P_{n-1}' \mid P_n')]\!]_{f'}^{g'}$$

by using the semantics in Fig. 2.8.

□

Thm. 7.12(2) follows as a corollary from Thm. 7.30 and Lem. 7.31, by applying induction on the number of transitions in $[\![P]\!]_f^g \longmapsto^* T$. Similarly, Thm. 7.13(1) follows from the composition of Lem. 3.35(2) and Thm. 7.12(1).

Corollary 7.32 (Refined Soundness for $[\![\cdot]\!]_f^g$). *For every well-typed $\pi_{\mathsf{R}}^{\lightning}$ program P and every* RML *process S such that $[\![P]\!]_f^g \longmapsto^* S$, there exists P' such that $P \hookrightarrow\!\!\!\rightarrow^* P'$ and $S \lesssim [\![P']\!]_{f'}^{g'}$, for some f' and some g'.*

Corollary 7.33 (Refined Completeness). *For every Q such that $P \longrightarrow^* Q$ there exists Q' such that $[\![P]\!]_f^g \longmapsto\lesssim [\![Q']\!]_{f'}^{g'}$ and $Q \longrightarrow^* Q'$ for some f' and g'.*

We now prove that operational correspondence holds for translation $[\![\cdot]\!]_f^g$. We prove a *valid operational* correspondence, used to demonstrate the translation is a valid encoding (cf. Def. 2.3) and a *refined* operational correspondence, used to show that the translation is a refined encoding (cf. Def. 2.6).

Theorem 7.12 (Valid Operational Correspondence). *Let P be a well-typed $\pi_{\mathsf{R}}^{\lightning}$ program (cf. Not. 2.21). Also, let f and g be renaming functions. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \hookrightarrow\!\!\!\rightarrow^* Q$, it holds that $[\![P]\!]_f^g \longmapsto^*\lesssim [\![Q]\!]_{f'}^{g'}$, for some f' and g'.*

2. **Soundness:** *For every* RML *process S such that $[\![P]\!]_f^g \longmapsto^* S$, there exists Q such that $P \hookrightarrow\!\!\!\rightarrow^* Q$ and $S \lesssim [\![Q]\!]_{f'}^{g'}$, for some f' and g'.*

Proof. Follows from Thm. 7.29 and Cor. 7.32. □

Theorem 7.13 (Refined Operational Correspondence). *Let P be a well-typed $\pi_{\mathsf{R}}^{\lightning}$ program (cf. Not. 2.21). Also, let f and g be renaming functions. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \longrightarrow^* Q$, then there exists Q' such that $[\![P]\!]_f^g \longmapsto^*\lesssim [\![Q']\!]_{f'}^{g'}$ and $Q \longrightarrow^* Q'$, for some f' and g'.*

2. **Soundness:** *For every* RML *process S such that $[\![P]\!]_f^g \longmapsto^* S$, there exists Q such that $P \longrightarrow^* Q$ and $S \lesssim [\![Q]\!]_{f'}^{g'}$, for some f' and g'.*

Proof. Follows from Cor. 7.33 and Thm. 7.30. □

We now summarize the correctness results to ensure that our translation is indeed an encoding. In the case of $[\![\cdot]\!]_f^g$, we have that the translation is both a valid and refined encoding—see Def. 2.3 and Def. 2.6.

Theorem 7.34 (RML **Encodes** $\pi_{\mathsf{R}}^{\lightning}$). *Consider the languages $\mathcal{L}_{\pi_{\mathsf{R}}^{\lightning}}$, $\mathcal{L}^{*}_{\pi_{\mathsf{R}}^{\lightning}}$, and $\mathcal{L}_{\mathsf{RML}}$ as they were given in Def. 3.100(3,4) and Def. 3.102(3), respectively. Then, the following holds:*

(1) *Translation $\langle [\![\cdot]\!]_f^g, \psi_{[\![\cdot]\!]_f^g} \rangle$, which maps $\mathcal{L}_{\pi_{\mathsf{R}}^{\lightning}}$ into $\mathcal{L}_{\mathsf{RML}}$ is a refined encoding, according to Def. 2.6.*

(2) *Translation $\langle [\![\cdot]\!]_f^g, \psi_{[\![\cdot]\!]_f^g} \rangle$, which maps $\mathcal{L}^{*}_{\pi_{\mathsf{R}}^{\lightning}}$ into $\mathcal{L}_{\mathsf{RML}}$ is a valid encoding according to Def. 2.3.*

Proof. Numeral (1) follows from Thm. 7.9, Thm. 7.10, and Thm. 7.13. Numeral (2) follows from Thm. 7.9, Thm. 7.10, and Thm. 7.12. □

7.4 Timed Patterns Revisited: $[\![\cdot]\!]_f^g$

In this section we present a ReactiveML representation of the timed patterns in § 1.6 using $[\![\cdot]\!]_f^g$. In § 7.4.1 we present an overview of the ideas needed to develop these representations in the encoding. In § 7.4.2 we discuss the request-response timeout pattern; in § 7.4.3, the messages in a time-frame pattern, and in § 7.4.4 we present the action duration pattern in RML. Finally, we present the repeated constraint pattern in ReactiveML (see § 7.4.5).

7.4.1 Overview: Exploiting Compositionality via Decompositions

To use the encoding $[\![\cdot]\!]_f^g$ to represent timed and reactive features we shall follow the same ideas developed in § 4.4.1. Our goal is to show that, similarly to `lcc`, it is possible to use ReactiveML as a foundation for a unified view of message-passing programs in which features such as timed and reactive behavior can be analyzed. Summarizing, there are two key insights for our approach:

(1) *Encoded processes can be used as "code snippets":* Thanks to the compositionality property of our translation (cf. Thm. 7.10), it is possible to use encoded processes as code snippets in RML programs. Moreover, thanks to the operational correspondence property, these code snippets will preserve the behavior inherited from their source $\pi_{\mathsf{R}}^{\lightning}$ terms (cf. Thm. 7.12 and Thm. 7.13). This ensures that the ReactiveML programs created using the translations preserve the desired behavior up to certain conditions that depend on the context in which the snippets are used.

(2) *Process decompositions give more control over the generated programs:* Using a process decomposition function for $\pi_{\mathsf{R}}^{\lightning}$ processes such as the one in Def. 4.51 allows us to gain more control over the code snippets obtained from the encoding. In this way, we can describe timed and reactive features in a more granular way than the descriptions we would obtain by considering exclusively encodings of processes that are not decomposed.

As an example of this approach, let us consider process P_h in (4.3):

$$P_h = (\boldsymbol{\nu} xy)(x\langle \mathsf{REQ}\rangle.x(z).\mathbf{0} \mid y(z).y\langle \mathsf{ACK}\rangle.\mathbf{0})$$

This process can be typed in $\pi_{\mathsf{R}}^{\lightning}$, as there are no races in it.

Now, assume that we would like to represent the fact that the accept-request communication in P_h can only occur whenever some event has been emitted; for example, an scenario for this situation could be a program in which P_h is only enabled after an event *error* has occurred. In this case, we could use the following ReactiveML expression:

$$\texttt{await}\ \mathit{error}\ \texttt{in}\ [\![P_h]\!]_f^g$$

where renaming functions f and g can be obtained as shown in Fig. 7.2. It would also be possible to think of more involved programs. For example, we could also describe a ReactiveML expression in which both the parallel subprocesses of P_h depend on different events to be activated (e.g., error_1 and error_2):

$$D_{\widetilde{x}}[\texttt{await}\ \mathit{error}_1\ \texttt{in}\ [\![x\langle\mathsf{REQ}\rangle.x(z).\mathbf{0}]\!]_f^g \parallel \texttt{await}\ \mathit{error}_2\ \texttt{in}\ [\![y(z).y\langle\mathsf{ACK}\rangle.\mathbf{0}]\!]_f^g]$$

where $\widetilde{x} = x$ and functions f and g can be obtained by following Fig. 7.2.

Because of the operational correspondence property of $[\![\cdot]\!]_f^g$ (cf. Thm. 7.13 and Thm. 7.12), the RML snippets from the encoding will preserve the behavior intended for P_h, provided that the events are present. In this sense, the encoding allow us to represent requirements that may appear in message-passing programs, but are not explicitly representable in $\pi_{\mathsf{R}}^{\lightning}$.

To represent the timed patterns in § 1.6, we shall lift the decomposition $\mathfrak{D}(\cdot)$ in Def. 4.51 to $\pi_{\mathsf{R}}^{\lightning}$. Notice that we can do this without adding more details because there are no differences between the syntax from $\pi_{\mathsf{OR}}^{\lightning}$ and $\pi_{\mathsf{R}}^{\lightning}$ processes.

7.4.2 Request-Response Timeout

The request-response timeout enforces quality of service by requiring a strict deadline on the acknowledgment of request messages (cf. § 1.6). From the server side, the specification of the request-response timeout pattern is:

(a) *Server side:* After receiving a message REQ from A, B must send the acknowledgment ACK within t time units.

Before giving an RML representation we first introduce a $\pi_{\mathsf{R}}^{\lightning}$ process annotated to represent the specification. For this we can reuse P_r in (4.7):

$$P_r = (\boldsymbol{\nu}xy)(\underbrace{x\langle\mathsf{REQ}\rangle.x(z).Q_1 \mid y(z).y\langle\mathsf{ACK}\rangle}_{t}.Q_2)$$

where the time elapsed between the reception of the request and the acknowledgment must not exceed t time units. Similarly, we can also reuse the decomposition in § 4.4.2 to obtain:

$$\mathfrak{D}(P_r) = (\boldsymbol{\nu}\widetilde{u})(\mathfrak{D}_1 \mid \mathfrak{D}_2 \mid \mathfrak{D}_3 \mid \mathfrak{D}_4 \mid R)$$

where $\widetilde{u}$ can be obtained by applying Def. 4.51 and each of the parallel sub-processes

$\mathfrak{D}_i$ $(i \in \{1,2,3,4\})$ is given below:

$$\begin{aligned}
\mathfrak{D}_1 &= c_2(\widetilde{u_x}).x_{1,1}\langle \mathsf{REQ}\rangle.d_3\langle \widetilde{u_x} \setminus x_{1,1}\rangle.\mathbf{0}\\
\mathfrak{D}_2 &= c_3(\widetilde{u_x} \setminus x_{1,1}).x_{1,2}(z\widetilde{z_1}).d_4\langle \widetilde{u_x}z\widetilde{z_1} \setminus x_{1,1}x_{1,2}\rangle.\mathbf{0}\\
\mathfrak{D}_3 &= c_5(\widetilde{u_y}).y_{1,1}(z'\widetilde{z_2}).d_6\langle \widetilde{u_y}z'\widetilde{z_2} \setminus y_{1,1}\rangle.\mathbf{0}\\
\mathfrak{D}_4 &= c_6(\widetilde{u_y}z'\widetilde{z_2} \setminus y_{1,1}).y_{1,2}\langle \mathsf{ACK}\rangle.d_7\langle \widetilde{u_y}z'\widetilde{z_2} \setminus y_{1,1}y_{1,2}\rangle.\mathbf{0}
\end{aligned}$$

To represent timed patterns in RML it is important to understand that the notion of time in synchronous reactive programming languages is logical—i.e., time units in RML are flexible. Therefore, when considering the timing constraint imposed by the pattern we will use events, rather than the time units of RML themselves. We first consider the encoding of $\mathfrak{D}(P_r)$, which thanks to the compositionality property of the encoding (cf. Thm. 7.10) is as follows:

$$[\![\mathfrak{D}(P_r)]\!]^g_f = D_{\widetilde{z}}[[\![\mathfrak{D}_1]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_2]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_3]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_4]\!]^{g_1}_{f_1} \parallel [\![R]\!]^{g_1}_{f_1}]$$

where $D_{\widetilde{z}}[-]$ is as in Def. 2.40, $\widetilde{z}$ can be derived by following the rules in Fig. 7.2. Similarly, the renaming functions f_1 and g_1 can be obtained by following the translation.

We can then build an RML program that represents the request-response timeout pattern by using the preemption construct `do` (e_1) `until` $t \to (e_2)$. Intuitively, this construct indicates that whenever the timeout signal t is present and the internal expression e_1 has not terminated, the failure must be reported. Using the code snippets obtained from $[\![\mathfrak{D}(P_r)]\!]^g_f$ we have:

$$\begin{aligned}
e_1 = D_{\widetilde{z}}[\mathtt{do}\ ([\![\mathfrak{D}_1]\!]^{g_1}_{f_1} \parallel \mathtt{do}\ ([\![\mathfrak{D}_2]\!]^{g_1}_{f_1})\ \mathtt{until}\ t \to (\mathtt{emit}\ \mathit{error}) \parallel [\![\mathfrak{D}_3]\!]^{g_1}_{f_1} \parallel\\
[\![\mathfrak{D}_4]\!]^{g_1}_{f_1} \parallel [\![R]\!]^{g_1}_{f_1})\ \mathtt{until}\ \mathit{error} \to (e_f)]
\end{aligned}$$

This program is obtained by placing the code snippets inside the body of a preemption construct that awaits signal *error*. This signal can only be emitted whenever the preemption construct that contains the code snippet $[\![\mathfrak{D}_2]\!]^{g_1}_{f_1}$ detects the timeout signal t. Expression e_f is a placeholder that represents the RML expression that is executed after a failure. Thanks to the operational correspondence property of the translation (cf. Thm. 7.13 and Thm. 7.12) we can ensure that e_1 preserves the behavior of P_r, provided that the timing constraints are met.

7.4.3 Messages in a Time-Frame

We consider the two variants of this timed pattern (cf.§ 1.6): (a) the first one enforces time intervals between messages, and (b) the second one enforces the overall time-frame in which a number of messages must be sent. We reuse the ideas developed in § 4.4.3. Hence, we consider processes P_{ti} (cf. (4.9)) and P_{to} (cf. (4.10)):

$$P_{ti} = (\boldsymbol{\nu} xy)(x\langle \underbrace{\mathsf{M}_1\rangle.x\langle}_{t_1} \underbrace{\mathsf{M}_2\rangle.x\langle}_{t_1} \underbrace{\mathsf{M}_3\rangle.x\langle}_{t_1} \mathsf{M}_4\rangle.P_x \mid P_y)$$

$$P_{to} = (\boldsymbol{\nu} xy)(\underbrace{x\langle \mathsf{M}_1\rangle.x\langle \mathsf{M}_2\rangle.x\langle \mathsf{M}_3\rangle.x\langle \mathsf{M}_4\rangle}_{t_2}.P_x \mid P_y)$$

As in § 4.4.3, P_{ti} represents the interval pattern and P_{to}, the overall time-frame pattern. The informal requirements for each variant of the pattern are: (a) the participant using endpoint x on P_{ti} can only send messages with *at least* t_1 time units between them, and (b) the participant in endpoint x on P_{to} can send at most *four* messages in a *at most* t_2 time units. We use the same decomposition as in § 4.4.3. Hence,

$$\mathfrak{D}(P_{to}) = \mathfrak{D}(P_{ti}) = (\boldsymbol{\nu}\widetilde{u})(\mathfrak{D}_1 \mid \mathfrak{D}_2 \mid \mathfrak{D}_3 \mid \mathfrak{D}_4 \mid R_x \mid R_y)$$

where processes $\mathfrak{D}_i$ ($i \in \{1,2,3,4\}$) correspond to:

$$\begin{aligned}
\mathfrak{D}_1 &= c_2(\widetilde{u_x}).x_{1,1}\langle \mathsf{M}_1\rangle.d_3\langle \widetilde{u_x} \setminus x_{1,1}\rangle.\mathbf{0}\\
\mathfrak{D}_2 &= c_3(\widetilde{u_x} \setminus x_{1,1}).x_{1,2}\langle \mathsf{M}_2\rangle.d_4\langle \widetilde{u_x} \setminus x_{1,1}x_{1,2}\rangle.\mathbf{0}\\
\mathfrak{D}_3 &= c_4(\widetilde{u_x} \setminus x_{1,1}x_{1,2}).x_{1,3}\langle \mathsf{M}_3\rangle.d_5\langle \widetilde{u_x} \setminus x_{1,1}x_{1,2}x_{1,3}\rangle.\mathbf{0}\\
\mathfrak{D}_4 &= c_5(\widetilde{u_x} \setminus x_{1,1}x_{1,2}x_{1,3}).x_{1,4}\langle \mathsf{M}_4\rangle.d_6\langle \widetilde{u_x} \setminus x_{1,1}x_{1,2}x_{1,3}x_{1,4}\rangle.\mathbf{0}
\end{aligned}$$

and where R_x and R_y correspond to all the processes obtained from the decomposition of P_x and P_y, respectively (cf. Def. 4.51). Now, by applying the encoding (cf. Fig. 7.2) and because of its compositionality property (cf. Thm. 7.10):

$$[\![\mathfrak{D}(P_{to})]\!]^g_f = [\![\mathfrak{D}(P_{ti})]\!]^g_f = D_{\widetilde{z}}[[\![\mathfrak{D}_1]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_2]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_3]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_4]\!]^{g_1}_{f_1} \parallel [\![R_x]\!]^{g_1}_{f_1} \parallel [\![R_y]\!]^{g_1}_{f_1}]$$

where $\widetilde{z}$, f_1, and g_1 can be derived using Fig. 7.2 (they are not needed for this example). The variants of the messages in a time-frame pattern can then be represented as follows:

(a) To represent the interval variant of the messages in a time-frame pattern we make use of the await construct of RML which can be used to suspend the process until a signal is present. Hence, we obtain the following RML expression:

$$\begin{aligned}
e_1 = D_{\widetilde{z}}[&[\![\mathfrak{D}_1]\!]^{g_1}_{f_1} \parallel\\
&\quad \texttt{await}\ t_1\ \texttt{in}\ ([\![\mathfrak{D}_2]\!]^{g_1}_{f_1} \parallel\\
&\qquad \texttt{await}\ t_1\ \texttt{in}\ ([\![\mathfrak{D}_3]\!]^{g_1}_{f_1} \parallel\\
&\qquad\quad \texttt{await}\ t_1\ \texttt{in}\ [\![\mathfrak{D}_4]\!]^{g_1}_{f_1})) \parallel\\
&[\![R_x]\!]^{g_1}_{f_1} \parallel [\![R_y]\!]^{g_1}_{f_1}]
\end{aligned}$$

Above, we make use of a nested sequence of awaiting constructs to represent each delay. In particular, whenever signal t_1 is emitted for the first time, translation $[\![\mathfrak{D}_2]\!]^{g_1}_{f_1}$ becomes active in the next time instant (i.e., $t_1 + 1$). Then, the second await blocks the execution of $[\![\mathfrak{D}_3]\!]^{g_1}_{f_1}$ until t_1 is emitted again. Similarly for $[\![\mathfrak{D}_4]\!]^{g_1}_{f_1}$.

(b) To represent the second variant of the pattern we use the do/until construct. In particular, we place the parallel sub-processes of the encoded decomposition inside the body of a do/until construct that tests for errors, represented by signal *error*. We also use a do/until process to check that the timing constraint is satisfied; if it is not, we emit *error*:

$$\begin{aligned}
e_2 = D_{\widetilde{z}}[&\texttt{do (do (}[\![\mathfrak{D}_1]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_2]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_3]\!]^{g_1}_{f_1} \parallel [\![\mathfrak{D}_4]\!]^{g_1}_{f_1}\texttt{) until}\ t_2 \to (\texttt{emit}\ \mathit{error}) \parallel\\
&[\![R_x]\!]^{g_1}_{f_1} \parallel [\![R_y]\!]^{g_1}_{f_1}\texttt{) until}\ \mathit{error} \to (P_f)]
\end{aligned}$$

In both of the RML programs above, the operational correspondence property of the encoding guarantees that the behavior of the $\pi_{\mathsf{R}}^{\sharp}$ process is preserved (cf. Thm. 7.13 and Thm. 7.12).

7.4.4 *Action Duration*

The action duration timed pattern enforces the time elapsed between two actions of the same participant. We recall (4.12):

$$P_a = (\boldsymbol{\nu} xy)(\underbrace{x\langle \mathtt{M}_1\rangle.x\langle \mathtt{M}_2\rangle}_{t}.P_x \mid P_y)$$

Whose decomposition can be obtained by using Def. 4.51 and contains the following processes:

$$\begin{aligned}\mathfrak{D}_1 &= c_2(\widetilde{u_x}).x_{1,1}\langle \mathtt{M}_1\rangle.d_3\langle \widetilde{u_x} \setminus x_{1,1}\rangle.\mathbf{0}\\ \mathfrak{D}_2 &= c_3(\widetilde{u_x} \setminus x_{1,1}).x_{1,2}\langle \mathtt{M}_2\rangle.d_4\langle \widetilde{u_x} \setminus x_{1,1}x_{1,2}\rangle.\mathbf{0}\end{aligned}$$

Notice that as with the previous decompositions, we assume that R_x and R_y gather all the processes that are obtained from the decomposition of P_x and P_y, respectively. Then, by applying the encoding and its compositionality property (cf. Thm. 7.10), we have that:

$$[\![\mathfrak{D}(P_a)]\!]_f^g = D_{\widetilde{z}}[[\![\mathfrak{D}_1]\!]_{f_1}^{g_1} \parallel [\![\mathfrak{D}_2]\!]_{f_1}^{g_1} \parallel [\![R_x]\!]_{f_1}^{g_1} \parallel [\![R_y]\!]_{f_1}^{g_1}]$$

where $\widetilde{z}$, f_1, and g_1 are obtained as indicated by Fig. 7.2.

To represent this pattern, we make use of do/until expressions again. Intuitively, we let $[\![\mathfrak{D}_1]\!]_{f_1}^{g_1}$ execute unchanged, but we place $[\![\mathfrak{D}_2]\!]_{f_1}^{g_1}$ inside the body of a do/until expression. This ensures that if the communication is not finished before signal t is emitted, an error signal is emitted:

$$\begin{aligned}e_1 = D_{\widetilde{z}}[&\mathtt{do}\ ([\![\mathfrak{D}_1]\!]_{f_1}^{g_1} \parallel \mathtt{do}\ ([\![\mathfrak{D}_2]\!]_{f_1}^{g_1})\ \mathtt{until}\ t \rightarrow (\mathtt{emit}\ e) \parallel [\![R_x]\!]_{f_1}^{g_1} \parallel\\ &[\![R_y]\!]_{f_1}^{g_1})\ \mathtt{until}\ e \rightarrow (P_f)]\end{aligned}$$

Notice that the operational correspondence property of $[\![\cdot]\!]_f^g$ (cf. Thm. 7.13 and Thm. 7.12) ensures that e_1 preserves the behavior of P_a, provided that the timing constraints are satisfied.

7.4.5 *Repeated Constraint*

In § 4.4.5, we argued that $[\![\cdot]\!]$ (cf. Ch. 4) was unable to model the repeated constraint pattern due to the incompatibility of `lcc` with the flexibility of unrestricted types in π (cf. § 2.2 and § 3.1.1). In this section we show that it is possible for RML to model this specification pattern.

Let us recall process P_c in (4.13). This process outputs message $\mathtt{M}_1$ an unbounded amount of times, provided that P_y implements the necessary receptors.

$$P_c = (\boldsymbol{\nu} loop_1 loop_2)(\boldsymbol{\nu} xy)(\underbrace{loop_1\langle loop_1\rangle.\mathbf{0} \mid *\, loop_2(z).(x\langle \mathtt{M}_1\rangle.loop_1\langle z\rangle.\mathbf{0})}_{t} \mid P_y)$$

The first thing to notice is that this process is not a typable $\pi_{\mathsf{R}}^{\lightning}$ process since it induces races. Therefore, we cannot encode this behavior by simply decomposing P_c. Rather, we would like to use the expressiveness of RML to encode the desired behavior. We do this by using the recursion expressions of RML. Intuitively, the behavior of P_c above is that of a process that sends a message constantly, provided that there are receptors. To model such behavior in RML, we could use the encoding $[\![x\langle \mathsf{M}_1\rangle.\mathbf{0}]\!]^g_f$, which represents a single output. Assuming that g and f are correctly defined (cf. Def. 7.2), we could then use the following RML expression:

$$\texttt{let rec process } \mathit{seq}\ \alpha = \texttt{do } (\alpha; \texttt{pause}; (\mathit{seq}\ \alpha)) \texttt{ until } t \rightarrow (P_f) \texttt{ in run } \mathit{seq}\ [\![x\langle \mathsf{M}_1\rangle.\mathbf{0}]\!]^g_f$$

The process above works by generating a sequence of outputs, in which each encoded output has to be executed in sequence with respect to the previous one. Notice that this can be done in ReactiveML because of the sequence operator that allows to execute any pair of RML expressions in sequence. This behavior, coupled with the timing checks that allow the internal body of the recursion to exit in case of a timeout signal t represent the repeated constraint pattern.

8 Encoding aπ in RMLq

This chapter introduces a correct translation from aπ into RMLq. In § 8.1 we present the translation and give intuitions on key formalizations. In § 8.2, we prove that the translation satisfies the static properties required by Def. 2.3. In § 8.3, we prove the operational correspondence property of the translation and summarize the correctness results for the translation, showing that it is both a valid and refined encoding (Def. 2.3 and Def. 2.6, respectively).

8.1 The Translation

In the communication model of aπ, interaction is represented by pairing every endpoint with an output and an input queue. Briefly, the input queue is in charge of receiving messages intended for a given endpoint k, whereas the output queue is in charge of transmitting the messages from k to the input queue of its complementary endpoint $\overline{k}$ (cf. § 3.2 for details).

Our intention is then to represent the communication mechanism of aπ using the queues and big-step semantics introduced for RMLq (cf. § 3.4). To achieve this, and because queues and expression in RMLq are separate entities (i.e., the queues are contained in states, which are not expressions), we require the translation to distinguish between aπ queue processes and non-queue processes (i.e., processes that live in a$\pi^\star$). Rather than using general aπ processes, the new translation will be defined for well-typed programs in aπ (cf. Def. 3.72). The reason is twofold: (1) we are mainly interested in translating well-behaved programs, which in turn ensures well-behaved RMLq programs (in terms of communication between processes) and (2) the distinction between queue and non-queue processes becomes clearer, as presented in Lem. 3.73.

Making an explicit distinction between queue and non-queue processes (i.e., a$\pi^\star$

$$
\begin{aligned}
I(k) &\overset{\text{def}}{=} \texttt{let rec process } I\ \alpha = \\
&\qquad (\texttt{present } ack_o^{\overline{k}}?\,(\texttt{emit } ack_i^{k};\texttt{await } \alpha(y,z)\ \texttt{in}\ (\texttt{put } y\ k_i);\texttt{run } I\ z) : (\texttt{run } I\ \alpha) \\
&\qquad \texttt{in run } I\ k \\
O(k) &\overset{\text{def}}{=} \texttt{let rec process } O\ \alpha = \\
&\qquad\quad \texttt{signal } \alpha', \overline{\alpha'}\ \texttt{in isEmpty } k_o;\texttt{emit } ack_o^{k}; \\
&\qquad\quad (\texttt{present } ack_i^{\overline{k}}?\,(\texttt{emit } \overline{\alpha}\ ((\texttt{pop } k_o), \overline{\alpha'});\texttt{pause};\texttt{run } O\ \alpha') : (\texttt{run } O\ \alpha) \\
&\qquad \texttt{in run } O\ k
\end{aligned}
$$

Figure 8.1: Components of RMLq handler processes (Def. 8.1)

processes) allows us to present a two-layered translation between aπ and RMLq. On one hand, we have the a$\pi^\star$ processes which interact with their local queues instantaneously. These processes can be directly translated into RMLq expressions without further insight. On the other hand, queue processes require a more careful approach. Namely, considering the structure of RMLq configurations, a queue will require two different steps for translation: (1) the actual values inside the aπ queue must be instantiated in a RMLq state, and (2) an expression dealing with queue synchronization must be defined to allow actual communication between processes. These controller processes are denominated *handler process* and are defined below:

Definition 8.1 (Handler Process). Given $\widetilde{k} = \{k_1, \dots, k_n\}$, the handler process $\mathcal{H}(\widetilde{k})$ is defined as:

$$\texttt{signal } \widetilde{ack_i^{k_j}}, \widetilde{ack_o^{k_j}}\ \texttt{in} \prod_{j\in\{1,\dots,n\}} I(k_j) \parallel O(k_j)$$

where $I(k)$ and $O(k)$ are as in Fig. 8.1.

Given an endpoint k, a handler defines processes $I(k)$ and $O(k)$ as its components. Handler components are in charge of managing communication between input and output queues from complementary endpoints. As with $[\![\cdot]\!]^g_f$, endpoints are translated as signals. We use notation $\overline{s}$ to refer to the complementary signal of s and let $\overline{\overline{s}} = s$.

Intuitively, interactions between handler components for complementary endpoints occurs in pairs (i.e., $I(k)$ with $O(\overline{k})$ and $I(\overline{k})$ with $O(k)$). Each pair takes control over one of the signals k or $\overline{k}$ for the communication, avoiding emissions on the same signal. Fig. 8.2 clarifies this by showing the communication direction in a synchronization between handlers for complementary endpoints x and $\overline{x}$.

Each handler component is defined as a recursive expression, where parameter α denotes the signal used for communication exclusively during that instant. In the first iteration of the handler, this signal is named after the endpoint said handler controls k, but this name may be replaced after further iterations. It is important to notice that in defining handler components, endpoint k plays a two-fold role:

(a) First, as mentioned above, k is the name of the signal used in the first iteration of the handler component execution (i.e., $\texttt{run } O\ k$ and $\texttt{run } I\ k$).

$$\begin{tikzcd}
I(x) \arrow[r, bend right, "ack_i^{x}"'] & O(\overline{x}) \arrow[l, bend right, "{ack_o^{\overline{x}}, x}"'] & O(x) \arrow[r, bend left, "{ack_o^{x}, \overline{x}}"] & I(\overline{x}) \arrow[l, bend left, "ack_i^{\overline{x}}"]
\end{tikzcd}$$

Figure 8.2: Handshake direction and signals used in handler components.

(b) Second, k is *fixed* in some of the terms that make up the internal body of the recursion. In particular, name k will be fixed in the names of signals ack_i^k, ack_o^k, $ack_i^{\overline{k}}$, and $ack_o^{\overline{k}}$, as these signals will remain unchanged during the handler execution.

To understand why some of the signals in handler components remain intact, while others change it is important to highlight that, as with RML, signals are not polymorphic in RMLq. Hence, whenever transmitting a value new signals must be created. On the other hand, signals such as ack_i^k, ack_o^k, $ack_i^{\overline{k}}$, and $ack_o^{\overline{k}}$ are pure, in that they do not transmit any value. Thus, they may remain unchanged during the execution of a handler component.

Following the same design decisions as in $[\![\cdot]\!]_f^g$, we let only one synchronization to occur in a single time unit. Assuming complementary endpoints x and $\overline{x}$, interaction starts by attempting a handshake where both $O(x)$ and $I(\overline{x})$ (or vice-versa) must be ready to communicate. The success or failure of a handshake is determined by the presence or absence of *acknowledgment signals* ack_i^x and ack_o^x. Subscripts i and o denote *input* and *output*, respectively.

Assuming that a synchronization can occur (i.e., queue x_o is not empty), process $O(x)$ broadcasts ack_o^x. Then, component $I(\overline{x})$ acknowledges the synchronization by emitting $ack_i^{\overline{x}}$. Then, $O(x)$ creates two signals: x' which it keeps for itself and $\overline{x'}$, intended for its counterpart $I(\overline{x})$. If ready, $O(x)$ sends a pair containing the message stored at the beginning of its output queue ($\mathsf{pop}\ x_o$) and a the fresh signal $\overline{x'}$ to enable further actions. Once the pair is received, the value, now stored in variable y, is enqueued in x_i (i.e., the input queue for x). The process is recursively called in the next instant with the fresh endpoints.

The second component necessary for translating queues into RMLq is properly instantiating the values contained in an aπ queue into a RMLq state. To achieve this, we use the following definition:

Definition 8.2. We define $\delta(\cdot)$ as a function that maps aπ processes into RMLq states:

$$\delta(k[i:\widetilde{m_1}; o:\widetilde{m_2}]) = \{k_i : \widetilde{m_1}, k_o : \widetilde{m_2}\} \quad \delta(P \mid Q) = \delta(P) \cup \delta(Q) \quad \delta((\boldsymbol{\nu} x)P) = \delta(P)$$

and as $\delta(P) = \emptyset$ for every other aπ process.

We are now ready to introduce the translation. Recall the formal languages $\mathcal{L}_{\mathsf{a}\pi}$, and $\mathcal{L}_{\mathsf{RMLq}}$ given in Def. 3.100(6) and Def. 3.102(4), respectively. Then, the translation between these two languages is given below.

Definition 8.3 (Translating aπ into RMLq). Let $\langle (\!|\cdot|\!), \psi_{(\!|\cdot|\!)} \rangle$ be a translation from $\mathcal{L}_{\mathsf{a}\pi}$ into $\mathcal{L}_{\mathsf{RMLq}}$, where:

$$\begin{aligned}
\llbracket x(y).P \rrbracket &\overset{\text{def}}{=} \texttt{let } y = \texttt{pop } x_i \texttt{ in } \llbracket P \rrbracket \\
\llbracket x\langle v\rangle.P \rrbracket &\overset{\text{def}}{=} \texttt{put } x_o\ v; \llbracket P \rrbracket \\
\llbracket x \triangleright \{l_i : P_i\}_{i\in I} \rrbracket &\overset{\text{def}}{=} \texttt{let } y = \texttt{pop } x_i \texttt{ in} \\
\llbracket x \triangleleft l.P \rrbracket &\overset{\text{def}}{=} \texttt{put } x_o\ l; \llbracket P \rrbracket \\
&\qquad\quad \texttt{match } l \texttt{ with } \{l_i : \llbracket P_i \rrbracket\}_{i\in I} \\
\llbracket P \mid Q \rrbracket &\overset{\text{def}}{=} \llbracket P \rrbracket \parallel \llbracket Q \rrbracket \\
\llbracket b?\,(P)\!:\!(Q) \rrbracket &\overset{\text{def}}{=} \texttt{if}\, b\, \texttt{then}\,(\llbracket P \rrbracket)\,\texttt{else}\,(\llbracket Q \rrbracket) \\
\llbracket (\boldsymbol{\nu} x)P \rrbracket &\overset{\text{def}}{=} \texttt{signal } x, \overline{x} \texttt{ in } \llbracket P \rrbracket \\
\llbracket \mu X.P \rrbracket &\overset{\text{def}}{=} \texttt{let rec process } \alpha_X = \llbracket P \rrbracket \texttt{ in run } \alpha_X \\
\llbracket X \rrbracket &\overset{\text{def}}{=} \texttt{pause}\,;\texttt{run } \alpha_X \\
\llbracket \mathbf{0} \rrbracket &\overset{\text{def}}{=} ()
\end{aligned}$$

Figure 8.3: Auxiliary translation from a$\pi^\star$ to RMLq (Def. 8.3).

- $\psi_{(\!|\cdot|\!)}(k) = k$, i.e., every variable in aπ is mapped to the same variable in RMLq.
- $(\!|\cdot|\!)$: aπ $\rightarrow$ RMLq is defined for well-formed aπ programs $C[P,Q]$, which are translated into RMLq configurations as follows:

$$(\!|C[P,Q]|\!) = \langle \texttt{signal } \widetilde{k} \texttt{ in } (\llbracket P \rrbracket \parallel \mathcal{H}(\widetilde{k})) \diamond \delta(Q) \rangle$$

  where $\forall k \in \widetilde{k}.(k \in \mathsf{fv}_\pi(P \mid Q))$; $\llbracket\cdot\rrbracket$: a$\pi^\star$ $\rightarrow$ RMLq is in Fig. 8.3; $\mathcal{H}(\widetilde{k})$ is in Def. 8.1; and $\delta(\cdot)$ is in Def. 8.2.

As mentioned above, $(\!|\cdot|\!)$ is a two-layered translation in which the first layer, $\llbracket\cdot\rrbracket$, deals with a$\pi^\star$ processes, and the second is in charge of the translation of queues.

Two key ideas are at the heart of translation $(\!|\cdot|\!)$: *queues local to processes* and *compositional* (*queue*) *handlers*. The first of these ideas, rooted in the semantics of aπ (cf. Def. 3.2), directly concerns the synchronous hypothesis for synchronous languages. Indeed, since we consider queues k_i and k_o to be local to endpoint k, communication between them must be instantaneous, for such queues should also be local to the process implementing session k.

The second idea is rooted in the fact that queue handlers effectively separate processes/behavior from data/state. As such, it is conceivable to have handlers that have more functionalities than those of $\mathcal{H}(\widetilde{k})$. We conjecture that, because of the compositional nature of the translation in Def. 8.3, these enhanced handlers can then be plugged into the translation.

We have discussed at length the purpose of process handlers and how communication appears in $(\!|\cdot|\!)$. Below, we briefly comment on the first layer of the translation: mapping $\llbracket\cdot\rrbracket$, whose goal is to translate a$\pi^\star$ processes in RMLq expressions:

- The output construct $x\langle v\rangle.P$ is translated as expression:

$$\texttt{put } x_o\ v; \llbracket P \rrbracket$$

  which puts value v into the output queue x_o and continues in the same instant as $\llbracket P \rrbracket$.

- Similarly, input process $x(y).P$ is translated as expression:

$$\texttt{let}\ y = \texttt{pop}\ x_i\ \texttt{in}\ \llbracket P \rrbracket$$

  which binds variable y in $\llbracket P \rrbracket$. Intuitively, the first value in the input queue x_i is popped and assigned to variable y.

- Selection and branching are modeled similarly. The main difference is in the matching construct used to model the actual selection. Notice that different from the $[\![\cdot]\!]^g_f$, the matching construct is evaluated instantaneously, and thus, all selections can be resolved on the same instant the label was received.

- The recursion construct in aπ is modeled as a pause-guarded recursion in RMLq:

$$\texttt{let rec process}\ \alpha_X = \llbracket P \rrbracket\ \texttt{in run}\ \alpha_X$$

  where the occurrences of X has been translated as $\texttt{pause}\,;\texttt{run}\ \alpha_X$. The additional pause goes in accordance with our design decision of dividing communication in instants. An intended consequence of this decision is the avoidance of undesired loops of instantaneous expressions, which may affect the encoding correctness.

- Translations for the conditional, inaction, and parallel composition are as expected.

The full translation $(\!|\cdot|\!)$ creates an RML configuration by composing the RMLq process obtained via $\llbracket\cdot\rrbracket$ with appropriate handlers and with the state obtained from the information in aπ queues.

8.2 Static Correctness

The layered structure of $(\!|\cdot|\!)$ implies that static correctness properties (cf. Def. 2.3) must be proven only for $\llbracket\cdot\rrbracket$, as substitution occurs only on expressions and we do not compose configurations.

Taking into account this consideration, it can be said that proving name invariance and compositionality for $\llbracket\cdot\rrbracket$ is less involved that their counterparts for $[\![\cdot]\!]^g_f$. The statements are given below:

Theorem 8.4 (Name invariance for $\llbracket\cdot\rrbracket$). *Let P, σ, x be an aπ* process, a substitution, and a variable in aπ*, respectively. Then $\llbracket P\sigma \rrbracket = \llbracket P \rrbracket\sigma$.*

Proof. By induction on the structure of P. There are eleven cases that are immediate by applying the IH.

□

Theorem 8.5 (Compositionality for $\llbracket\cdot\rrbracket$). *Let P, σ, x, and $E[\cdot]$ be an aπ* process, a substitution, a variable in aπ*, and an evaluation context as in Def. 3.45, respectively. Then $\llbracket E[P] \rrbracket = \llbracket E \rrbracket\big[\llbracket P \rrbracket\big]$.*

Proof. By induction on the structure of P. There are eleven cases. All of them are immediate. □

8.3 Operational Correspondence

In this section we prove that translation $(\!|\cdot|\!)$ satisfies operational correspondence. In particular, we show that the translation satisfies both the operational correspondence for valid encodings (cf. Def. 2.3), and the operational correspondence for refined encodings (cf. Def. 2.6).

In § 8.3.1 we discuss on how the considerations previously mentioned in § 7.3.1 appear in the new translation with some examples. Next, in § 8.3.2, we prove that $(\!|\cdot|\!)$ is operationally complete, and in § 8.3.3 we prove it is operationally sound.

8.3.1 Considerations

Observe that, contrary to the static correctness results (cf. § 8.2), the operational correspondence results for $(\!|\cdot|\!)$ involve RMLq configurations, as indicated by its semantics.

As with $[\![\cdot]\!]^g_f$, we take into account two important considerations: (1) signal declaration are not persistent, and (2) big-step reductions do not behave as reduction steps. Although these issues were already presented in § 7.3.1, we would like to discuss their presence in RMLq.

Persistence of Signal Declarations

The semantic rules of RMLq closely follow those of RML. Hence, it is not surprising that rule $\lfloor$Sig-Dec$\rfloor$ for RMLq (cf. Fig. 3.10) consumes the signal names, removing the declaration in a single big-step reduction. It is the previous what motivates the use of $\lesssim$ (cf. Def. 3.99). Intuitively, the pre-order $\lesssim$ uses declaration contexts (cf. Def. 2.40) to recover the lost signal names to obtain the full encoding of a configuration. As we will show later, the existence of these declaration contexts can be proven for the target terms induced by $(\!|\cdot|\!)$ (cf. Lem. 8.13).

Synchronous Semantics of RMLq

The semantics of RMLq are synchronous, as those of RML. Therefore, a single big-step reduction for a RMLq configuration obtained from the translation $(\!|\cdot|\!)$ will correspond to several steps from aπ. Since we have decided to consider processes interactions with queues as instantaneous, the correspondence between the semantics of RMLq and aπ it is not that clear. This is the reason to introduce the big-step semantics of aπ in § 3.2.7. The following example allow us to understand how the semantics of RMLq configurations obtained from $(\!|\cdot|\!)$ compared with those of aπ.

Example 8.6. Let us consider the following PI aπ process:

$$\begin{aligned}C[P,\mathcal{K}(x\overline{x})] = (\boldsymbol{\nu} x)(&x\langle v_1\rangle.x\langle v_2\rangle.x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \mid \\ &x[i:\epsilon,o:\epsilon] \mid \overline{x}[i:\epsilon,o:\epsilon])\end{aligned}$$

where:

$$\begin{aligned}P &= x\langle v_1\rangle.x\langle v_2\rangle.x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0}\\ \mathcal{K}(x\overline{x}) &= x[i:\epsilon,o:\epsilon] \mid \overline{x}[i:\epsilon,o:\epsilon]\end{aligned}$$

The translation is as follows:

$$\begin{aligned}(\!|C[P,\mathcal{K}(x\overline{x})]|\!) &= \langle \llbracket C[P,\mathbf{0}]\rrbracket \parallel \mathcal{H}(x\overline{x}) \diamond \delta(\mathcal{K}(x\overline{x}))\rangle \\ &= \langle \llbracket C[P,\mathbf{0}]\rrbracket \parallel \mathcal{H}(x\overline{x}) \diamond \{x_i:\epsilon, x_o:\epsilon, \overline{x}_i:\epsilon, \overline{x}_o:\epsilon\}\rangle\end{aligned}$$

In the process above we have that:

$$\begin{aligned}\llbracket C[P,\mathbf{0}]\rrbracket \parallel \mathcal{H}(x\overline{x}) = \; & \mathtt{signal}\ x,\overline{x}\ \mathtt{in} \\ & \mathtt{put}\ x_o\ v_1; \mathtt{put}\ x_o\ v_2; \mathtt{let}\ y_1 = \mathtt{pop}\ x_i\ \mathtt{in}\ \mathtt{put}\ x_o\ v_3; () \parallel \\ & \mathtt{let}\ y_2 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ \mathtt{let}\ y_3 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ \mathtt{put}\ \overline{x}_o\ y_3; \\ & \mathtt{let}\ y_4 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ () \parallel \\ & I(x) \parallel O(x) \parallel I(\overline{x}) \parallel O(\overline{x}))\end{aligned}$$

and $\Sigma = \delta(\mathcal{K}(x\overline{x})) = \{x_i:\epsilon, x_o:\epsilon, \overline{x_i}:\epsilon, \overline{x_o}:\epsilon\}$. Let us now analyze the big-step reductions of $(\!|C[P,\mathcal{K}(x\overline{x})]|\!)$. In a single big-step reduction we have that:

$$\begin{aligned}(\!|C[P,\mathcal{K}(x\overline{x})]|\!) \vdash\!\!\dashrightarrow & \langle \mathtt{let}\ y_1 = \mathtt{pop}\ x_i\ \mathtt{in}\ \mathtt{put}\ x_o\ v_3; () \parallel \\ & \mathtt{let}\ y_2 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ \mathtt{let}\ y_3 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ \mathtt{put}\ \overline{x}_o\ y_3; \\ & \mathtt{let}\ y_4 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ () \parallel \\ & I'(x) \parallel O'(x) \parallel \\ & I'(\overline{x}) \parallel O'(\overline{x})) \diamond \{x_i:\epsilon, x_o: v_1\cdot v_2, \overline{x}_i:\epsilon, \overline{x}_o:\epsilon\}\rangle = K_1\end{aligned}$$

where $I'(x) = I(x)$ and $I'(\overline{x}) = I(\overline{x})$ up-to unfolding and:

$$\begin{aligned}O'(x) = \; & \mathtt{isEmpty}\ x_o; \mathtt{emit}\ ack_o^x; \\ & (\mathtt{present}\ ack_i^{\overline{x}}?\,(\mathtt{emit}\ \overline{x}\ ((\mathtt{pop}\ x_o), \overline{x'}); \mathtt{pause}\,; \mathtt{run}\ O_1\ x') : (\mathtt{run}\ O_1\ x) \\ O'(\overline{x}) = \; & \mathtt{isEmpty}\ \overline{x}_o; \mathtt{emit}\ ack_o^{\overline{x}}; \\ & (\mathtt{present}\ ack_i^{x}?\,(\mathtt{emit}\ x\ ((\mathtt{pop}\ \overline{x}_o), x'); \mathtt{pause}\,; \mathtt{run}\ O_2\ \overline{x'}) : (\mathtt{run}\ O_2\ \overline{x})\end{aligned}$$

assuming that O_1 and O_2 correspond to the substitutions induced by the unfolding of $O(x)$ and $O(\overline{x})$. We show the complete sequence of big-step reductions below.

$$\begin{aligned}K_1 \vdash\!\!\dashrightarrow & \langle \mathtt{let}\ y_1 = \mathtt{pop}\ x_i\ \mathtt{in}\ \mathtt{put}\ x_o\ v_3; () \parallel \\ & \mathtt{let}\ y_2 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ \mathtt{let}\ y_3 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ \mathtt{put}\ \overline{x}_o\ v_2; \mathtt{let}\ y_4 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ () \parallel \\ & I'(x) \parallel O(x') \parallel I(\overline{x'}) \parallel O'(\overline{x})) \diamond \{x_i:\epsilon, x_o:v_2, \overline{x}_i:v_1, \overline{x}_o:\epsilon\}\rangle = K_2 \\ K_2 \vdash\!\!\dashrightarrow & \langle \mathtt{let}\ y_1 = \mathtt{pop}\ x_i\ \mathtt{in}\ \mathtt{put}\ x_o\ v_3; () \parallel \\ & \mathtt{let}\ y_3 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ \mathtt{put}\ \overline{x}_o\ v_2; \mathtt{let}\ y_4 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ () \parallel \\ & I'(x) \parallel O(x'') \parallel I(\overline{x''}) \parallel O'(\overline{x})) \diamond \{x_i:\epsilon, x_o:\epsilon, \overline{x}_i:v_2, \overline{x}_o:\epsilon\}\rangle = K_3 \\ K_3 \vdash\!\!\dashrightarrow & \langle \mathtt{let}\ y_1 = \mathtt{pop}\ x_i\ \mathtt{in}\ \mathtt{put}\ x_o\ v_3; () \parallel \mathtt{let}\ y_4 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ () \parallel \\ & I'(x) \parallel O(x'') \parallel I(\overline{x''}) \parallel O'(\overline{x})) \diamond \{x_i:\epsilon, x_o:\epsilon, \overline{x}_i:\epsilon, \overline{x}_o:v_2\}\rangle = K_4 \\ K_4 \vdash\!\!\dashrightarrow & \langle \mathtt{let}\ y_1 = \mathtt{pop}\ x_i\ \mathtt{in}\ \mathtt{put}\ x_o\ v_3; () \parallel \mathtt{let}\ y_4 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ () \parallel \\ & I(x') \parallel O'(x'') \parallel I'(\overline{x''}) \parallel O(\overline{x'})) \diamond \{x_i:v_2, x_o:\epsilon, \overline{x}_i:\epsilon, \overline{x}_o:\epsilon\}\rangle = K_5 \\ K_5 \vdash\!\!\dashrightarrow & \langle () \parallel \mathtt{let}\ y_4 = \mathtt{pop}\ \overline{x}_i\ \mathtt{in}\ () \parallel\end{aligned}$$

$$
\begin{aligned}
& \qquad I'(x') \parallel O'(x'') \parallel I'(\overline{x''}) \parallel O'(\overline{x'})) \diamond \{x_i : \epsilon, x_o : v_3, \overline{x}_i : \epsilon, \overline{x}_o : \epsilon\}\rangle = K_6 \\
K_6 & \dashrightarrow \langle () \parallel \texttt{let } y_4 = \texttt{pop } \overline{x}_i \texttt{ in } () \parallel \\
& \qquad I(x') \parallel O(x''') \parallel I'(\overline{x'''}) \parallel O'(\overline{x'})) \diamond \{x_i : \epsilon, x_o : \epsilon, \overline{x}_i : v_3, \overline{x}_o : \epsilon\}\rangle = K_7 \\
K_7 & \dashrightarrow \langle () \parallel () \parallel I'(x') \parallel O'(x''') \parallel I'(\overline{x'''}) \parallel O'(\overline{x'})) \diamond \{x_i : \epsilon, x_o : \epsilon, \overline{x}_i : v_3, \overline{x}_o : \epsilon\}\rangle
\end{aligned}
$$

where x' and $\overline{x'}$, x'' and $\overline{x''}$, and x''' and $\overline{x'''}$ are fresh signals created by the output handler components.

In the sequence above, the first big-step reduction (i.e., K to K_1) puts v_1 and v_2 in queue x_o. Next, in the big-step reduction from K_1 to K_2 only one interaction occurs: process $O'(x)$ communicates v_1 by popping it from the beginning of queue x_o and sending it via signal $\overline{x}$ to its complementary process $I'(\overline{x})$, which puts it in $\overline{x}_i$. Observe that only one synchronization is allowed between a pair of complementary handlers during each instant.

The big-step reduction from K_2 to K_3 executes two actions. First, value v_1 is popped from $\overline{x}_i$ and then value v_2 is communicated to queue $\overline{x}_i$. In the big-step reduction from K_3 to K_4, value v_2 is popped from $\overline{x}_i$ and stored in y_3; then, v_2 is put in queue $\overline{x}_o$. In the big-step reduction from K_4 to K_5, v_2 is communicated to x_i. Next, in the big-step reduction that goes from from K_5 to K_6, v_2 is popped from queue x_i and value v_3 is put into the output queue x_o. The next two big-step reductions finish the execution of the RMLq process.

The previous big-step reduction sequence, is equivalent to the following reduction sequence in aπ.

$$
\begin{aligned}
C[P, \mathcal{K}(x\overline{x})] & \longrightarrow_{\mathsf{A}}^2 C[P_1, x[i : \epsilon, o : v_1 \cdot v_2] \mid \overline{x}[i : \epsilon, o : \epsilon]] \\
& \longrightarrow_{\mathsf{A}} C[P_2, x[i : \epsilon, o : v_2] \mid \overline{x}[i : v_1, o : \epsilon]] \\
& \longrightarrow_{\mathsf{A}}^2 C[P_3, x[i : \epsilon, o : \epsilon] \mid \overline{x}[i : v_2, o : \epsilon]] \\
& \longrightarrow_{\mathsf{A}}^2 C[P_4, x[i : \epsilon, o : \epsilon] \mid \overline{x}[i : \epsilon, o : v_2]] \\
& \longrightarrow_{\mathsf{A}} C[P_5, x[i : v_2, o : \epsilon] \mid \overline{x}[i : \epsilon, o : \epsilon]] \\
& \longrightarrow_{\mathsf{A}}^2 C[P_6, x[i : \epsilon, o : v_3] \mid \overline{x}[i : \epsilon, o : \epsilon]] \\
& \longrightarrow_{\mathsf{A}} C[P_7, x[i : \epsilon, o : \epsilon] \mid \overline{x}[i : v_3, o : \epsilon]] \\
& \longrightarrow_{\mathsf{A}} C[\mathbf{0}, x[i : \epsilon, o : v_3] \mid \overline{x}[i : \epsilon, o : \epsilon]]
\end{aligned}
$$

where:

$$
\begin{aligned}
& P_1 = P_2 = x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_2).\overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \\
& P_3 = x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_3).\overline{x}\langle y_3\rangle.\overline{x}(y_4).\mathbf{0} \\
& P_4 = P_5 = x(y_1).x\langle v_3\rangle.\mathbf{0} \mid \overline{x}(y_4).\mathbf{0} \\
& P_6 = \mathbf{0} \mid \overline{x}(y_4).\mathbf{0}
\end{aligned}
$$

An important thing to notice is that this reduction sequence cannot be succinctly expressed using the usual semantics of aπ. This illustrates the need for a big-step semantics for aπ. △

Proof Outline

Similarly as with $[\![\cdot]\!]_f^g$, the main challenge for the operational correspondence proof lies in the fact that the behavioral semantics of RMLq mimic several reduction steps in aπ, and therefore goes faster. Thus, we must prove that our translation is a valid encoding (cf. Def. 2.3) with respect to the big-step semantics in § 3.2.7, and a refined encoding (cf. Def. 2.6) with respect to the reduction semantics in Fig. 3.5. The operational correspondence statements follow:

Theorem 8.7 (Valid Operational Correspondence). *Let P be a well-typed* aπ *program. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \ggg\!\!\to^* Q$, it holds that $(\!|P|\!) \vdash\!\!\dashrightarrow^* \lesssim (\!|Q|\!)$.*
2. **Soundness:** *For every* RMLq *process T such that $(\!|P|\!) \vdash\!\!\dashrightarrow^* T$, there exists Q such that $P \ggg\!\!\to^* Q$ and $T \lesssim (\!|Q|\!)$.*

Theorem 8.8 (Refined Operational Correspondence). *Let P be a well-typed* aπ *program. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \longrightarrow_{\mathsf{A}}^* Q$, then there exists Q' such that $(\!|P|\!) \vdash\!\!\dashrightarrow^* \lesssim (\!|Q'|\!)$ and $Q \longrightarrow_{\mathsf{A}}^* Q'$.*
2. **Soundness:** *For every* RMLq *process T such that $[\![P]\!]^g_f \vdash\!\!\dashrightarrow^* T$, there exists Q such that $P \longrightarrow_{\mathsf{A}}^* Q$ and $T \lesssim (\!|Q|\!)$.*

Intuitively, the relation existing between these two theorems can be used as we did in § 7.3: Thm. 8.8 can be derived as a consequence from the semantic correspondence property (cf. Lem. 3.86) and Thm. 8.7.

Moreover, notice that the formal language used in the translation (cf. Def. 8.3) require the use of pre-order $\lesssim$ (cf. Def. 3.99). Similarly, we show that $(\!|\cdot|\!)$ induces a simplified notion of initialized translations (i.e., analogous to Def. 7.16), only characterized by process handler components $I(\cdot)$ and $O(\cdot)$ (cf. Def. 8.10). In this sense, we must prove some basic properties regarding process handlers components (cf. Lem. 8.11 and Lem. 8.12). Briefly, Lem. 8.11 formalizes characterizes the syntactic shape of process handlers, and Lem. 8.12 proves the existence of the signal declaration contexts required to recover the initial handler; this result is used in both soundness and completeness proofs. The proof outline for operational correspondences follows:

Operational Completeness: The operational completeness property is comprised of two main results: the valid completeness result (cf. Thm. 8.15) and a refined completeness result (cf. Thm. 8.16). Thm. 8.16 is obtained from the composition of Thm. 8.15 and Lem. 3.86(2). Proving Thm. 8.15 is a bit more delicate. The lemma heavily relies on Lem. 8.14, which provides all the necessary components to prove completeness: (1) it proves that the information of queues is preserved by the translation, and (2) it proves the existence of the necessary signal declaration contexts used to validate $\lesssim$ and shows that the translation of a$\pi^\star$ processes is preserved.

The proof of Lem. 8.14 consists of a series of nested inductions: first, on the length of the big-step reduction $P \ggg\!\!\to^* Q$. Then, in the inductive step (i.e., big-step reduction $P \ggg\!\!\to^* Q_0 \ggg\!\!\to Q$), we apply simultaneous induction on the length r_1 and r_2 of reductions $Q_0 \rightarrowtail Q_0' \rightharpoonup^* Q_0'' \not\rightharpoonup$. All the generated cases then proceed by either induction on the aπ process or directly by applying the definitions.

Operational Soundness: The proof of operational soundness is very similar to the one above. There are two results: a valid soundness (cf. Thm. 8.17) and a more general refined soundness (Thm. 8.18). Briefly, Thm. 8.18 follows from Lem. 3.86(1) and Lem. 8.13. As with completeness, Lem. 8.13 is the cornerstone for proving Thm. 8.17. It provides the same assurances given by Lem. 8.14—i.e., it shows that RMLq big-step reductions can be mimicked by the big-step semantics of aπ.

The proof of Lem. 8.13 proceeds similarly to Lem. 8.14. We proceed by induction on the length of the big-step reduction $(\!|P|\!) \dashrightarrow^* T$. We then proceed by induction on the structure of the processes composing T.

Auxiliary Results

We start by defining target terms for our translation:

Definition 8.9 (Target Terms). We define *target terms* as the set of RMLq configurations that are induced by the translation $(\!|\cdot|\!)$ of well-typed aπ programs and is closed under $\dashrightarrow$: $\{S \mid [\![P]\!]^g_f \longmapsto S$ and $\vdash_P\}$. We shall use $S, S', T, T, \ldots$ to range over target terms.

We define the notion of *initialized handler component*; the analogous of Def. 7.16 for $(\!|\cdot|\!)$. Notice that the notion of initialized translations induced by $(\!|\cdot|\!)$ is simpler than the one for $[\![\cdot]\!]^g_f$. This occurs as only handler components $I(\cdot)$ and $O(\cdot)$ generate intermediate processes in the translation.

Definition 8.10 (Initialized Handler Component). Let $I(x)$ and $O(x)$, be handler components for channel x as in Fig. 8.1. The initialized handler components, written $init(I(x))$ and $init(O(x))$, are defined below:

$$\begin{aligned}
&init(I(x)) = I(x)\\
&init(O(x)) = \texttt{isEmpty}\ x_o; \texttt{emit}\ ack^x_o;\\
&\qquad\qquad(\texttt{present}\ ack^{\overline{x}}_i ? (\texttt{emit}\ \overline{x}\ ((\texttt{pop}\ x_o), \overline{x'}); \texttt{pause}\,; \texttt{run}\ O_1) : (\texttt{run}\ O_2)
\end{aligned}$$

where O_1 denotes the unfolding of $O(x)$ in $O\ x'$ and O_2 denotes the unfolding $O(x)$ in $O\ x$.

We proceed to clarify the execution of handler processes. As we show below, there are two possible behaviors for them: (1) if a synchronization between the handlers of complementary processes is possible, then synchronization occurs and the handlers are now executed on fresh signals, or (2) if no synchronization is possible, then the process handler component reduces to an initialized handler component.

Lemma 8.11. *Let x and $\widetilde{x}$ be complementary endpoints. Then, the following holds:*

1. *If* $K = \langle \texttt{signal}\ x, \overline{x}, ack^x_i, ack^x_o, ack^{\overline{x}}_i, ack^{\overline{x}}_o\ \texttt{in}\ I(x) \parallel O(\overline{x}) \diamond x_i : \widetilde{h_1}, \overline{x_o} : v \cdot \widetilde{h_2}\rangle$ *then* $K \dashrightarrow \langle I(x') \parallel O(\overline{x'}) \diamond x_i : \widetilde{h_1} \cdot v, \overline{x_o} : \widetilde{h_2}\rangle = K'$

2. *If* $K = \langle \texttt{signal}\ x, \overline{x}, ack^x_i, ack^x_o, ack^{\overline{x}}_i, ack^{\overline{x}}_o\ \texttt{in}\ I(x) \parallel O(\overline{x}) \diamond x_i : \widetilde{h_1}, \overline{x_o} : \epsilon\rangle$ *then*

$$K \dashrightarrow \langle init(I(x)) \parallel init(O(\overline{x})) \diamond x_i : \widetilde{h_1}, \overline{x_o} : \epsilon\rangle = K' \dashrightarrow^* K'$$

Proof. We prove each item below:

1. Using the definitions in Fig. 8.1 and the semantics in Fig. 3.9, Fig. 3.10, and Fig. 3.11. Consider configuration $K = \langle \texttt{signal}\ ack_i^x, ack_o^x, ack_i^{\overline{x}}, ack_o^{\overline{x}}\ \texttt{in}\ I(x) \parallel O(\overline{x}) \diamond x_i : \widetilde{h_1}, \overline{x_o} : v \cdot \widetilde{h_2}\rangle$. We have the following derivation using the rules in Fig. 3.9, Fig. 3.10, and Fig. 3.11. The first two rules applied are $\lfloor$Sig-Dec$\rfloor$ and $\lfloor$L-Par$\rfloor$:

$$\dfrac{\dfrac{\begin{array}{c}\lfloor\text{L-Done}\rfloor, \lfloor\text{Recur}\rfloor, \lfloor\text{Sig-P}\rfloor, \lfloor\text{L-Done}\rfloor\\ \lfloor\text{Emit}\rfloor, \lfloor\text{DU-P}\rfloor, \lfloor\text{L-Done}\rfloor, \lfloor\text{Put-Q}\rfloor\end{array}}{\langle I(x) \diamond x_i : \widetilde{h_1}\rangle \vdash\!\!\dashrightarrow \langle I(x) \diamond x_i : \widetilde{h_1} \cdot v\rangle} \quad \dfrac{\begin{array}{c}\lfloor\text{L-Done}\rfloor, \lfloor\text{Recur}\rfloor, \lfloor\text{Sig-Dec}\rfloor, \lfloor\text{L-Done}\rfloor,\\ \lfloor\text{NEmpty}\rfloor, \lfloor\text{L-Done}\rfloor, \lfloor\text{Emit}\rfloor, \lfloor\text{Sig-P}\rfloor,\\ \lfloor\text{L-Done}\rfloor, \lfloor\text{Emit}\rfloor, \lfloor\text{Pop-Q}\rfloor, \lfloor\text{Pause}\rfloor\end{array}}{\langle O(\overline{x}) \diamond \overline{x_o} : v \cdot \widetilde{h_2}\rangle \vdash\!\!\dashrightarrow \langle O(\overline{x}) \diamond \overline{x_o} : \widetilde{h_2}\rangle}}{K \vdash\!\!\dashrightarrow \langle I(x) \parallel O(\overline{x}) \diamond x_i : \widetilde{h_1} \cdot v, \overline{x_o} : \widetilde{h_2}\rangle}$$

2. Using the definitions in Fig. 8.1 and the semantics in Fig. 3.9, Fig. 3.10, and Fig. 3.11. Consider configuration $K = \langle \texttt{signal}\ ack_i^x, ack_o^x, ack_i^{\overline{x}}, ack_o^{\overline{x}}\ \texttt{in}\ I(x) \parallel O(\overline{x}) \diamond x_i : \widetilde{h}, \overline{x}_o : \epsilon\rangle$. The derivation is as follows (see Fig. 3.9). The first two rules applied are $\lfloor$Sig-Dec$\rfloor$ and $\lfloor$L-Par$\rfloor$:

$$\dfrac{\dfrac{\lfloor\text{L-Done}\rfloor, \lfloor\text{Recur}\rfloor, \lfloor\text{Sig-NP}\rfloor}{\langle I(x) \diamond x_i : \widetilde{h}\rangle \vdash\!\!\dashrightarrow \langle init(I(x)) \diamond x_i : \widetilde{h}\rangle} \quad \dfrac{\begin{array}{c}\lfloor\text{L-Done}\rfloor, \lfloor\text{Recur}\rfloor, \lfloor\text{Sig-Dec}\rfloor,\\ \lfloor\text{L-Par}\rfloor, \lfloor\text{Empty}\rfloor\end{array}}{\langle O(\overline{x}) \diamond \overline{x}_o : \epsilon\rangle \vdash\!\!\dashrightarrow \langle init(O(\overline{x})) \diamond \overline{x}_o : \epsilon\rangle}}{K \vdash\!\!\dashrightarrow \langle init(I(x)) \parallel init(O(\overline{x})) \diamond x_i : \widetilde{h}, \overline{x}_o : \epsilon\rangle}$$

   Furthermore, it can be shown that:

$$\begin{aligned}&\langle init(I(x)) \parallel init(O(\overline{x})) \diamond x_i : \widetilde{h}, \overline{x}_o : \epsilon\rangle\\ &\vdash\!\!\dashrightarrow^* \langle init(I(x)) \parallel init(O(\overline{x})) \diamond x_i : \widetilde{h}, \overline{x}_o : \epsilon\rangle\end{aligned}$$

□

From the previous statement, it is possible to deduce that initialized output handler components are missing signal declarations to reconstruct the process in Fig. 8.1. The following result generalizes this fact:

Lemma 8.12. *Let $\mathcal{H}(\widetilde{k})$ be a process handler as in Def. 8.1. Then, if $\langle \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k}) \diamond \Sigma\rangle \vdash\!\!\dashrightarrow^* \langle T \diamond \Sigma'\rangle$ for some Σ', it holds that:*

1. *$T = T_1 \parallel \cdots \parallel T_n$ with $n \geq 1$*
2. *There exists $D_{\widetilde{z}}, D_{\widetilde{z_1}}, \ldots, D_{\widetilde{z_n}}$ such that*

$$D_{\widetilde{z}}[D_{\widetilde{z_1}}[T_1] \parallel \cdots \parallel D_{\widetilde{z_n}}[T_n]] \equiv_{\mathsf{R}} \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k})$$

Proof. By induction on the length m of $\widetilde{k}$. The base case is $m = 2$ since by typing our translation will not allow handler processes with a single channel. For details see App. E.1. □

The following auxiliary result is the cornerstone for ensuring that translation $(\!|\cdot|\!)$ is sound. In a sense, the statement below breaks down the necessary elements for proving the soundness of our translation.

Lemma 8.13. *For every well-typed* aπ *program* $C[P,Q] = (\boldsymbol{\nu}\widetilde{k})(P \parallel Q)$ *the following holds: If* $(\!|C[P,Q]|\!) \longmapsto^* K$, *then* $K = \langle T_1 \parallel \cdots \parallel T_n \parallel T_{n+1} \parallel \cdots \parallel T_m \diamond \Sigma\rangle$, $1 \leq n \leq m$, *where:*

1. *There exists* R *such that* $P \ggg\!\!\rightarrow^* R = C[P',Q']$ *and* $\Sigma = \delta(Q')$.
2. *There exist contexts* $D_{\widetilde{z}}, D_{\widetilde{z_{n+1}}}, \ldots, D_{\widetilde{z_m}}$ *such that:* (a) $D_{\widetilde{z}}[D_{\widetilde{z_{n+1}}}[T_{n+1}] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]] \equiv_{\mathsf{R}} \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k})$, *and* (b) $T_1 \parallel \cdots \parallel T_n = [\![P']\!]$.
3. *There exist* $\widetilde{s}$ *and* $\widetilde{s'}$ *such that* $\widetilde{z} = \widetilde{s}\widetilde{s'}$ *and* $\langle D_{\widetilde{s}}[T_1 \parallel \cdots \parallel T_n \parallel D_{\widetilde{s'}}[D_{\widetilde{z_{n+1}}}[T_{n+1}] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]]] \diamond \Sigma\rangle \equiv_{\mathsf{R}} (\!|C[P',Q']|\!)$.

Proof. By induction on the length m of the big-step reduction $(\!|C[P,Q]|\!) \longmapsto^* K$. The base case is immediate. For details on the inductive case see App. E.1. □

We now prove a statement that provides all the necessary ingredients for demonstrating that our translation is complete. This statements shows that for the translation of every well-typed aπ program it is possible to obtain a configuration which, using the corresponding signal declaration environments, corresponds to the translation of a process reachable from the initial aπ program.

Lemma 8.14. *For every well-typed* aπ *program* $C[P,Q] = (\boldsymbol{\nu}\widetilde{k})(P \parallel Q)$ *the following holds: If* $C[P,Q] \ggg\!\!\rightarrow^* C[P',Q']$, *then there exists* K *such that* $(\!|C[P,Q]|\!) \longmapsto^* K$ *where:*

1. $K = \langle T_1 \parallel \cdots \parallel T_n \parallel T_{n+1} \parallel \cdots \parallel T_m \diamond \Sigma\rangle$, $1 \leq n \leq m$ *and* $\Sigma = \delta(Q')$.
2. *There exist contexts* $D_{\widetilde{z}}, D_{\widetilde{z_{n+1}}}, \ldots, D_{\widetilde{z_m}}$ *such that:* (a) $D_{\widetilde{z}}[D_{\widetilde{z_{n+1}}}[T_{n+1}] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]] \equiv_{\mathsf{R}} \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k})$, *and* (b) $T_1 \parallel \cdots \parallel T_n = [\![P']\!]$.
3. *There exist* $\widetilde{s}$ *and* $\widetilde{s'}$ *such that* $\widetilde{z} = \widetilde{s}\widetilde{s'}$ *and* $\langle D_{\widetilde{s}}[T_1 \parallel \cdots \parallel T_n \parallel D_{\widetilde{s'}}[D_{\widetilde{z_{n+1}}}[T_{n+1}] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]]] \diamond \Sigma\rangle \equiv_{\mathsf{R}} (\!|C[P',Q']|\!)$.

Proof. By induction on the length of big-step reduction $C[P,Q] \ggg\!\!\rightarrow^* C[P',Q']$. The base case is immediate. We show the inductive step.

Assume that $C[P,Q] \ggg\!\!\rightarrow^* C[P_0,Q_0] \ggg\!\!\rightarrow C[P',Q']$. By IH1 we have that the statement holds for $C[P_0,Q_0]$. We then analyze the big-step reduction $C[P_0,Q_0] \ggg\!\!\rightarrow C[P',Q']$. By Def. 3.81, we have that:

$$C[P_0,Q_0] \rightarrowtail C[P_0,Q_0'] \rightharpoonup^* C[P_0',Q_0''] \not\rightharpoonup$$

where $P' = \mathsf{unm}(P_0')$ and $Q' = \mathsf{unm}(Q_0'')$. By Def. 3.78 we know that $\rightarrowtail$ corresponds to several reductions (i.e., $\rightharpoonup^*$). Thus, we proceed by simultaneous induction on the lengths r_1 and r_2 of $\rightarrowtail$ and $\rightharpoonup^*$.

Base Case: For the base case we prove the following three cases:

$r_1 = 0 \wedge r_2 = 0$: Immediate, as it corresponds to the base case for the initial induction.

$r_1 > 0 \wedge r_2 = 0$**:** In this case, we have that:

$$C[P_0, Q_0] \rightarrowtail C[P_0, Q_0'] \not\rightarrowtail$$

which means there are only queue synchronizations. This means that P_0 cannot contain processes making outputs or selections at top-level, as these are always enabled. Thus, using the rules in Fig. 3.9, Fig. 3.10, and Fig. 3.11 we can show that:

$$(\!|C[P_0, Q_0]|\!) \dashrightarrow T_1 \parallel T_2 \parallel \cdots \parallel T_m$$

with $m \geq 1$ and $T_1 = (\!|P_0|\!)$ and we can conclude by presenting an argument similar to the one for Lem. 8.13.

$r_1 = 0 \wedge r_2 > 0$**:** This case assumes that there are no synchronizations between queues. We proceed then by induction on the structure of P_0, building derivations similarly to Lem. 8.13.

Inductive Step: The inductive step corresponds to the case where $r_1 > 0$ and $r_2 > 0$. This case follows by applying the IH for both $\rightarrowtail$ and $\ggg\!\rightarrow$. Then, the results can be merged to obtain the desired RMLq big-step reduction.

□

8.3.2 Operational Completeness

Using the previous result we show that our translation is complete. First, we show that completeness holds the big-step semantics for aπ:

Theorem 8.15 (Valid Completeness for $(\!|\cdot|\!)$). *For every well-typed aπ program P, and for every process Q, if $P \ggg\!\rightarrow^* Q$ then there exists T such that $(\!|P|\!) \dashrightarrow^* T$ and $T \lesssim (\!|Q|\!)$.*

Proof. Follows as direct consequence of Lem. 8.14. Notice that $T \lesssim (\!|Q|\!)$ follows from the existence of signal declaration environments shown in Lem. 8.14. □

We can now then prove a more general completeness statement for our translation.

Theorem 8.16 (Refined Completeness for $(\!|\cdot|\!)$). *For every well-typed aπ program P and every process Q, if $P \longrightarrow_{\mathsf{A}}^* Q$ then there exists a RMLq process T and a aπ process Q' such that $(\!|P|\!) \dashrightarrow^* T$, $Q \longrightarrow_{\mathsf{A}}^* Q'$, and $T \lesssim (\!|Q'|\!)$.*

Proof. This is a consequence of Thm. 8.15 and Lem. 3.86(2). □

8.3.3 Operational Soundness

The statement of soundness can be proven as it is. However, aiming to keep the symmetry in the operational correspondence statement, we first prove soundness for the big-step semantics of aπ. Then, as a consequence, the more general soundness statement holds.

Theorem 8.17 (Valid Soundness for $(\!|\cdot|\!)$). *For every well-typed aπ program P and every process Q, if $(\!|P|\!) \dashrightarrow^* T$ then there exist Q such that $P \ggg\!\rightarrow^* Q$ and $T \lesssim (\!|Q|\!)$.*

Proof. Follows as a direct consequence of Lem. 8.13. Notice that $T \lesssim (\!|Q|\!)$ follows by proving the existence of the signal declaration environments as in Lem. 8.13. □

Theorem 8.18 (Refined Soundness for $(\!|\cdot|\!)$). *For every well-typed $\mathsf{a}\pi$ program P and every process Q, if $(\!|P|\!) \dashrightarrow^{*} T$ then there exist Q such that $P \longrightarrow_{\mathsf{A}}^{*} Q$ and $T \lesssim (\!|Q|\!)$.*

Proof. Follows from the composition of Lem. 3.86(1) and Thm. 8.17. □

We conclude this section by proving that $(\!|\cdot|\!)$ is an encoding, following Def. 2.3 and Def. 2.6. First we state operational correspondence:

Theorem 8.7 (Valid Operational Correspondence). *Let P be a well-typed $\mathsf{a}\pi$ program. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \ggg\!\!\rightarrow^{*} Q$, it holds that $(\!|P|\!) \dashrightarrow^{*}\lesssim (\!|Q|\!)$.*
2. **Soundness:** *For every RMLq process T such that $(\!|P|\!) \dashrightarrow^{*} T$, there exists Q such that $P \ggg\!\!\rightarrow^{*} Q$ and $T \lesssim (\!|Q|\!)$.*

Proof. Follows directly from Thm. 8.15 and Thm. 8.17. □

Theorem 8.8 (Refined Operational Correspondence). *Let P be a well-typed $\mathsf{a}\pi$ program. Then, the following properties hold:*

1. **Completeness:** *For every Q such that $P \longrightarrow_{\mathsf{A}}^{*} Q$, then there exists Q' such that $(\!|P|\!) \dashrightarrow^{*}\lesssim (\!|Q'|\!)$ and $Q \longrightarrow_{\mathsf{A}}^{*} Q'$.*
2. **Soundness:** *For every RMLq process T such that $[\![P]\!]_{f}^{g} \dashrightarrow^{*} T$, there exists Q such that $P \longrightarrow_{\mathsf{A}}^{*} Q$ and $T \lesssim (\!|Q|\!)$.*

Proof. Follows directly from Thm. 8.16 and Thm. 8.18. □

Theorem 8.19 (RMLq Encodes $\mathsf{a}\pi$). *Consider the formal languages $\mathcal{L}_{\mathsf{a}\pi}$, $\mathcal{L}_{\mathsf{a}\pi}^{*}$ as presented in Def. 3.100(5,6) and $\mathcal{L}_{\mathsf{RMLq}}$ as it was defined in Def. 3.102(4). Then, the following holds:*

(1) *Translation $\langle (\!|\cdot|\!), \psi_{(\!|\cdot|\!)} \rangle$, which maps $\mathcal{L}_{\mathsf{a}\pi}$ into $\mathcal{L}_{\mathsf{RMLq}}$ is a refined encoding (cf. Def. 2.6).*

(2) *Translation $\langle (\!|\cdot|\!), \psi_{(\!|\cdot|\!)} \rangle$, which maps $\mathcal{L}_{\mathsf{a}\pi}^{*}$ into $\mathcal{L}_{\mathsf{RMLq}}$ is a valid encoding (cf. Def. 2.3).*

Proof. Numeral (1) follows from Thm. 8.4, Thm. 8.5, and Thm. 8.8. Numeral (2) follows from Thm. 8.4, Thm. 8.5, and Thm. 8.7. □

9 Conclusions and Related Work

In this chapter we present the overall conclusions and related work for the two translations presented in Part III. In § 9.1 conclude by summarizing our work and discussing some of the most interesting parts. In § 9.2 we give related work.

9.1 Concluding Remarks

We have presented two *encodings* of session-based π-calculi into *synchronous reactive programming*, a programming paradigm that naturally models timed and reactive behavior. We have proven that our encodings are correct, and therefore, we argue that they can be used to reason about message-passing programs.

The first encoding, in Ch. 7, translates a session π-calculus without races (cf. § 3.1.2) into ReactiveML (cf. § 2.4), a synchronous reactive programming language. An important characteristic of this encoding is that it yields executable ReactiveML programs and therefore, we believe it can be used in practical settings. It is important to mention that to generate actual implementations, the encoding must be used as a code snippet that can be inserted in ReactiveML implementations. Hence, given a well-typed $\pi_{\mathsf{R}}^{\lightning}$ program P we can use its encoding to declare a ReactiveML process:

$$\texttt{let process}\ \mathit{model} = [\![P]\!]^{g}_{f}\ \texttt{in run}\ \mathit{model}$$

which can then be compiled and run. Moreover, given the correctness properties proven for $[\![\cdot]\!]^{g}_{f}$, it is expected that the models obtained by the previous procedure are correctly executed by the ReactiveML compiler. Another highlight of the first encoding is the fact that it can be used to generate *enhanced translations* in which we can *extend* the capabilities of $\pi_{\mathsf{R}}^{\lightning}$ (cf. § 7.4). We conjecture that using this feature in particular, can enable the integration of session-based concurrency in actual RML programs featuring declarative, reactive, timed, and contextual behavior.

The second encoding has shown that it is also possible to encode an *asynchronous* session calculus into a synchronous reactive programming language. In particular, this encoding shows that it is more natural to consider an asynchronous semantics for modeling session-based concurrency in SRP. Notice that since signal emission is a non-blocking construct, a non-blocking output construct can be more faithfully represented than a synchronous one. In this sense, the operational correspondence results obtained for the second encoding were derived much more naturally.

It is also important to note that the most important challenge in this encoding arises from the fact that the semantics of ReactiveML are intrinsically different from those of process calculi such as $\pi^{\lightning}_{\mathsf{R}}$ and $\mathsf{a}\pi$. Indeed, the *synchronous*[1] big-step semantics of ReactiveML provide much coarser semantics, than those for $\pi^{\lightning}_{\mathsf{R}}$ and $\mathsf{a}\pi$. This fact led to the conception of alternative notions of operational correspondence that are dependent on the coarseness of the semantics of the source and target languages (cf. § 2.1). Nonetheless, we also show that it is possible to derive Gorla's operational correspondence [Gor10] by using so-called *semantic correspondence results* (cf. § 2.4.2 and § 3.2.8).

Our encodings are also an improvement with respect to previous works which extend the π-calculus with either timed or event-based behavior, but not both. Similarly, the fact that our encodings yield runnable ReactiveML programs improves on the fact that some of these π-calculus extensions lack programming support. Interestingly, since ReactiveML has a well-defined semantics, it already offers a firm basis for both foundational and practical studies on session-based concurrency.

Besides generalizing the results in [CAP17], our results consider typed source processes. Although not stated in this work, we do not foresee any problems for proving *type soundness*: if P is a well-typed $\pi^{\lightning}_{\mathsf{R}}$ process, then $[\![P]\!]^{g}_{f}$ is a well-typed RML expression. We conjecture a similar result for $(\!|\cdot|\!)$, using the type system herein presented. On the ReactiveML side, we can exploit the type-and-effect system in [MP14] to enforce *cooperative* programs (roughly, programs without infinite loops). Since $[\![\cdot]\!]^{g}_{f}$ and $(\!|\cdot|\!)$ already produce well-typed, executable ReactiveML expressions, we further conjecture that they are also cooperative, in the sense of [MP14].

9.2 Related Work

SRP was introduced in the 1980s [BCE+03] as a way to implement and design critical real-time systems. Since then, several works have provided solid foundations for SRP programming languages. In particular, the work on ESTEREL [BG92] and the model presented in [BdS96] offer foundations for languages such as RML [MP05, MP14] and ULM [Bou04]. Also worth mentioning are works that relate synchronous languages to the π-calculus; for instance, the work [Ama07] develops a non-deterministic variant of the SRP model of ESTEREL. The paper [Hal98] offers a survey of synchronous reactive programming languages, including ESTEREL, LUSTRE [CPHP87], and several others.

Session types [HVK98] have been thoroughly studied. Previous works have extended the foundations of session-based concurrency to include event-based behav-

[1] *Synchronous communication* as in the (session) π-calculus should not be confused with the *synchronous programming* model of ReactiveML.

ior [KYHH16], adaptive behavior [CDV15], and timed behavior [BYY14]. All these extensions use (variants of) the π-calculus as their base language. A key difference with our work is that we propose an SRP language (i.e., RML) to obtain a natural integration of some of the aforementioned features. Practical approaches to session types have resulted in a variety of implementations, including [NT04, YHNN13, SY16]. The paper [ABB$^+$16] offers a recent survey of session types and behavioral types in practice.

Another relevant implementation is [Pad], a source of inspiration for our work: it integrates session-based concurrency in the OCaml programming language. As in our encoding, the implementation in [Pad] uses the notion of continuation-passing style developed in [DGS12]. A distinguishing feature of our work with respect to [Pad] is our interest in reactive, timed behaviors, not supported by OCaml, and therefore not available in [Pad]. Our current implementation still lacks some features present in [Pad], such as the integration of duality and linearity-related checks into the OCaml type system.

Our approach using encodings is related to our prior works on declarative interpretations of session π-calculi [LOP09, CRLP15]. The first of such encodings is developed in [LOP09], where it is shown that declarative languages can support *mobility* in the sense of the π-calculus. The encoding developed in [CRLP15] improves over [LOP09] by supporting linearity and non-determinism. The works [LOP09, CRLP15] are related to the present work due to the declarative flavor of SRP. In contrast, our reactive encodings yield practical implementations in ReactiveML, which are not possible in the foundational encodings in [LOP09, CRLP15].

Different from process calculus (and type-based validation techniques), other approaches to the formal specification and analysis of services use automata- and graph-based techniques. For instance, the work [FBS04] uses Büchi automata to specify and analyze the conversation protocols that underlie electronic services.

PART IV

A SYNCHRONOUS REACTIVE SESSION-BASED CALCULUS

10
Multiparty Reactive Sessions

This chapter differs from the rest in structure and content, as we distance ourselves from translations and introduce *Multiparty Reactive Sessions*, a synchronous reactive process calculus equipped with a session type system based on multiparty session types. In § 10.1 we introduce MRS. Then, § 10.2 illustrates our approach by means of the auction example discussed below. In § 10.3, we introduce the syntax and semantics of MRS. We prove that our model is *reactive*, namely that every reachable configuration instantaneously converges in a number of steps that is bounded by the size of the process (Theorem 10.18). Finally, we introduce the type system in § 10.4 and present correctness and time-related properties.

10.1 Introduction

We study the integration of *synchronous reactive programming* (SRP) [BdS96, MP05, MPP15b, BMS15] and *session-based concurrency* [HVK98, YV07, HYC08]. Our goal is to devise a uniform programming model for communication-centric systems in which some components are *reactive* and/or *timed*. Synchronous reactive programming is a well-established model rooted on a few features: *broadcast signals*, *logical instants*, and *event-based preemption*. This makes it an ideal vehicle for specifying and analyzing reactive systems; programming languages based on SRP include Esterel [BG92, PBEB07], Céu [SLS$^+$18], and ReactiveML [MP05]. On the other hand, session-based concurrency is the model induced by *session types* [YV07, HYC08], a rich typing discipline for message-passing programs. Session types specify protocols by stipulating the sequence in which messages should be sent/received by participants along a channel.

The interplay of message-passing concurrency with time- and event-based requirements is very common. In many protocols, participants are subject to time-related constraints (e.g., “the request must be answered within n seconds”). Also, protocols may depend, in various ways, on events that trigger run-time adaptations (e.g., “react to a timeout by executing an alternative protocol”). As a concrete example, consider a *buyer-seller protocol* in which a smart fridge manages groceries on behalf of a buyer, and only interacts with a supermarket in reaction to some event (say, “running out of milk”). Another example is an *electronic auction*, where an auctioneer offers a good for sale and buyers compete for this good by bidding the price upward. Here, the auctioneer supervises the bidding and decrees the knock-down price as soon as a standstill is reached. Like a physical auction, the electronic auction follows a multiparty protocol in which messages are broadcast to all participants, but they are “fetched” only by some of them. Bidders must be able to react in real time to the offers issued by other bidders and to the auctioneer’s decisions. These two examples are representative of a wide class of scenarios requiring both:

- the ability of broadcasting messages that are not fetched by all participants (“orphan messages” become the norm rather than the exception) and
- a synchronous preemption mechanism, allowing participants’ behaviors to be simultaneously reset in reaction to some event.

Unfortunately, existing frameworks based on session types lack these two key features—they are not expressive enough to model reactive and time-dependent interactions, essential in the two examples above. The framework in [KYHH16] handles contextual information through events, but does not support reactive behavior nor multiparty protocols. Models such as [BYY14, BCM$^+$15, BMVY19] account for multiparty protocols with time-related conditions, but do not support reactive and event-based behaviors. The work [CAP17] integrates SRP and session-based concurrency, but it is restricted to binary session types (protocols with two participants).

To address the limitations of current approaches, we propose a new typed framework for multiparty protocols, expressive enough to support reactive, structured communications. Our framework builds on a new process language dubbed MRS (Multiparty Reactive Sessions), which combines constructs from (session) π-calculi with typical features of synchronous reactive languages, namely:

- *Logical instants*, or simply *instants*, which are periods in which all components compute until they cannot evolve anymore (instants are what make SRP “synchronous”);
- Broadcast communication (instead of point-to-point communication);
- A “pause” construct, which suspends execution for the current instant;
- A “watch” construct implementing preemption, which is equipped with a standard and an alternative behavior that is triggered in reaction to a given event. This construct generalizes the exception mechanism provided by many programming languages, endowing it with a notion of time.
- Event emission, which is used here simply to control the watch construct.

The operational semantics of MRS is given in a style that is typical of synchronous languages. A process resides within a *configuration* with its memory and emitted events. There are two reduction relations on configurations: the first one formalizes small-step execution *within an instant*, until the configuration *converges*, namely *suspends* or terminates. Suspension occurs when all participants have exercised their right to send/receive for the current instant, or have reached a "pause" instruction. The second reduction relation formalizes how a suspended configuration evolves *across different instants*.

In more detail, during each instant, every participant can broadcast at most once and receive at most once from the same sender. This is a sensible requirement to discipline interaction in a reactive setting with valued messages. Indeed, allowing participants to broadcast multiple messages (valued events) in the same instant would amount to collect all their values at the end of the instant; then, an additional mechanism would be required to handle these "flattened" values and dispatch them in the expected order to the receivers.

Our semantics for MRS satisfies (*bounded*) *reactivity*, a standard soundness property of SRP which requires that small-step execution converges to a suspension or termination point at every instant [MP05]. This property, also called *instantaneous convergence* or *instant termination* in the SRP literature [TS05], is key for a reactive computation to evolve through a succession of instants and thus proceed as expected.

In session-based concurrency, protocol conformance typically follows from safety and liveness properties that stipulate how processes adhere to their session types, namely: session fidelity, communication safety, and (often, although not necessarily) some progress/deadlock-freedom property. In MRS, we further target the following two *time-related properties*:

P1. *Output persistence*: Every participant broadcasts exactly once during every instant;

P2. *Input timeliness*: Every unguarded input is matched by an output during the current instant, if not preceded by another input with equal source and target, or during the next instant, if not preempted.

Our main contribution is a type system, inspired by multiparty session types [HYC08], that enforces session fidelity, communication safety, as well as output persistence and input timeliness for MRS processes . One crucial technical challenge consists in properly handling *explicit* and *implicit* pauses in MRS processes. Explicit (or *syntactic*) pauses correspond to occurrences of the pause construct in processes. In contrast, implicit (or *semantic*) pauses are those induced by the synchronous reactive semantics between two broadcasts by the same participant, or between two inputs by the same participant from the same source.

Our type system relies on the usual ingredients of multiparty session types: *global types* entirely describe a multiparty protocol; *local types* stipulate the protocol associated to each participant; a *projection function* relates global and local types. However, because of the interplay between sessions and SRP, these ingredients have rather different definitions in our framework. In particular, we require a new pre-processing phase over global types called *saturation*, which complements protocols with implicit pauses. Unique to our setting, saturation is essential to conduct our static analysis on MRS processes and, ultimately, to reduce the conceptual gap between SRP and session-based concurrency.

10.2 Two Motivating Examples

We illustrate our typed process model by formalizing the two examples mentioned in § 10.1. We first present a reactive variant of the well-known *Buyer-Seller protocol*; then, we model the *Electronic Auction* protocol.

10.2.1 A Reactive Buyer-Seller Protocol

Consider a scenario involving three participants: a *Smart Fridge* (F), a *Client* (C), and a *Supermarket* (S). These participants interact with the following goal: F acts on behalf of C to purchase groceries from S. Being a smart, autonomous agent, F should react whenever a low level of groceries is detected, and initiate a protocol with S and C so as to restore a predefined level of groceries. F should obtain authorization from C before issuing a purchase order to S.

Before formalizing this protocol as a global type with reactive constructs, we introduce global types informally (a formal description shall be given in § 10.4):

- $\mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.G$ denotes a global type in which participant p broadcasts a message of sort S[1] which will be fetched by the participants in set Π; after that, the protocol continues as specified by G.
- $\mu\mathbf{t}.G$ represents a recursive protocol given by G, which includes occurrences of variable $\mathbf{t}$.
- Given event ev, we introduce $\mathtt{watch}\ ev\ \mathtt{do}\ G_1\ \mathtt{else}\ G_2$ as a reactive, event-dependent global type. This type says that protocol G_1 will be executed until termination or suspension. When G_1 suspends there are two possibilities: if ev has not occurred, then the remainder of G_1 is invoked again as the governing protocol in the next instant; otherwise, as a reaction to the occurrence of ev, the protocol G_1 is discarded and G_2 is invoked in the next instant.
- $\mathtt{pause}.G$ is also peculiar to our reactive, timed setting: it says that all participants should move to the next instant to execute protocol G.
- end represents the terminated protocol, as usual.

We then have the following global type G, which describes the multiparty protocol between the fridge, the client, and the supermarket. We use two events, named *lf* (for *low food*) and *ok*, which stands for a *confirmation* event:

$$
\begin{array}{l}
G = \mu\mathbf{t}_1.\mathtt{watch}\ \mathit{lf}\ \mathtt{do}\ \mu\mathbf{t}_2.\mathtt{S}\!\uparrow\!\langle stat, \{\mathtt{F}\}\rangle.\mathtt{F}\!\uparrow\!\langle stat, \{\mathtt{C}\}\rangle.\mathtt{C}\!\uparrow\!\langle stat, \{\mathtt{F}\}\rangle.\mathtt{pause}.\mathbf{t}_2 \\
\qquad\quad \mathtt{else} \\
\qquad\qquad \mathtt{S}\!\uparrow\!\langle stat, \{\mathtt{F}\}\rangle.\mathtt{F}\!\uparrow\!\langle lst, \{\mathtt{C}\}\rangle.\mathtt{C}\!\uparrow\!\langle lst, \{\mathtt{F}\}\rangle. \\
\qquad\qquad \mathtt{watch}\ ok\ \mathtt{do}\ \mathtt{pause}.\mu\mathbf{t}_3.\mathtt{S}\!\uparrow\!\langle stat, \{\mathtt{F}\}\rangle.\mathtt{F}\!\uparrow\!\langle stat, \{\mathtt{C}\}\rangle.\mathtt{C}\!\uparrow\!\langle stat, \{\mathtt{F}\}\rangle.\mathtt{pause}.\mathbf{t}_3 \\
\qquad\qquad \mathtt{else} \\
\qquad\qquad\quad \mathtt{C}\!\uparrow\!\langle stat, \{\mathtt{F}\}\rangle.\mathtt{F}\!\uparrow\!\langle lst, \{\mathtt{S}\}\rangle.\mathtt{S}\!\uparrow\!\langle prc, \{\mathtt{F}\}\rangle.\mathtt{pause}.
\end{array}
$$

[1]Basic types are called "sorts" here, following the terminology introduced by Milner [Mil91] and widely adopted in the session type literature.

$$\mathsf{C}\!\uparrow\!\langle stat, \{\mathsf{F}\}\rangle.\mathsf{F}\!\uparrow\!\langle cc, \{\mathsf{S}\}\rangle.\mathsf{S}\!\uparrow\!\langle iv, \{\mathsf{F}\}\rangle.\mathsf{pause}.\mathbf{t}_1$$

To describe the protocol specified by G, we spell out the instants it involves. We assume that *lf* is emitted at time t_{lf} and that ok is emitted at time t_{ok}.

Instant $t \leq t_{lf}$**:** the protocol enters the 'do' branch of the outermost watch, which is guarded by event *lf*, indicating low food levels in the fridge, and it starts executing a loop. In the body of this loop, S sends a status update ($stat$) to F indicating its availability for future purchases; next, F sends an update to C with information about the current items in the fridge (say, expiration dates), and C answer by updating his own status; finally, a pause is reached and the presence of event *lf* is checked:

1. If *lf* has not yet been emitted, then the 'do' branch is executed again in the next instant.
2. If *lf* is present, then the 'else' branch will be executed in the next instant.

Instant $t_{lf} + 1$**:** now the 'else' branch of the outermost watch is executed: once again, S updates his status; then, F sends to C a list with grocery items to buy (lst) and C can update the list with more items. Finally, the 'do' branch of the innermost watch is executed, guarded by event ok, indicating a confirmation. The pause makes the protocol move to the next instant, ensuring that the event ok may be immediately accounted for. If ok has not been emitted, then S, F and C keep issuing their status. Upon the emission of event ok, the innermost watch is exited and its 'else' branch is selected for the next instant.

Instant $t_{ok} + 1$**:** now the 'else' branch of the innermost watch is executed: C updates his status, F orders the groceries from S and gets back their price. Then the protocol moves to the next instant.

Instant $t_{ok} + 2$**:** once again, C updates his status, F sends to S the information required to complete the payment (e.g., credit card number), and S sends back the invoice iv to F.

As usual for multiparty session types, the global type G should be projected into local types for F, S, and C, which will be used to type-check against process implementations. In MRS, the local types are as follows:

- Local types $!S.T$ and $?(\mathsf{p}, S).T$, represent output and input. In the former type, a value of sort S is broadcast and then the type continues as T. In the latter, a value of sort S is received from participant p; the type then continues as T.

- Local types pause and $\langle T_1, T_2\rangle^{ev}$ specify the reactive behavior of participants, and they are similar to the corresponding constructs for global types.

- Type $\mu\mathbf{t}.T$ represents a recursive type, where **t** may occur in T.

We write $G\lfloor_{\mathsf{p}}$ to denote the local type obtained from the projection of global type G into participant p. This way, e.g., we have the following local type for C:

$$G\lfloor_{\mathsf{C}} = \mu\mathbf{t}_1.\langle \mu\mathbf{t}_2.?(\mathsf{F}, stat).!stat.\mathsf{pause}.\mathbf{t}_2, T\rangle^{lf}$$

$$\begin{array}{l}
System = Fridge \mid SMarket \mid Client \\
Fridge = \mathsf{rec}\ X_1\,.\,\mathtt{watch}\, \mathit{lf}\ \mathsf{do}\ \mathsf{rec}\ X_2\,.\,s[\mathsf{F}]?(\mathsf{S}, x_1).s[\mathsf{F}]!\langle stat\rangle.s[\mathsf{F}]?(\mathsf{C}, x_2). \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{if}\ (status(food) = 0)\ \mathtt{then}\ \mathtt{emit}\, \mathit{lf}.\, \mathsf{pause}.\, X_2\ \mathtt{else}\ \mathsf{pause}.\, X_2 \\
\qquad\qquad\qquad \{s[\mathsf{F}]?(s[\mathsf{S}], x_3).s[\mathsf{F}]!\langle \widetilde{food}\rangle.s[\mathsf{F}]?(\mathsf{C}, x_4). \\
\qquad\qquad\qquad\quad \mathtt{watch}\, ok\ \mathsf{do}\ \mathsf{pause}.\, \mathsf{rec}\ X_3\,.\,s[\mathsf{F}]?(\mathsf{S}, x_5).s[\mathsf{F}]!\langle stat\rangle.s[\mathsf{F}]?(\mathsf{C}, x_5).X_3 \\
\qquad\qquad\qquad\quad \{s[\mathsf{F}]?(\mathsf{C}, x_6).s[\mathsf{F}]!\langle\widetilde{x_7}\rangle.s[\mathsf{F}]?(s[\mathsf{S}, x_8]).\mathsf{pause}. \\
\qquad\qquad\qquad\qquad s[\mathsf{F}]?(\mathsf{C}, x_9).s[\mathsf{F}]!\langle cc\rangle.s[\mathsf{F}]?(\mathsf{S}, x_{10}).X_1\}\} \\
SMarket = \mathsf{rec}\ Y_1\,.\,\mathtt{watch}\, \mathit{lf}\ \mathsf{do}\ \mathsf{rec}\ Y_2\,.\,s[\mathsf{S}]!\langle stat\rangle.Y_2\{\mathtt{watch}\, ok\ \mathsf{do}\ \mathsf{pause}.\, \mathsf{rec}\ Y_3\,.\,s[\mathsf{S}]!\langle stat\rangle.Y_3 \\
\qquad\qquad\qquad \{s[\mathsf{S}]?(\mathsf{F}, y_1).s[\mathsf{S}]!\langle price(y_1)\rangle.\mathsf{pause}.\, s[\mathsf{S}]?(\mathsf{F}, y_2).s[\mathsf{S}]!\langle invoice\rangle.Y_1\}\} \\
Client = \mathsf{rec}\ Z_1\,.\,\mathtt{watch}\, \mathit{lf}\ \mathsf{do}\ \mathsf{rec}\ Z_2\,.\,s[\mathsf{C}]?(\mathsf{F}, z_2).s[\mathsf{C}]!\langle stat\rangle.Z_2 \\
\qquad\qquad\qquad \{s[\mathsf{C}]?(\mathsf{F}, z_4).s[\mathsf{C}]!\langle z_4\rangle. \\
\qquad\qquad\qquad\quad \mathtt{watch}\, ok\ \mathsf{do}\ \mathtt{emit}\, ok.\, \mathsf{pause}.\, \mathsf{rec}\ Z_3\,.\,s[\mathsf{C}]?(\mathsf{F}, z_6). \\
\qquad\qquad\qquad s[\mathsf{C}]!\langle stat\rangle.Z_3\{s[\mathsf{C}]!\langle stat\rangle.Z_1\}\}
\end{array}$$

Figure 10.1: MRS implementation of the Reactive Buyer-Seller Protocol.

where $T = ?(\mathsf{F}, lst).!lst.\langle \mathsf{pause}.\mu\mathbf{t}_3?(\mathsf{F}, stat).!stat.\mathsf{pause}.\mathbf{t}_3, !stat.\mathbf{t}_1\rangle^{ok}$.

In Fig. 10.1, we show a process implementation of our protocol; it allows us to introduce some salient constructs in MRS:

- Process $s[\mathsf{p}]!\langle v\rangle.P$ represents the broadcast of value v from participant p along session s; also, $s[\mathsf{p}]?(\mathsf{q}, x).P$ represents p receiving a message coming from q along session s.

- Process $\mathsf{pause}.\, P$ suspends for the current instant, and executes P in the next instant.

- Process $\mathtt{emit}\, ev.\, P$ emits an event ev, visible by all participants during the current instant, and then continues as P within the same instant.

- Process $\mathtt{watch}\, ev\ \mathsf{do}\ P\{Q\}$ is defined to correspond with local type $\langle T_1, T_2\rangle^{ev}$. This process executes P up to termination or suspension. In the former case, the whole process disappears. When P evolves to P' and suspends, there are two possibilities at the end of the instant, depending on ev: if ev has not occurred then $\mathtt{watch}\, ev\ \mathsf{do}\ P'\{Q\}$ is executed in the next instant; otherwise, as a reaction to the occurrence of ev, $\mathtt{watch}\, ev\ \mathsf{do}\ P'\{Q\}$ is discarded and Q is executed in the next instant.

The protocol implementation in Fig. 10.1 is given by process $System$, which is composed of three parallel processes ($Fridge$, $SMarket$, and $Client$), implementing participants F, S, and C, respectively:

Process $Fridge$ runs a recursive loop while there is enough food: it first receives a status update from S and broadcasts its own updates (to be received by $Client$);

it also receives and update from $Client$. Then, it checks the current level of food: if there is enough food $(status(food) \neq 0)$, then the status update loop is repeated. Otherwise, it emits *lf*, thus triggering the alternative behavior, which consists in sending the groceries list $(\widetilde{food})$ to $Client$, which in turn should answer with possible modifications to the list and then confirm the purchase by emitting ok. The status update loop will execute as long as ok has not occurred; when $Client$ confirms, $Fridge$ will first receive and status update from the client and then send the groceries list to $SMarket$, which will return the total price. Finally, another status update from the client is received and $Fridge$ exchanges payment information and invoice with $SMarket$.

Process $SMarket$ is engaged in the update loop with $Fridge$ until *lf* is emitted. Once this event occurs, the update loop will continue until *ok* is detected. After confirmation, $SMarket$ and $Fridge$ interact to finalize the purchasing protocol.

Process $Client$ is similar, and takes part in the status update loop until $Fridge$ emits *lf*. When this occurs, and after having received the groceries list from $Fridge$, $Client$ simply resends the list it received without modifications and confirms the purchase (clearly, more elaborate authorization procedures are possible). Once $SMarket$ and $Fridge$ have completed the purchase, $Client$ engages again into the status update loop.

10.2.2 An Electronic Auction Protocol

We now formalize the electronic auction protocol sketched in § 10.1. This example exhibits again the distinctive features of MRS: the slicing of computation into instants, broadcast communication, and the synchronous preemption mechanism. In addition, it illustrates the specific treatment of recursion in MRS, and it shows how parameterized recursion may be used to transmit values across instants.

Assuming n bidders $(n \geq 2)$, the protocol has $n+1$ participants: participant $\mathtt{A}$, which is the *Auctioneer*, and participants $\mathtt{B}_1, \ldots, \mathtt{B}_n$, which are the *Bidders*. Bidding rounds are represented by instants: at the start of each instant, all the bidders send their new bids to the auctioneer, which responds by broadcasting the new tuple of bids, whose maximum represents the current price of the good. We suppose that the starting price of the good is the same for all bidders, i.e., $\mathit{initPrice}_i = \mathit{initPrice}_j$ for any $i, j \in \{1, \ldots, n\}$; we also suppose that each Bidder_i has a maximal budget budget_i and that her initial bid $\mathit{initBid}_i$ is such that $\mathit{initPrice}_i < \mathit{initBid}_i \leq \mathit{budget}_i$ and $\mathit{initBid}_i \neq \mathit{initBid}_j$ for $i \neq j$. Moreover, we assume that each Bidder_i bids up by a fixed amount Δ_i such that $\Delta_i \neq \Delta_j$ for $i \neq j$. The two above conditions $\mathit{initBid}_i \neq \mathit{initBid}_j$ and $\Delta_i \neq \Delta_j$ are used to prevent equal bids from different bidders, which would make the protocol more involved[2]. Then, a session s of the protocol may be described by the *Auction* process in Fig. 10.2, where $\tilde{\sigma}$ represents the tuple $(\sigma_1, \ldots, \sigma_n)$ and for any such tuple $\tilde{\sigma}$, the function $max(\tilde{\sigma})$ is defined by $max(\tilde{\sigma}) = max\{\sigma_1, \ldots, \sigma_n\}$.

[2]Since multiparty session protocols are usually deterministic, if equal bids were allowed there should be some predefined criterion to choose between them. Moreover, in a physical auction all bids must be different, since they are issued one after the other. The requirement $\Delta_i \neq \Delta_j$ allows this to be mimicked using simultaneous bids.

$$
\begin{aligned}
&\mathit{Auction} = \mathit{Auctioneer} \mid \mathit{Bidder}_1 \mid \cdots \mid \mathit{Bidder}_n\\
&\mathit{Auctioneer} = \mathtt{watch}\, \mathit{bis}\ \mathtt{do}\ \big(\mathtt{rec}\ X(\widetilde{x})\ .\ s[\mathsf{A}]?(\mathsf{B}_1, \mathit{bid}_i).\ \cdots\ .\ s[\mathsf{A}]?(\mathsf{B}_n, \mathit{bid}_n).\\
&\qquad\qquad\qquad \mathtt{if}\ \textstyle\bigwedge_{i\in I}(\mathit{bid}_i = x_i)\ \mathtt{then}\ \mathtt{emit}\, \mathit{bis}.\ s[\mathsf{A}]!\langle\widetilde{\mathit{bid}}\rangle.X(\widetilde{\mathit{bid}})\\
&\qquad\qquad\qquad \mathtt{else}\ s[\mathsf{A}]!\langle\widetilde{\mathit{bid}}\rangle.X(\widetilde{\mathit{bid}})\big)(\widetilde{\mathit{initPrice}})\\
&\qquad\qquad \{s[\mathsf{A}]!\langle \mathit{max}(\tilde{\mathit{bid}})\rangle.\mathbf{0}\}\\
&\mathit{Bidder}_i = \mathtt{watch}\, \mathit{bis}\ \mathtt{do}\ \big(\mathtt{rec}\ Z_i(z_i)\ .\ s[\mathsf{B}_i]!\langle z_i\rangle.s[\mathsf{B}_i]?(\mathsf{A}, \tilde{w}).\\
&\qquad\qquad \mathtt{if}\ (\ \mathit{max}(\tilde{w}) \neq z_i \wedge \mathit{max}(\tilde{w}) + \Delta_i \leq \mathit{budget}_i)\ \mathtt{then}\ Z_i(\mathit{max}(\tilde{w}) + \Delta_i)\\
&\qquad\qquad \mathtt{else}\ Z_i(z_i)\big)(\mathit{initBid}_i)\\
&\qquad \{s[\mathsf{B}_i]?(\mathsf{A}, z_i').s[\mathsf{B}_i]!\langle \mathit{eog}\rangle.\mathbf{0}\}
\end{aligned}
$$

Figure 10.2: MRS implementation of the Electronic Auction Protocol.

Note that *Auctioneer* and the *Bidder*$_i$ have a similar structure: they consist of a watch statement guarded by the event bis, whose main branch executes a loop and whose alternative branch does an I/O action and terminates. The event bis (standing for "bis repetita") is emitted by *Auctioneer* to signal that the same tuple of bids has occurred twice and hence the auction is over.

Supposing event bis is emitted at instant t_{bis}, let us see how instants build up in our protocol.

Instant $t \leq t_{bis}$ **:** At the beginning, all participants enter the main branch of the `watch` guarded by event *bis*, and they start executing a loop. In the body of their loop, all *Bidder*$_i$ broadcast their new bids (which in the first iteration are just their initial bids $initBid_i$), and then wait for a new tuple of bids from *Auctioneer*. Now, *Auctioneer* inputs all the new bids from the *Bidder*$_i$ and compares them with their previous bids (which in the first iteration of *Auctioneer* are just the $initPrice_i$). If the new bid from each *Bidder*$_i$ is equal to her previous bid, then a standstill is reached and *Auctioneer* emits the event bis to trigger the alternative behavior of all participants at the next instant; then *Auctioneer* broadcasts the same tuple of bids (this broadcast is required to match the expectations of the *Bidder*$_i$). Otherwise, *Auctioneer* simply broadcasts the new tuple of bids. In both cases, *Auctioneer* suspends before starting the next iteration, because the semantic rule for recursion inserts a `pause` before the next occurrence of the recursive call. After the broadcast from *Auctioneer*, all *Bidder*$_i$ fetch the tuple of new bids and check that the maximum $max(\tilde{z})$ of this tuple (the best offer so far) is a bid different from their own, and that their budget allows them to bid up; if this is the case, they increment $max(\tilde{z})$ by Δ_i and suspend; otherwise, they issue again their previous bid z_i and suspend. The execution goes on similarly until the tuple of bids reaches a standstill (this is ensured by the fact that each *Bidder*$_i$ bids upwards by a fixed amount Δ_i, while not overriding budget_i), leading eventually to the emission of event *bis* by *Auctioneer*. At this point, all participants are deviated from their main behavior and their alternative behavior is triggered at the next instant.

Instant $t_{bis}+1$: Now *Auctioneer* broadcasts the knock-down price, which is received by all the *Bidder*$_i$, who then react by sending an "end of game" message. Note that, because of our hypotheses, the knock-down price uniquely identifies the winner.

The global type for the protocol is as follows (see § 10.4 for the local types):

$$\begin{aligned} G = &\,\texttt{watch}\ bis\ \texttt{do}\ \mu\mathbf{t}.\mathsf{B}_1\!\uparrow\!\langle\mathsf{int},\{\mathsf{A}\}\rangle.\ \cdots\ .\mathsf{B}_n\!\uparrow\!\langle\mathsf{int},\{\mathsf{A}\}\rangle.\mathsf{A}\!\uparrow\!\langle\widetilde{\mathsf{int}},\{\mathsf{B}_1,\ldots,\mathsf{B}_n\}\rangle.\mathbf{t}\\ &\quad\texttt{else}\ \mathsf{A}\!\uparrow\!\langle\mathsf{int},\{\mathsf{B}_1,\ldots,\mathsf{B}_n\}\rangle.\mathsf{B}_1\!\uparrow\!\langle\mathsf{string},\emptyset\rangle.\ \cdots\ .\mathsf{B}_n\!\uparrow\!\langle\mathsf{string},\emptyset\rangle.\texttt{end}\end{aligned}$$

Summing Up. This example illustrates some distinctive features of our framework:

- Messages are *valued events* which are broadcast, hence they are not consumed when they are read; instead, they are consumed by the *passage of time*, since they are erased at the end of each instant;
- Our typed calculus imposes a strong common structure in protocol participants; while this may seem contrived, it is the source of the correctness properties enforced by our type system. For instance, bidders who wish to drop from the auction still have to issue their last bid until the end of the auction. This is because the bidders must match the expectations of the *Auctioneer*, who waits for a bid from all bidders at each instant since she cannot foresee at which point they will give up.
- In our calculus, as in most multiparty session frameworks, the set of participants is fixed and participants cannot dynamically enter or leave a session. It is possible however that some participants may terminate before the others, and the output persistence property is only required for nonterminated participants (although this does not appear in the above example).

Having illustrated informally MRS and its typed system, we now introduce them formally.

10.3 Our Process Model: MRS

We introduce MRS, our calculus of Multiparty Reactive Sessions. It integrates constructs from Synchronous Reactive Programming and from multiparty session π-calculi.

10.3.1 Syntax

We assume the following basic sets: *values* (booleans, integers), ranged over by v, v'; *value variables*, ranged over by x, y, z; and *expressions*, ranged over by e, e'. Expressions are built from variables and values via standard operators, and their evaluation is terminating. A set of *process variables* $X, Y, \ldots$, possibly parameterised by a tuple of parameters (written in this case as $X(\tilde{x})$ or $X(\tilde{e})$), is assumed to define recursive behaviors. We also use two sets that are specific to multiparty session calculi [HYC08, CDPY15]: *service names*, ranged over by a, b, each of which has an *arity* $n \geq 2$ (its number of participants), and *sessions*, denoted by s, s'. A session represents a particular activation of a service. We use p, q, r to denote generic (*session*) *participants*. In an n-ary session, participants will often be assumed to range over the natural numbers $1, \ldots, n$ (in particular, we will use this assumption when defining the operational semantics). We denote by Π a non empty set of participants, and by Part_s the set of participants of session s. Finally, we assume a set of events *Events*, ranged over by ev, ev', which will be used for defining the reactive constructs.

Each n-ary session s has an associated set of *session channels* $\{s[1], \ldots, s[n]\}$, one per participant: channel $s[\mathsf{p}]$ is the private channel through which p communicates with other participants in session s. We also assume a set of *channel variables*, ranged over by α, β, γ; we use c to range over both channel variables and session channels.

The syntax of *processes*, ranged over by $P, Q \ldots$, is given in Fig. 10.4. A new session s on the n-ary service a is opened when the *initiator* $\bar{a}[n]$ synchronizes with n processes of the form $a[i](\alpha_i).P_i$, whose channels α_i then get replaced by $s[i]$ in the body of P_i. (This synchronization will be made precise by our operational semantics.) The initiator $\bar{a}[n]$ simply marks the presence of the service a, therefore it has no continuation behavior. Processes of the form $a[i](\alpha_i).P_i$ are called “candidate participants” for service a.

Rather than typical point-to-point communication, we consider communication based on *broadcast* and *directed input*. The former is denoted $c!\langle e\rangle.P$: this is an “undirected output” on c, for it does not mention any intended recipient for message e. The latter, denoted $c?(\mathsf{p}, x).P$, represents the input of a message sent by p. For simplicity we do not consider branching/selection operators here. Constructs for conditional expressions, parallel composition are standard. and have expected meanings.

With respect to usual calculi for multiparty sessions, the main novelty in MRS is the addition of three reactive constructs typical of synchronous languages, given on the bottom right of Fig. 10.4. They are:

- $\mathtt{pause}.\, P$, which postpones the execution of P to the next instant;
- $\mathtt{emit}\, ev.\, P$, which emits event ev in the current instant and then executes P;
- $\mathtt{watch}\, ev\, \mathtt{do}\, P\{Q\}$, a construct that we call “watch-and-replace”. It executes P and, in the presence of event ev, it replaces whatever is left of P by Q at the end of the instant. P and Q are respectively the *main behavior* and the *alternative behavior* of the construct.

Our watching construct is slightly more general than similar constructs in synchronous languages. Without a sequential composition operator–not present in MRS (nor in most session calculi), but common in synchronous languages–this added generality is actually needed. Indeed, without sequential composition a watching statement cannot be followed by another statement; therefore, if we were to use the standard $\mathtt{watch}\, ev\, \mathtt{do}\, P$ construct, this would just lead to termination at the end of the instant in case ev is present.

For recursion we assume the standard *guardedness* condition, adapted to our language:

$$
\begin{array}{rcll}
v & ::= & \texttt{tt} \mid \texttt{ff} \mid 1 \mid \ldots & \text{(Value)}\\
e & ::= & x \mid v \mid \mathsf{not}\ e \mid & \\
 & & e\ \mathsf{and}\ e' \mid \ldots \mid f(x_1, \ldots, x_n) & \text{(Expression)}\\
u & ::= & a \mid s & \text{(Service/Session Name)}\\
c & ::= & \alpha \mid s[\mathsf{p}] & \text{(Channel variable/Session channel)}\\
\Pi & ::= & \{\mathsf{p}\} \mid \Pi \cup \{\mathsf{p}\} & \text{(Set of participants)}\\
m & ::= & \epsilon \mid (v, \Pi) & \text{(Message—with set of readers)}\\
M & ::= & \emptyset \mid M \cup \{c : m\} \quad c \notin dom(M) & \text{(Memory)}
\end{array}
$$

Figure 10.3: MRS: Syntax of expressions, sessions, channels, messages and memories.

$$
\begin{array}{rcll}
P & ::= & \bar{a}[n] & \text{(Session initiator)}\\
 & \mid & a[\mathsf{p}](\alpha).P & \text{(Session participant)}\\
 & \mid & c!\langle e\rangle.P & \text{(Broadcast Output)}\\
 & \mid & c?(\mathsf{p}, x).P & \text{(Input)}\\
 & \mid & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q & \text{(Conditional)}\\
 & \mid & P \mid Q & \text{(Parallel)}\\
 & \mid & \mathbf{0} & \text{(Inaction)}\\
 & \mid & X(\tilde{e}) & \text{(Variable)}\\
 & \mid & (\texttt{rec}\ X(\tilde{x}) \,.\, P)\ (\tilde{e}) & \text{(Recursion)}\\
 & \mid & \texttt{pause}.\, P & \text{(Pause)}\\
 & \mid & \texttt{emit}\, ev.\, P & \text{(Emit)}\\
 & \mid & \texttt{watch}\, ev\ \mathsf{do}\ P\{Q\} & \text{(Watch \& Replace)}
\end{array}
$$

Figure 10.4: MRS: Syntax of processes.

Definition 10.1 (Guardedness). A variable X is *guarded* in P if it only occurs in subprocesses $c!\langle e\rangle.Q$ or $c?(\mathsf{p}, x).Q$ or $\texttt{emit}\, ev.\, Q$ or $\texttt{pause}.\, Q$ of P.

Note that the syntax of MRS is quite liberal. In particular, it allows processes with interleaved communications in different sessions, such as $s[1]!\langle e\rangle.s'[1]!\langle e'\rangle.P$. However, in the rest of this chapter we focus on *processes without session interleaving*, and our technical treatment is developed only for them. The restriction to single sessions is standard in session calculi, as interleaving introduces inter-session dependencies that cannot be captured by session types. Moreover, interleaving would raise new issues in our setting, particularly as regards suspension of configurations involved in more than one session.

10.3.2 *Semantics*

We present now the semantics of MRS. In SRP, parallel components communicate via *broadcast events*, which may be either *valued*, if they carry some content, or *pure*. We call valued events *messages*, and pure events simply *events*. To model broadcast communication we assume that all processes share a *message set* or *memory* M, recording the messages exchanged in ongoing sessions during the current instant, and an *event set* E, recording the events emitted during the instant. Both sets are emptied at the end of each instant.

A memory M is a finite set of *named messages* $c : m$, where c is the name of the channel on which the message was sent, and m is the message content: this content may be either empty or of the form (v, Π), where v is the carried value and Π is the set of current *Readers* of the value. We will see in the next section why we need to record the set Π of readers for non-empty messages. A memory M may be viewed as a partial function from channels to messages, whose domain $dom(M) = \{c \mid \exists\, m \,.\, c : m \in M\}$ is finite.

The *session memory* M_s of session s has the form $\bigcup_{i \in \mathsf{Part}_s} \{s[i] : m_i\}$, where each $s[i] : m_i$ represents the (one-place) *output buffer* of participant i, which contains the empty message if participant i has not yet broadcast in the current instant, and a proper message otherwise. We shall use the following auxiliary notations: $M_s^{\emptyset} = \bigcup_{i \in \mathsf{Part}_s} \{s[i] : \epsilon\}$ and $M^{\emptyset} = \{c : \epsilon \mid c \in dom(M)\}$. Fig. 10.3 (bottom) summarizes the notation for memories.

The sets of *free names*, *bound names*, and *names* of a process P, denoted respectively by *fn*(P), *bn*(P), *nm*(P), are defined as usual. Assuming Barendregt's convention, no bound name can occur free, and the same bound name cannot occur in two different bindings.

The semantics of MRS is defined on *configurations*. In its simplest form, a configuration is a triple $C = \langle P, M, E \rangle$, where P is a process, M is a memory, and E is a set of events. A configuration may also be restricted with respect to a session name s, namely have the form $(\nu s)\, C$. Our semantics will not be defined on arbitrary configurations, but only on those that may occur in the execution of a single session. Intuitively, these are the configurations that may be reached in zero or more steps from an initial configuration, as defined below (Def. 10.3). A formal definition of reachability will be given at the end of this section (Def. 10.4).

We first introduce the notion of sequential and session-closed process:

Definition 10.2 (Sequentiality and Session-Closedness). A process P is said to be *sequential* if it is built without the parallel construct $\mid$, and *session-closed* if it is built without the constructs $\bar{a}[n]$ and $a[\mathsf{p}](\alpha).Q$.

An initial configuration represents a state from which a single session may start:

Definition 10.3 (Initial Configuration). A configuration C_0 is *initial* if it is of the form

$$C_0 = \langle a[1](\alpha_1).P_1 \mid ... \mid a[n](\alpha_n).P_n \mid \bar{a}[n], \emptyset, \emptyset \rangle$$

where for each $i = 1, \ldots, n$, process P_i is sequential and session-closed, and $c \in$ *nm*(P_i) implies $c = \alpha_i$.

$$P \mid \mathbf{0} \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad \mathtt{watch}\, ev\ \mathtt{do}\ \mathbf{0}\{Q\} \equiv \mathbf{0}$$
$$P \equiv Q \ \Rightarrow\ \langle P, M, E\rangle \equiv \langle Q, M, E\rangle \quad C \equiv C' \ \Rightarrow\ (\nu s)C \equiv (\nu s)C'$$

Figure 10.5: Structural congruence.

[Init]
$$\langle a[1](\alpha_1).P_1 \mid ... \mid a[n](\alpha_n).P_n \mid \bar{a}[n], \emptyset, \emptyset\rangle \longrightarrow (\nu s)\langle P_1\{^{s[1]}/_{\alpha_1}\} \mid \ldots \mid P_n\{^{s[n]}/_{\alpha_n}\}, M_s^{\emptyset}, \emptyset\rangle$$

$$[\text{Out}]\ \frac{e \downarrow v}{\langle s[\mathsf{p}]!\langle e\rangle.P, M \cup \{s[\mathsf{p}] : \varepsilon\}, E\rangle \longrightarrow \langle P, M \cup \{s[\mathsf{p}] : (v, \emptyset)\}, E\rangle}$$

$$[\text{In}]\ \frac{\mathsf{q} \notin \Pi}{\langle s[\mathsf{q}]?(\mathsf{p}, x).P, M \cup \{s[\mathsf{p}] : (v, \Pi)\}, E\rangle \longrightarrow \langle P\{^{v}/_{x}\}, M \cup \{s[\mathsf{p}] : (v, \Pi \cup \mathsf{q})\}, E\rangle}$$

[Emit]
$$\langle \mathtt{emit}\, ev.\, P, M, E\rangle \longrightarrow \langle P, M, E \cup \{ev\}\rangle$$

[If-T]
$$\frac{e \downarrow \mathtt{tt}}{\langle \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q, M, E\rangle \longrightarrow \langle P, M, E\rangle}$$

[Rec]
$$\frac{\langle P\{^{\tilde{v}}/_{\tilde{x}}\}\{^{(\mathtt{pause}.\,\mathtt{rec}\, X(\tilde{x})\,.\,P)}/_{X}\}, M, E\rangle \longrightarrow \langle P', M, E\rangle \quad \tilde{e} \downarrow \tilde{v}}{\langle (\mathtt{rec}\, X(\tilde{x})\,.\,P)(\tilde{e}), M, E\rangle \longrightarrow \langle P', M, E\rangle}$$

$$[\text{Cont}]\ \frac{\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle}{\langle \mathcal{E}[P], M, E\rangle \longrightarrow \langle \mathcal{E}[P'], M', E'\rangle} \qquad [\text{Res}]\ \frac{\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle}{(\nu s)\langle P, M, E\rangle \longrightarrow (\nu s)\langle P', M', E'\rangle}$$

[Struct]
$$\frac{C \equiv C' \quad C' \longrightarrow C'' \quad C'' \equiv C'''}{C \longrightarrow C'''}$$

Figure 10.6: Reduction rules for MRS (with Rule [If-F] omitted).

We define two reduction relations on configurations, denoted $\longrightarrow$ and $\hookrightarrow_E$: while $\longrightarrow$ describes the evolution within an instant, $\hookrightarrow_E$ describes the evolution from one instant to the next one.

Reduction is defined modulo a structural congruence $\equiv$, whose rules are given in Fig. 10.5 and are standard [Mil99]. The *reduction relation* $\longrightarrow$ describes the step-by-step execution of a configuration within an instant. It is defined by the rules in Fig. 10.6. Let us discuss some of them.

Rule [Init] describes the initiation of a new session s of service a among n processes of the required form. After the initiation, participants share a private session name s, and the channel variable α_{p} is replaced by the session channel $s[\mathsf{p}]$ in each process P_{p}.

Rule [Out] allows a sender p to broadcast a message by adding it to the memory, if p has not already sent a message in the current instant, namely if the output buffer of p has the form $s[\mathsf{p}] : \epsilon$. In the premise, $e \downarrow v$ denotes the evaluation of expression e to value v. If the message can be added, its content is set to v and its *reader set* is initialized to $\emptyset$.

Rule [In] allows a receiver q to fetch a message from sender p in the memory, if there exists one and if q has not already read it, namely if q does not belong to the

$$(\textsc{pause})\ \frac{}{\langle \mathsf{pause}.\,P, M, E\rangle\ddagger} \qquad (\textsc{out}_s)\ \frac{}{\langle s[\mathsf{p}]!\langle e\rangle.P, M \cup \{s[\mathsf{p}] : (v,\Pi)\}, E\rangle\ddagger} \qquad (\textsc{par}_s)\ \frac{\langle P, M, E\rangle\ddagger \quad \langle Q, M, E\rangle\ddagger}{\langle P \mid Q, M, E\rangle\ddagger}$$

$$(\textsc{watch}_s)\ \frac{\langle P, M, E\rangle\ddagger}{\langle \mathsf{watch}\, ev\ \mathsf{do}\ P\{Q\}, M, E\rangle\ddagger} \qquad (\textsc{in}_s)\ \frac{}{\langle s[\mathsf{q}]?(\mathsf{p},x).P, M \cup \{s[\mathsf{p}] : \varepsilon\}, E\rangle\ddagger}$$

$$(\textsc{restr}_s)\ \frac{\langle P, M, E\rangle\ddagger}{(\nu s)\langle P, M, E\rangle\ddagger} \qquad (\textsc{in}^2_s)\ \frac{\mathsf{q} \in \Pi}{\langle s[\mathsf{q}]?(\mathsf{p},x).P, M \cup \{s[\mathsf{p}] : (v,\Pi)\}, E\rangle\ddagger}$$

$$(\textsc{rec}_s)\ \frac{\langle P\{\tilde{v}/\tilde{x}\}\{(\mathsf{pause}.\,\mathsf{rec}\ X(\tilde{x}) \,.\, P)/X\}, M, E\rangle\ddagger \quad \tilde{e}\downarrow\tilde{v}}{\langle(\mathsf{rec}\ X(\tilde{x}) \,.\, P)(\tilde{e}), M, E\rangle\ddagger} \qquad (\textsc{cong}_s)\ \frac{C \equiv C' \quad C'\ddagger}{C\ddagger}$$

Figure 10.7: Suspension Predicate.

$$[P]_E \overset{\mathsf{def}}{=} \begin{cases} R & \text{if } P = \mathsf{pause}.\,R \\ [R]_E \mid [Q]_E & \text{if } P = R \mid Q \\ Q & \text{if } P = \mathsf{watch}\, ev\ \mathsf{do}\ R\{Q\}, ev \in E \text{ and } R \not\equiv \mathbf{0} \\ \mathsf{watch}\, ev\ \mathsf{do}\ [R]_E\{Q\} & \text{if } P = \mathsf{watch}\, ev\ \mathsf{do}\ R\{Q\}, ev \notin E \\ P & \text{otherwise} \end{cases}$$

Figure 10.8: Reconditioning Function.

$$(\textsc{tick})\ \frac{(\nu s)\langle P, M, E\rangle\ddagger}{(\nu s)\langle P, M, E\rangle \hookrightarrow_E (\nu s)\langle [P]_E, M^{\emptyset}, \emptyset\rangle}$$

Figure 10.9: Tick transition.

reader set Π of the message. If q can read the message, then its value is substituted for the bound variable in the continuation process P, and the name q is added to the reader set Π.

Rule [Rec] inserts a pause before each recursive call, as usual in SRP, in order to allow at most one loop iteration at each instant and thus prevent the phenomenon known as *instantaneous loop* or *instantaneous divergence* [TS05]. Although parameterized recursion is used in our examples, for the sake of simplicity we will focus on unparameterized recursion in the rest of the chapter. Including parameters would have no impact on our results, but it would be tedious to carry them throughout our technical development.

The evaluation contexts $\mathcal{E}$ used in Rule [Cont] are defined by:

$$\mathcal{E} ::= [\,] \ \big|\ \mathcal{E} \mid P \ \big|\ P \mid \mathcal{E} \ \big|\ \mathsf{watch}\, ev\ \mathsf{do}\ \mathcal{E}\{Q\}$$

Note that the watch construct behaves as a static context as far as the reduction relation is concerned: the body of a watch process is executed up to the end of the instant, disregarding the event ev (which is relevant only for the relation $\hookrightarrow_E$ across instants).

To define the *tick transition relation* $\hookrightarrow_E$, we require two additional notions: the *suspension predicate* and the *reconditioning function*.

The suspension predicate $\langle P, M, E \rangle \ddagger$ (cf. Fig. 10.7) holds when all non-terminated components of P are in one of the following situations:

- wanting to release the control explicitly via a pause. Q instruction;
- wanting to send a message after having already sent a message during the instant;
- awaiting a message from a participant who has not sent anything during the instant;
- awaiting a second message from the same participant during the same instant.

The reconditioning function (Fig. 10.8) "cleans-up" a process P and prepares it for the next instant: it erases all guarding pauses from pause. Q processes, and triggers the alternative behavior Q of all the processes watch ev do $P\{Q\}$ whose controlling event ev has been emitted.

The tick relation $\hookrightarrow_E$ applies only to suspended configurations: it formalizes the passage of (logical) time and delimits the duration of broadcast by clearing out the memory and the event environment at the end of each instant. Formally, this relation is specified by the rule in Fig. 10.9, where $[P]_E$ is the reconditioning of P with respect to E.

As usual, we use $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. We write $\rightsquigarrow$ to denote either $\longrightarrow$ or $\hookrightarrow_E$, and $\rightsquigarrow^*$ to stand for the reflexive and transitive closure of $\rightsquigarrow$.

The following definition of reachability characterizes the configurations that may occur in the execution of a single session.

Definition 10.4 (Reachable Configuration). A configuration C is *reachable* if there exists an initial configuration C_0 such that $C_0 \rightsquigarrow^* C$.

Proposition 10.5. *If C is a* reachable *configuration that is not initial, then*

$$C = (\nu s)\langle P, M, E \rangle$$

and there exist an initial configuration $C_0 = \langle P_0, M_0, E_0 \rangle$ and P_i, M_i, E_i for $i = 1, \ldots, n$ such that $C_0 \rightsquigarrow (\nu s)\langle P_1, M_1, E_1 \rangle \rightsquigarrow \cdots \rightsquigarrow (\nu s)\langle P_n, M_n, E_n \rangle = (\nu s)\langle P, M, E \rangle$.

From now on we will focus only on reachable configurations, without explicitly mentioning it.

10.3.3 Reactivity

In this section, we prove that single sessions are *reactive* in our calculus, namely that every instant in their execution terminates. The single-session assumption is required, as it is well-known that interleaved sessions are subject to deadlock, and the possibility of deadlock would impair reactivity. On the other hand, it would be easy to extend our reactivity result to a pool of disjoint sessions evolving in parallel.

We now introduce some preliminary notation. First, we define the multi-step transition relation $C \Rightarrow C'$, together with its decorated variant $C \Rightarrow_n C'$ that keeps track of the number of execution steps between two configurations within an instant.

Definition 10.6 (Multi-Step Transition Relation). The *decorated multi-step transition relation* $C \Rightarrow_n C'$ is defined by:

$$C \Rightarrow_0 C \qquad\qquad (\, C \longrightarrow C' \;\wedge\; C' \Rightarrow_n C''\,) \;\;\Rightarrow\;\; C \Rightarrow_{n+1} C''$$

Then the *multi-step transition relation* $C \Rightarrow C'$ is given by:

$$C \Rightarrow C' \;\text{ if }\; \exists\, n\,.\, C \Rightarrow_n C'$$

Next, we define the notion of instantaneous convergence, which formalizes the fact that a configuration may reach a state of termination or suspension in the current instant.

Definition 10.7 (Instantaneous Convergence). The *immediate convergence* predicate is defined by:

$$\begin{array}{rl} \langle P, M, E\rangle \underset{\curlyvee}{\ddagger} & \text{if } \;\langle P, M, E\rangle \ddagger \;\vee\; (P \equiv \mathbf{0}) \\ (\nu s)\langle P, M, E\rangle \underset{\curlyvee}{\ddagger} & \text{if } \;\langle P, M, E\rangle \underset{\curlyvee}{\ddagger} \end{array}$$

Then the *instantaneous convergence* relation and predicate are given by:

$$C \Downarrow C' \;\text{ if }\; C \Rightarrow C' \;\wedge\; C' \underset{\curlyvee}{\ddagger} \qquad\qquad C \Downarrow \;\text{ if }\; \exists\, C'\,.\; C \Downarrow C'$$

The annotated variants $\Downarrow_n$ may be defined in the obvious way:

$$C \Downarrow_n C' \;\text{ if }\; C \Rightarrow_n C' \;\wedge\; C' \underset{\curlyvee}{\ddagger} \qquad\qquad C \Downarrow_n \;\text{ if }\; \exists\, C'\,.\; C \Downarrow_n C'$$

By abuse of notation, if $\sigma = C_0 \longrightarrow \cdots \longrightarrow C_n$ is a computation of C_0 and $C_n \underset{\curlyvee}{\ddagger}$, we shall say that the computation σ converges (or converges to C_n).

We proceed now to prove reactivity. In fact, we shall prove a stronger property, *bounded reactivity*, which says that every configuration $\langle P, M, E\rangle$ instantaneously converges in a number of steps that is bounded by the *instantaneous size of process P in memory M*, denoted $size_M(P)$. Intuitively, $size_M(P)$ is an upper bound for the number of steps that P can execute during the first instant when run in memory M. Therefore, the idea for defining $size_M(P)$ is that it should not take into account the portion of P that follows a pause instruction (a "syntactic pause"). Moreover, $size_M(P)$ should span at most one iteration of recursive subprocesses, and ignore the alternative behavior in watching subprocesses. Finally, in order to account for implicit pauses (or "semantic pauses"), $size_M(P)$ should stop counting when it meets an output on channel c, respectively an input on channel c from participant p, in both process P and memory M.

Let $\mathcal{M}$ denote the set of memories.

Definition 10.8 (Communications and Fired Channels). For any memory $M \in \mathcal{M}$, we define:

$$\begin{array}{rcl} Fired(M) & \overset{\mathsf{def}}{=} & \{(s, \mathsf{p}) \mid \exists\, v, \Pi.\; s[\mathsf{p}] : (v, \Pi) \in M\} \\ Comm(M) & \overset{\mathsf{def}}{=} & \{(s, \mathsf{p}, \mathsf{q}) \mid \exists\, v, \Pi.\, (\, s[\mathsf{p}] : (v, \Pi) \in M \;\wedge\; \mathsf{q} \in \Pi\,)\} \end{array}$$

The above functions allow us to extract useful information from the memory: $Fired(\cdot)$ identifies the participants that have sent a message in the current instant, and $Comm(\cdot)$ yields the pairs of participants that have successfully communicated in the current instant.

Definition 10.9 (Instantaneous Size). The partial function *size* : $(\mathcal{P} \times \mathcal{M}) \to$ **Nat** is defined inductively by:

$$size_M(\mathbf{0}) \overset{\mathsf{def}}{=} size_M(X) \overset{\mathsf{def}}{=} size_M(\mathtt{pause}.\, P) \overset{\mathsf{def}}{=} size_M(\bar{a}[n]) \overset{\mathsf{def}}{=} 0$$

$$size_M(\mathtt{emit}\, ev.\, P) \overset{\mathsf{def}}{=} 1 + size_M(P)$$

$$size_\emptyset(a[\mathsf{p}](\alpha).P) \overset{\mathsf{def}}{=} 1 + size_{M_s^\emptyset}(P\{s[\mathsf{p}]/\alpha\}), \text{ for any } s$$

$$size_M(s[\mathsf{p}]!\langle e\rangle.P) \overset{\mathsf{def}}{=} \begin{cases} 0, & \text{if } (s,\mathsf{p}) \in Fired(M) \\ 1 + size_{M'}(P), \text{where} & \text{if } s[\mathsf{p}] : \epsilon \in M \\ M' = M\{s[\mathsf{p}] \mapsto (val(e), \emptyset)\} & \end{cases}$$

$$size_M(s[\mathsf{q}]?(\mathsf{p},x).P) \overset{\mathsf{def}}{=} \begin{cases} 0, & \text{if } s[\mathsf{p}]:\epsilon\in M \\ & \quad\text{or } (s,\mathsf{p},\mathsf{q}) \in Comm(M) \\ 1 + size_{M'}(P\{v/x\}), \text{where} & \text{if } s[\mathsf{p}]:(v,\Pi)\in M \wedge \mathsf{q} \notin \Pi \\ M' = M\{s[\mathsf{p}] \mapsto (v, \Pi \cup \mathsf{q})\} & \end{cases}$$

$$size_M(\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) \overset{\mathsf{def}}{=} 1 + max\{size_M(P_1), size_M(P_2)\}$$

$$size_M(P_1 \mid P_2) \overset{\mathsf{def}}{=} size_M(P_1) + size_M(P_2)$$

$$size_M(\mathtt{rec}\ X\ .\ P) \overset{\mathsf{def}}{=} size_M(P)$$

$$size_M(\mathtt{watch}\, ev\ \mathtt{do}\ P\{Q\}) \overset{\mathsf{def}}{=} size_M(P)$$

As mentioned previously, the function $size_M(P)$ yields a bound for the number of steps that a configuration $\langle P, M, E\rangle$ may execute before reaching a state of suspension or termination. Note that the set of events E is irrelevant for this measure. Indeed, in our calculus (unlike in other SRP languages), the set of events only plays a role at the end of instants, and does not affect the reduction relation.

Definition 10.10 (Configuration Instantaneous Size). Function $size_M(P)$ induces a function $Size(C)$ on configurations, defined by:

$$Size(\langle P, M, E\rangle) \overset{\mathsf{def}}{=} size_M(P) \qquad\qquad Size((\nu s)C) \overset{\mathsf{def}}{=} Size(C)$$

Although partial (because $size_M(P)$ is partial), the function $Size(C)$ is always defined for reachable configurations C, as established by the following lemma:

Lemma 10.11. *Let C be a reachable configuration. Then $Size(C)$ is defined.*

Proof. We distinguish two cases, depending on whether C is initial or reachable in at least one step.

(1) Let $C = \langle P, \emptyset, \emptyset\rangle$ where $P = a[1](\alpha_1).P_1 \mid ... \mid a[n](\alpha_n).P_n \mid \bar{a}[n]$. Then it is immediate to see that $Size(\langle P, \emptyset, \emptyset\rangle) = size_\emptyset(P)$ is defined.

(2) Let $C = (\nu s)\langle P, M, E\rangle$.There are only two possible cases where $\mathit{size}_M(P)$ may not be defined, namely the I/O cases $P = s[\mathsf{p}]!\langle e\rangle.Q$ and $P = s[q]?(\mathsf{p}, x).Q$, when $s[\mathsf{p}] \notin \mathit{dom}(M)$. However, this cannot happen, since C is derived from an initial configuration

$$C_0 = \langle a[1](\alpha_1).P_1 \mid ... \mid a[n](\alpha_n).P_n \mid \bar{a}[n], \emptyset, \emptyset\rangle$$

whose first reduction is necessarily of the form:

$$C_0 \longrightarrow (\nu s)\langle P_1\{^{s[1]}/_{\alpha_1}\} \mid \ \dots \ \mid P_n\{^{s[n]}/_{\alpha_n}\}, M_s^{\emptyset}, \emptyset\rangle = C_1$$

By definition $s[i] \in \mathit{dom}(M_s)$ for each $i = 1, \dots, n$. Then we may conclude, since $C_1 \longrightarrow^* C$ is deduced using rules different from [Init] and all these rules preserve $\mathit{dom}(M_s)$.

□

In the forthcoming proofs, we shall also use the following property:

Property 10.12. For any process P and memory M:

$$\mathit{size}_M(\mathsf{rec}\ X\,.\,P) = \mathit{size}_M(P\{^{(\mathsf{pause.}\ \mathsf{rec}\ X\,.\,P)}/_X\})$$

Proof. Easy consequence of Def. 10.9, since $\mathit{size}_M(\mathsf{rec}\ X\,.\,P) = \mathit{size}_M(P)$ and for any process Q, we have $\mathit{size}_M(\mathsf{pause.}\ Q) = 0 = \mathit{size}_M(X)$. □

Before proving reactivity we will present some auxiliary results. We first prove that $\mathit{size}_M(P)$ decreases at each step of execution during an instant:

Lemma 10.13 (Size Reduction During Instantaneous Execution). *Let C be a reachable configuration. Then:*

$$C \longrightarrow C' \ \Rightarrow\ \mathit{Size}(C) > \mathit{Size}(C')$$

Proof. We distinguish two cases, depending on whether C is an initial configuration or not.

(1) Let $C = \langle P, M, E\rangle$. Then $C \longrightarrow C' = (\nu s)\langle P', M', E'\rangle$ is deduced by Rule [Init]. Here:

$$C = \langle a[1](\alpha_1).P_1 \mid ... \mid a[n](\alpha_n).P_n \mid \bar{a}[n], \emptyset, \emptyset\rangle$$

and $C' = (\nu s)\langle P_1\{^{s[1]}/_{\alpha_1}\} \mid \ \dots \ \mid P_n\{^{s[n]}/_{\alpha_n}\}, M_s^{\emptyset}, \emptyset\rangle$. Then we may conclude, since $\mathit{size}_M(P) = \sum_{i=1}^{n} \mathit{size}_M(a[i](\alpha_i).P_i) = n + \sum_{i=1}^{n} \mathit{size}_{M_s^{\emptyset}}(P_i\{s[i]/\alpha_i\}) = n + \mathit{size}_{M_s^{\emptyset}}(P') > \mathit{size}_{M_s^{\emptyset}}(P')$.

(2) Let $C = (\nu s)\langle P, M, E\rangle$. In this case we have $(\nu s)\langle P, M, E\rangle \longrightarrow (\nu s)\langle P', M', E'\rangle$ if and only $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$, and thus it is enough to prove the following statement:

If $(\nu s)\langle P, M, E\rangle$ is reachable then $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$ implies $\mathit{size}_M(P) > \mathit{size}_{M'}(P')$

To prove this statement we proceed by induction on the inference of the transition $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$, and case analysis on the last rule used in the inference. For details see App. F.1.

□

We define now a specific notion of guardedness for recursive processes, which will be used in the proofs of the next two lemmas. In the sequel, a recursive process will often be referred to as "recursive call" or simply "call". A recursive call that is guarded by a `pause` statement is said to be *pause-guarded*. Formally:

Definition 10.14 (Pause-Guardedness of Recursive Calls). A recursive call $\texttt{rec}X\,.\,Q$ is *pause-guarded* in process P if it appears within some subprocess $\texttt{pause}.\,P'$ of P. It is called *pause-unguarded* in P otherwise.

The following lemma establishes that every reachable configuration $\langle P, M, E\rangle$ immediately converges if and only if its size is equal to 0.

Lemma 10.15 (Immediate Convergence of 0-*size* **Configurations).** *Let C be a reachable configuration. Then*

$$(\mathit{Size}(C) = 0) \;\;\Leftrightarrow\;\; C \overset{\ddagger}{\curlyvee}$$

Proof. Note that C cannot be an initial configuration, since in this case we would have $\mathit{Size}(C) > 0$. Hence $C = (\nu s)\langle P, M, E\rangle$. Since

$$\mathit{Size}((\nu s)\langle P, M, E\rangle) = \mathit{Size}(\langle P, M, E\rangle) = \mathit{size}_M(P)$$

and $(\nu s)\langle P, M, E\rangle \overset{\ddagger}{\curlyvee} \;\Leftrightarrow\; \langle P, M, E\rangle \overset{\ddagger}{\curlyvee}$, it is enough to prove the statement for $C = \langle P, M, E\rangle$. We prove each side of the biconditional in turn. For details see App. F.1. □

There are two reasons why reactivity could fail in our calculus: 1) a process that loops forever during an instant - what is generally called an *instantaneous loop* in SRP [TS05]; 2) a deadlock due to a mismatch between a participant and the memory: this happens when a participant p in a session s wants to broadcast a message while the output buffer $s[\mathsf{p}] : m$ is not in the memory, or to receive a message from participant q while the output buffer $s[\mathsf{q}] : m$ is not in the memory. Our semantic rule for recursion is designed to prevent instantaneous loops, and will be the key for proving Reactivity (Theorem 10.18). The absence of mismatches between participants and the memory is guaranteed by the reachability assumption. The following lemma establishes that reachable configurations are deadlock-free.

Lemma 10.16 (Deadlock freedom). *Let C be a reachable configuration. Then*

$$\textit{either } C \overset{\ddagger}{\curlyvee} \textit{ or } \exists\, C'\,.\, C \longrightarrow C'$$

Proof. We distinguish two cases, depending on whether C is an initial configuration or not.

(1) Let $C = \langle P, M, E\rangle$ be an initial configuration. Then there is a reduction $C \longrightarrow C'$ with $C' = (\nu s)\langle P', M', E'\rangle$ deduced by Rule [Init].

(2) Let $C = (\nu s)\langle P, M, E\rangle$. Then C reduces if and only if $\langle P, M, E\rangle$ reduces and $C \overset{\ddagger}{\curlyvee}$ if and only if $\langle P, M, E\rangle \overset{\ddagger}{\curlyvee}$. Hence it is enough to prove the property for $\langle P, M, E\rangle$. We proceed by induction on the structure of P. Note that the reachability assumption rules out the cases $P = X$, $P = \bar{a}[n]$ and $P = a[\mathsf{p}](\alpha).Q$. For details see App. F.1.

□

We are now ready to prove two (increasingly strong) reactivity results for reachable configurations:

(1) Standard *reactivity*, which amounts to the convergence of all computations of C within an instant.

(2) *Bounded reactivity*, which gives a bound for the number of steps of such converging computations.

Theorem 10.17 (Reactivity). *Let C be a reachable configuration. Then $C \Downarrow$.*

Proof. Every sequence of consecutive reductions from a configuration C must be finite, given that $\mathit{Size}(C)$ is defined by Lem. 10.11, and it strictly decreases along execution by Lem. 10.13. Moreover, no derivative of C may be deadlocked, by Lem. 10.16. Hence the size of the configuration eventually becomes 0 Lem. 10.15. □

We proceed now to prove bounded reactivity, namely that every reachable configuration C instantaneously converges in a number of steps that is bounded by $\mathit{Size}(C)$.

Theorem 10.18 (Bounded Reactivity). *Let C be a reachable configuration. Then*

$$\exists\, n \leq \mathit{Size}(C)\,.\, C \Downarrow_n$$

Proof. We distinguish two cases, depending on whether C is an initial configuration or not. For details see App. F.1. □

10.4 Types for MRS

In this section we present the session type system for MRS. We show that typability implies the classical properties of session calculi, namely the absence of communication errors (communication safety) and the conformance to the session protocol (session fidelity). Furthermore, our type system enforces the following properties, which are specific to our synchronous reactive setting and will be discussed in more detail later:

P1. *Output persistence*: Every participant broadcasts exactly one message during every time instant.

P2. *Input timeliness*: Every unguarded input is matched by an output during the current instant, if not preceded by another input with equal source and target, or during the next instant, if not preempted.

10.4.1 Global and Local Types

Our calculus allows multiparty communication [HYC08, CDPY15]. Hence, typing relies on *global types* to describe communication protocols and on *local types* to describe the contributions of protocol participants.

Sorts	S	$::=$	bool $\mid$ int $\mid$ ...	
Global types	G	$::=$	$\mathtt{p}{\uparrow}\langle S,\Pi\rangle.G$	*communication*
		$\mid$	$\mathtt{pause}.G$	*explicit pause*
		$\mid$	$\mathtt{tick}.G$	*implicit pause*
		$\mid$	$\mathtt{watch}\ ev\ \mathtt{do}\ G\ \mathtt{else}\ G$	*global watch*
		$\mid$	$\mathbf{t}$	*recursive variable*
		$\mid$	$\mu\mathbf{t}.G$	*recursion*
		$\mid$	$\mathtt{end}$	*end*
Local Types	T	$::=$	$!S.T$	*send*
		$\mid$	$?(\mathtt{p},S).T$	*receive*
		$\mid$	$\mathtt{pause}.T$	*explicit pause*
		$\mid$	$\mathtt{tick}.T$	*implicit pause*
		$\mid$	$\langle T,T'\rangle^{ev}$	*local watch*
		$\mid$	$\mu\mathbf{t}.T$	*recursion*
		$\mid$	$\mathbf{t}$	*recursive variable*
		$\mid$	$\mathtt{end}$	*end*
Message Types	ϑ	$::=$	$\mathtt{void}\ \mid (S,\Pi)$	

Figure 10.10: Sorts, Global types, Local types and Message types.

Our type syntax, given in Fig. 10.10, uses sorts, ranged over by $S, S', \ldots$, global types ranged over by $G, G', \ldots$, local types, ranged over by $T, T', \ldots$, and type variables, ranged over by $\mathbf{t}, \mathbf{t}', \ldots$. Sorts denote basic types such as int and bool and type variables are used when defining recursive types. We recall that participants are denoted by $\mathtt{p}, \mathtt{q}, \ldots$ or by natural numbers. Similarly, sets of participants are denoted by Π, Π'. A peculiarity of our type system is that for every sort S we assume a default value d_S, representative of the particular basic type S. We present the syntax of both global and local types below.

Global types:

- Type $\mathtt{p}\uparrow\langle S,\Pi\rangle.G$ represents a participant p broadcasting a message of sort S, with participants in Π as intended receivers; subsequently, the interaction continues as specified by G. As a well-formedness condition for the type above, we require $\mathtt{p} \notin \Pi$.

- Type $\mathtt{pause}.G$ stipulates that all participants should jointly move to the next time instant in order to execute protocol G.

- Type $\mathtt{tick}.G$ represents an implicit pause arise from the semantics of processes. In a sense, it can be said that `tick` is "runtime" type, which is added dynamically by the saturation function that will be presented in Fig. 10.11.

- Type $\texttt{watch}\ ev\ \texttt{do}\ G\ \texttt{else}\ G'$ represents a protocol which has an alternative behavior. Intuitively, protocol G is executed until the end of the current instant; then, if event ev has been emitted, the governing protocol at the next instant will be G', otherwise it will be the continuation of G. Note that if G' becomes the governing protocol, protocol G is pre-empted.

- Type end represents the terminated protocol.

Local types:

- The *send* type $!S.T$ indicates the broadcast of a value of sort S, followed by the behavior described by T.

- The *receive* type $?(\mathsf{p}, S).T$ describes the reception of a value of sort S sent by participant p, followed by the behavior described by T.

- The *pause* type $\texttt{pause}.T$ signals the change of time instant, followed by the behavior described by T.

- Type $\texttt{tick}.T$ corresponds to the projection of global ticks.

- The *local watch* type $\langle T, T'\rangle^{ev}$ is meant to be assigned to a participant that behaves as specified by type T until the end of instant; then, if event ev has appeared, it will behave according to type T', otherwise according to the continuation of T.

We assume recursive types $\mu\mathbf{t}.G$ and $\mu\mathbf{t}.T$ to be contractive (i.e., type variables only appear under the prefixes). Furthermore, as we have done in previous sections, we take an equi-recursive view of types, meaning that we do not distinguish between $\mu\mathbf{t}.G$ (resp. $\mu\mathbf{t}.T$) and its unfolding $G\{^{\mu\mathbf{t}.G}/_{\mathbf{t}}\}$ (resp. $T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\}$). As a consequence, we never consider types of the form $\mu\mathbf{t}.T$ in typing rules. Indeed, whenever we find a type $\mu\mathbf{t}.T$ in a typing rule, we pick another type equal to it (i.e., its unfolding $T\{^{\mu\mathbf{t}.T}/_{\mathbf{t}}\}$) [Pie02].

Example 10.19. We now present some examples of global and local types that can be written using the syntax in Fig. 10.10. Assume participants p, q, r:

$$\mathsf{p}\uparrow\langle S, \{\mathsf{q}, \mathsf{r}\}\rangle.\mathsf{pause}.\mathsf{r}\uparrow\langle S, \{\mathsf{p}, \mathsf{q}\}\rangle.\mathsf{end} \qquad \mathsf{pause}.\mathsf{r}\uparrow\langle S, \{\mathsf{p}, \mathsf{q}\}\rangle.\mathsf{end}$$
$$\mu\mathbf{t}.\mathsf{r}\uparrow\langle S, \{\mathsf{p}, \mathsf{q}\}\rangle.\mathbf{t} \qquad \mu\mathbf{t}.\mathsf{pause}.\mathbf{t}$$
$$!S.?(\mathsf{p}, S').\mathsf{end} \qquad \mu\mathbf{t}.!S.!S'.\mathbf{t}$$

△

$$
\begin{array}{ll}
\mathsf{S}_{\mathcal{R}}(\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.G',\mathcal{P})=\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.\mathsf{S}_{\mathcal{R}}(G',\mathcal{P}\cup\{\mathsf{p}\}) & \text{if } \mathsf{p}\notin\mathcal{P}\\
\mathsf{S}_{\mathcal{R}}(\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.G',\mathcal{P}) =\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\emptyset\rangle.\mathtt{tick}. & \text{if } \mathsf{p}\in\mathcal{P} \text{ and}\\
\qquad\qquad \mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G')}}(G',\{\mathsf{p}\}) & \mathcal{R}\setminus\mathcal{P}=\{\mathsf{p}_1,\ldots,\mathsf{p}_n\}\\[2ex]
\mathsf{S}_{\mathcal{R}}(\mathsf{pause}.G',\mathcal{P})=\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\emptyset\rangle.\mathsf{pause}.\mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset) & \text{with } \mathcal{R}\setminus\mathcal{P}=\{\mathsf{p}_1,\ldots,\mathsf{p}_n\}\\
\mathsf{S}_{\mathcal{R}}(\mathtt{tick}.G',\mathcal{P})=\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\emptyset\rangle.\mathsf{pause}.\mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset) & \text{with } \mathcal{R}\setminus\mathcal{P}=\{\mathsf{p}_1,\ldots,\mathsf{p}_n\}\\[2ex]
\mathsf{S}_{\mathcal{R}}(\mathsf{end},\mathcal{P})=\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\emptyset\rangle.\mathsf{end} & \text{with } \mathcal{R}\setminus\mathcal{P}=\{\mathsf{p}_1,\ldots,\mathsf{p}_n\}\\[2ex]
\mathsf{S}_{\mathcal{R}}(\mathtt{watch}\ ev\ \mathsf{do}\ G'\ \mathtt{else}\ G'',\mathcal{P}) = s\mathtt{watch}\ ev\ \mathsf{do}\ \mathsf{S}_{\mathcal{R}}(G',\mathcal{P}) & \\
\qquad\qquad\qquad \mathtt{else}\ \mathsf{S}_{\mathsf{Part(G'')}}(G'',\emptyset) & \\[2ex]
\mathsf{S}_{\mathcal{R}}(\mu\mathbf{t}.G',\emptyset)=\mu\mathbf{t}.\ \mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset) & \\
\mathsf{S}_{\mathcal{R}}(\mu\mathbf{t}.G',\mathcal{P})=\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\emptyset\rangle.\mathtt{tick}.\mu\mathbf{t}.\ \mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset) & \text{with } \mathcal{R}\setminus\mathcal{P}=\{\mathsf{p}_1,\ldots,\mathsf{p}_n\}\\[2ex]
\mathsf{S}_{\mathcal{R}}(\mathbf{t},\mathcal{P})=\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\emptyset\rangle.\mathsf{pause}.\mathbf{t} & \text{with } \mathcal{R}\setminus\mathcal{P}=\{\mathsf{p}_1,\ldots,\mathsf{p}_n\}
\end{array}
$$

Figure 10.11: Saturation of global types.

10.4.2 *Projection and Saturation*

As usual in multiparty session calculi, global and local types are related by the notion of *projection*. Intuitively, the projection of a global type G onto its participants generates the local types for every participant using the information given by G.

In our reactive setting, however, projection requires a pre-processing phase in which the global type is modified by adding the necessary implicit pauses. As hinted before, `tick` represents implicit pauses, which are induced by the semantics, rather than by a pause explicitly written in the protocol specification. This pre-processing phase is called *saturation* and besides adding the necessary implicit pauses, it adds outputs that may be missing to ensure output persistence.

Before formally introducing saturation and projection, we define some auxiliary notation. We write $\mathsf{Part}(G)$ to represent the set of participants declared in G. We also find it useful to write $\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\Pi\rangle.G$ to abbreviate the global type $\mathsf{p}_1\!\uparrow\!\langle S,\Pi\rangle.\cdots.\mathsf{p}_n\!\uparrow\!\langle S,\Pi\rangle.G$, describing n consecutive broadcasts of messages of the same sort S from each of the participants $\mathsf{p}_1,\ldots,\mathsf{p}_n$ to the set of participants in Π. Moreover, whenever $n=0$, $\mathsf{p}_{\{1,n\}}\!\uparrow\!\langle S,\Pi\rangle.G$ denotes the global type G.

We now introduce saturation (cf. Fig. 10.11) and projection (cf. Fig. 10.12). As hinted above, the saturation function makes the global type reflect precisely the slicing into instants of the protocol's behavior. Intuitively, a global type is saturated as follows: first, the saturation function identifies the correct instant slicing in the protocol adding the necessary `tick`. Next, the function saturates the global type with outputs, guaranteeing that in each instant all participants broadcast a message (even if it is not received by anyone). Formally, the function $\mathsf{S}_{\mathcal{R}}(G,\mathcal{P})$, takes a global type G , the set of currently active participants, written $\mathcal{R}$, and a set that collects the participants of $\mathcal{R}$ that have already sent a message in the current instant, denoted $\mathcal{P}\subseteq\mathcal{R}$.

Our type system will then require *saturated projections*, which simply stand for the

$$(\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.G')\restriction \mathsf{q} = \begin{cases} !S.(G'\restriction \mathsf{q}) & \text{if } \mathsf{q}=\mathsf{p} \\ ?(\mathsf{p},S).(G'\restriction \mathsf{q}) & \text{if } \mathsf{q}\in\Pi, \\ G'\restriction \mathsf{q} & \text{otherwise.} \end{cases}$$

$$(\mathtt{pause}.G')\restriction \mathsf{q} = \begin{cases} \mathsf{end} & \text{if } \mathsf{q}\notin \mathsf{Part}(G') \wedge \text{no type variable } \mathbf{t} \text{ occurs in } G' \\ \mathtt{pause}.(G'\restriction \mathsf{q}) & \text{otherwise.} \end{cases}$$

$$(\mathtt{tick}.G')\restriction \mathsf{q} = \begin{cases} \mathsf{end} & \text{if } \mathsf{q}\notin \mathsf{Part}(G') \wedge \text{no type variable } \mathbf{t} \text{ occurs in } G' \\ \mathtt{tick}.(G'\restriction \mathsf{q}) & \text{otherwise} \end{cases}$$

$$(\mathtt{watch}\ ev\ \mathtt{do}\ G_1\ \mathtt{else}\ G_2)\restriction \mathsf{q} = \langle G_1\restriction \mathsf{q}, G_2\restriction \mathsf{q}\rangle^{ev}$$

$$(\mu\mathbf{t}.G')\restriction \mathsf{q} = \begin{cases} \mu\mathbf{t}.(G'\restriction \mathsf{q}) & \text{if } \mathsf{q} \text{ or any type variable } \mathbf{t}' \text{ occur in } G' \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$\mathbf{t}\restriction \mathsf{q} = \mathbf{t} \quad \mathsf{end}\restriction \mathsf{q} = \mathsf{end}$$

Figure 10.12: Projection of global types onto participants.

projection of the saturated global type. Intuitively, the *saturated projection* of G onto q, denoted $G\lfloor_{\mathsf{q}}$, yields a local type representing q's involvement in the protocol described by G. It is obtained in two steps: first, G is saturated as described in Fig. 10.11; then the resulting global type is projected onto q as described in Fig. 10.12:

$$G\lfloor_{\mathsf{q}} = (\mathsf{S}_{\mathsf{Part(G)}}(G,\emptyset)\restriction \mathsf{q}) \tag{10.1}$$

To illustrate our definitions of projection and saturation, let us look back at the auction example of § 10.2.

Example 10.20 (Types for the *Auction* Process). Recall that the set of participants of the *Auction* process is $\{\mathsf{A},\mathsf{B}_1,\ldots,\mathsf{B}_n\}$, where A represents the *Auctioneer* and B_i the process $Bidder_i$ $(1\le i\le n)$. The global type G is as follows:

$$\begin{aligned} G = {} & \mathtt{watch}\ bis\ \mathtt{do}\ \mu\mathbf{t}.\mathsf{B}_1\!\uparrow\!\langle \mathsf{int},\{\mathsf{A}\}\rangle.\ \cdots\ .\mathsf{B}_n\!\uparrow\!\langle \mathsf{int},\{\mathsf{A}\}\rangle.\mathsf{A}\!\uparrow\!\langle \widetilde{\mathsf{int}},\{\mathsf{B}_1,\ldots,\mathsf{B}_n\}\rangle.\mathbf{t} \\ & \mathtt{else}\ \mathsf{A}\!\uparrow\!\langle \mathsf{int},\{\mathsf{B}_1,\ldots,\mathsf{B}_n\}\rangle.\mathsf{B}_1\!\uparrow\!\langle \mathsf{string},\emptyset\rangle.\ \cdots\ .\mathsf{B}_n\!\uparrow\!\langle \mathsf{string},\emptyset\rangle.\mathsf{end} \end{aligned}$$

Then, the saturated global type $\mathcal{G} = \mathsf{S}(G,\emptyset)$ is built according to Fig. 10.11, as follows:

$$\begin{aligned} \mathsf{S}(G,\emptyset) = {} & \mathtt{watch}\ bis\ \mathtt{do}\ \mu\mathbf{t}.\mathsf{B}_1\!\uparrow\!\langle \mathsf{int},\{\mathsf{A}\}\rangle.\ \cdots\ .\mathsf{B}_n\!\uparrow\!\langle \mathsf{int},\{\mathsf{A}\}\rangle.\mathsf{A}\!\uparrow\!\langle \widetilde{\mathsf{int}},\{\mathsf{B}_1,\ldots,\mathsf{B}_n\}\rangle.\mathsf{pause}.\mathbf{t} \\ & \mathtt{else}\ \mathsf{A}\!\uparrow\!\langle \mathsf{int},\{\mathsf{B}_1,\ldots,\mathsf{B}_n\}\rangle.\mathsf{B}_1\!\uparrow\!\langle \mathsf{string},\emptyset\rangle.\ \cdots\ .\mathsf{B}_n\!\uparrow\!\langle \mathsf{string},\emptyset\rangle.\mathsf{end} \end{aligned}$$

The only difference with respect to G is the addition of a pause before the recursion variable **t**. Finally, the saturated projections of G (i.e., the projections of $\mathcal{G}$) onto participants are as follow:

$$G\lfloor_{\mathsf{A}} = \langle \mu\mathbf{t}_1.?(\mathsf{B}_1,\mathsf{int}).\ \cdots\ .?(\mathsf{B}_n,\mathsf{int}).!\widetilde{\mathsf{int}}.\mathsf{pause}.\mathbf{t}_1, !\mathsf{int}.\mathit{end}\rangle^{bis}$$
$$G\lfloor_{\mathsf{B}_i} = \langle \mu\mathbf{t}_2.!\mathsf{int}.?(\mathsf{A},\widetilde{\mathsf{int}}).\mathsf{pause}.\mathbf{t}_2, ?(\mathsf{A},\mathsf{int}).!\mathsf{string}.\mathit{end}\rangle^{bis}$$

△

10.4.3 Type System

Typing judgments for our type system rely on three kinds of typing environments: *standard environments* given by Γ, Γ', *session environments* ranged over by Δ, Δ', and *message environments* denoted by Θ. Standard environments, map variables to sort types, service names to global types, and process variables to local types. Formally:

$$\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, a : G \mid \Gamma, X : T$$

We write $\Gamma, x : S$ only if x does not occur in $dom(\Gamma)$, where $dom(\Gamma)$ denotes the domain of Γ, i.e., the set of identifiers occurring in Γ. We adopt the same convention for $\Gamma, a : G$ and $\Gamma, X : T$.

Session environments assign local types to channels occurring in processes, while message environments, assign message types (cf. Fig. 10.10) to channels occurring in memories. More specifically, a message environment assigns to a channel c in the memory the type ϑ of the message carried by c, namely the type `void` if no message has been sent on c, and the type (S, Π) if a message of type S has been sent on c and has been read by the participants in Π.

The syntax of message types ϑ is given in Fig. 10.10. Then, session and message environments are formally defined by the following grammars:

$$\Delta ::= \emptyset \mid \Delta, c : T \qquad\qquad \Theta ::= \emptyset \mid \Theta, c : \vartheta$$

For $\Delta, c : T$ and $\Theta, c : \vartheta$ we use the same conventions as for Γ, meaning that a session environment $\Delta, c : T$ (resp. a message environment $\Theta, c : \vartheta$) is only well-defined if $c \notin dom(\Delta)$ (rep. $c \notin dom(\Theta)$). Thus, a session environment Δ_1, Δ_2 is only well-defined if $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$.

Following the same notation introduced for memories, we write $\Theta^{\emptyset}$ to represent the message environment obtained from Θ by turning all message types to `void`. Intuitively, if Θ types memory M then $\Theta^{\emptyset}$ types memory $M^{\emptyset}$. Formally, $\Theta^{\emptyset} =_{\mathsf{def}} \bigcup_{c \in dom(\Theta)} \{c : \mathsf{void}\}$.

Below, we give two auxiliary definitions. The first one introduces notations for referring to session environments that contain only ended behaviors and environments that contains at least one "live" behavior. The second one defines a way to extract the "active" participants in the current instant from the memory M and memory environment Θ.

Definition 10.21 (Live and Terminated Session Environments). A session environment Δ is said to be *live* if $c : T \in \Delta$ implies $T \neq \mathsf{end}$, and *terminated* if $c : T \in \Delta$ implies $T = \mathsf{end}$. Any session environment Δ may be partitioned in two session environments Δ^{live} and Δ^{end} defined by:

$$\begin{aligned} \Delta^{live} &\overset{\mathsf{def}}{=} \{c : T \in \Delta \mid T \neq \mathsf{end} \wedge T \neq \langle \mathsf{end}, T \rangle^{ev}\} \\ \Delta^{end} &\overset{\mathsf{def}}{=} \{c : T \in \Delta \mid T = \mathsf{end} \vee T = \langle \mathsf{end}, T \rangle^{ev}\} \end{aligned}$$

Definition 10.22 (Visible Domain of Memories and Message Environments). The *visible domain* of memories and message environments is defined by:

$$\begin{aligned} vdom(M) &= \{c \mid c : m \in M \ \wedge \ m \neq \epsilon\} \\ vdom(\Theta) &= \{c \mid c : \vartheta \in \Theta \ \wedge \ \vartheta \neq \mathsf{void}\} \end{aligned}$$

$$\lfloor\textsc{Service}\rfloor \quad \Gamma, a : G \vdash a : G \qquad\qquad \lfloor\textsc{ProcVar}\rfloor \quad \Gamma, X : T \vdash X : S\,T$$

Figure 10.13: Typing rules for services and process variables.

$$\lfloor\textsc{BoolVal}\rfloor \quad \Gamma \vdash \texttt{tt}, \texttt{ff} : \mathsf{bool} \qquad \lfloor\textsc{IntVal}\rfloor \quad \Gamma \vdash 1, 2, \ldots : \mathsf{int} \qquad \lfloor\textsc{DVal}\rfloor \quad \Gamma \vdash d_S : S$$

$$\lfloor\textsc{Var}\rfloor \; \Gamma, x : S \vdash x : S \qquad \lfloor\textsc{And}\rfloor \; \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \;\mathsf{and}\; e_2 : \mathsf{bool}} \qquad \lfloor\textsc{Sum}\rfloor \; \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

Figure 10.14: Typing rules for expressions.

$$\lfloor\textsc{VoidMsg}\rfloor \; \Gamma \vdash c : \epsilon \triangleright c : \texttt{void} \qquad \lfloor\textsc{FullMsg}\rfloor \; \frac{\Gamma \vdash v : S}{\Gamma \vdash c : (v, \Pi) \triangleright c : (S, \Pi)}$$

$$\lfloor\textsc{EmptyMem}\rfloor \; \Gamma \vdash \emptyset \triangleright \emptyset \qquad \lfloor\textsc{MergeMem}\rfloor \; \frac{\Gamma \vdash M \triangleright \Theta \quad \Gamma \vdash c : m \triangleright c : \vartheta}{\Gamma \vdash M \cup \{c : m\} \triangleright \Theta, \{c : \vartheta\}}$$

Figure 10.15: Typing rules for memories.

Our type system uses three kinds of type judgments. The first one, used for typing expressions, is defined by the rules in Fig. 10.14 and has the form:

$$\Gamma \vdash e : S$$

where Γ represents a standard environment, e and expression and S a sort. The second one, used for typing memories, is defined by the rules in Fig. 10.15 and has the form:

$$\Gamma \vdash M \triangleright \Theta$$

where Θ is a message environment associating a message type with each channel in M. Finally, the third kind of judgment is used for typing configurations and has the form:

$$\Gamma \vdash C \triangleright \langle \Delta \diamond \Theta \rangle$$

where $\langle \Delta \diamond \Theta \rangle$ is called a *configuration environment*. The reason for using configuration environments is because the state of the memory directly impact the behavior of the process. Thus, it becomes necessary for types to have knowledge about the memory at every point of the typing derivation.

Intuitively, if $\Gamma \vdash \langle P, M, E \rangle \triangleright \langle \Delta \diamond \Theta \rangle$, then Δ is a session environment typing the channels of P in memory M, and Θ is a message environment typing the messages of M.

Before presenting the typing rules for configurations, we introduce some auxiliary notions and results. We say that a local type is *output granting* if the first pause in it (if any) is preceded by an output.

Definition 10.23 (Output-Granting Type). A local type T is *output-granting* if it satisfies the predicate $\mathsf{OG}(T)$ defined by:

$$\mathsf{OG}(T) \overset{\mathsf{def}}{=} \begin{cases} \mathtt{tt} & \text{if } T = !S.T' \\ \mathsf{OG}(T') & \text{if } T = ?(\mathsf{p}, S).T' \vee T = \langle T', T'' \rangle^{ev} \vee T = \mu \mathbf{t}.T' \\ \mathtt{ff} & \text{otherwise} \end{cases}$$

The predicate is extended to session environments by letting $\mathsf{OG}(\Delta)$ if $\mathsf{OG}(T)$ for all $c{:}T$ in Δ.

We can then show that local types that come from the projection of a saturated global type are output granting.

Lemma 10.24 (Correctness of Saturation with Respect to $\mathsf{OG}(T)$). *Let G be a global type such that $|\mathsf{Part}(G)| \geq 2$, p be a participant, and $\mathcal{P}$ be a set of participants such that (1) $\mathcal{P} \subseteq \mathsf{Part}(G)$, and (2) $\mathsf{Part}(G) \setminus \mathcal{P} \neq \emptyset$. Then, $\mathsf{OG}(\mathsf{S}_{\mathsf{Part(G)}}(G, \mathcal{P}) \upharpoonright \mathsf{p})$ holds for every $\mathsf{p} \in \mathsf{Part}(G) \setminus \mathcal{P}$.*

Proof. By induction on the structure of G. The base cases are $G = \mathsf{end}$ and $G = \mathbf{t}$, which are vacuously true since $|\mathsf{Part}(\mathsf{end})| = |\mathsf{Part}(\mathbf{t})| = 0$. For the inductive step we assume that the property holds for a global type G' and prove the statement for every type that contains G' as a sub-expression. There are 5 inductive cases. For details see App. F.2. □

Corollary 10.25 (Projection Implies $\mathsf{OG}(\cdot)$). *For any global type G such that $\mathsf{Part}(G) \geq 2$ and any participant $\mathsf{p} \in \mathsf{Part}(G)$, $\mathsf{OG}(G\lfloor_{\mathsf{p}})$ holds.*

Proof. The proof follows from Lem. 10.24. Notice that in this case $\mathcal{P} = \emptyset$, hence $\mathsf{Part}(G) \setminus \mathcal{P} = \mathsf{Part}(G)$.

□

Next, we define the *pair composition* of session types, which intuitively generates end or $\langle T_1, T_2 \rangle^{ev}$, depending on the value of T_1.

Definition 10.26. The *pair composition* of session types T_1 and T_2 under event ev, written $T_1 \star_{ev} T_2$, is defined by:

$$T_1 \star_{ev} T_2 \overset{\mathsf{def}}{=} \begin{cases} T_1 & if \;\; T_1 = \mathsf{end} \\ \langle T_1, T_2 \rangle^{ev} & otherwise. \end{cases}$$

Abusing notation, we extend this definition to session environments by writing $\Delta_1 \star_{ev} \Delta_2$.

Given that our typing rules are defined over session environments and that our semantics induce suspension points for all processes in parallel inside a configuration, we define functions $\mathsf{pause}(\cdot)$ and $\mathsf{tick}(\cdot)$ to extract all the suspended sessions of a session environment.

Definition 10.27. Given a session environment Δ, we write $\mathsf{pause}(\Delta)$ to denote the environment defined as $\mathsf{pause}(\Delta) \overset{\mathsf{def}}{=} \{c : \mathtt{pause}.T \mid c : T \in \Delta\}$. Analogously $\mathsf{tick}(\Delta)$ denotes the environment defined as $\mathsf{tick}(\Delta) \overset{\mathsf{def}}{=} \{c : \mathtt{tick}.T \mid c : T \in \Delta\}$.

We next introduce generalized types for session channels occurring in configurations. Note that in a configuration, channels may occur on the process side, on the memory side, or on both sides. Since our semantics allows at most one message to be sent on each channel during an instant, both our message types and our generalized types are simpler than those of standard asynchronous multiparty session calculi [CDPY15]. In particular, in our calculus the syntax of generalized types coincides with that of local types.

Definition 10.28 (Generalized Types). A *generalized type* $\mathcal{T}$ is a local type T or a send type (extracted from a message type ϑ) followed by a local type T. Moreover, let $\langle\Delta\diamond\Theta\rangle$ be a configuration environment and $s[\mathsf{p}] \in dom(\langle\Delta\diamond\Theta\rangle)$. Then the generalized type of $s[\mathsf{p}]$ in $\langle\Delta \diamond \Theta\rangle$ is given by:

$$\langle\Delta \diamond \Theta\rangle(s[\mathsf{p}]) \overset{\mathsf{def}}{=} \begin{cases} T & \text{if } s[\mathsf{p}] : T \in \Delta \ \wedge \ s[\mathsf{p}] \notin vdom(\Theta) \\ !S.\,T & \text{if } s[\mathsf{p}] : (S, \Pi) \in \Theta \ \wedge \\ & \quad (s[\mathsf{p}] : T \in \Delta \ \vee \ (s[\mathsf{p}] \notin dom(\Delta) \ \wedge \ T = \mathsf{end})\,) \\ \mathsf{end} & \text{if } s[\mathsf{p}] \notin dom(\Delta) \ \wedge \ s[\mathsf{p}] : \mathtt{void} \in dom(\Theta) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The generalized type $\langle\Delta \diamond \Theta\rangle(s[\mathsf{p}])$ represents the usage of channel $s[\mathsf{p}]$ in the configuration environment $\langle\Delta\diamond\Theta\rangle$: it is the concatenation of the send type extracted from the message sent by p in the current instant, if any, with the local type describing the remaining behavior of p.

Example 10.29 (Generalized Types). We present examples of generalized types below:

Let $\Delta = s[1] : \mathsf{end}$ and $\Theta = \emptyset$. Then $\langle\Delta \diamond \Theta\rangle(s[1]) = \mathsf{end}$.

Let $\Delta = \emptyset$ and $\Theta = s[1] : \mathtt{void}$. Then $\langle\Delta \diamond \Theta\rangle(s[1]) = \mathsf{end}$.

Let $\Delta = s[1] : \mathsf{end}$ and $\Theta = s[1] : \mathtt{void}$. Then $\langle\Delta \diamond \Theta\rangle(s[1]) = \mathsf{end}$.

Let $\Delta = s[1] :!S.\mathsf{end}$ and $\Theta = s[1] : \mathtt{void}$. Then $\langle\Delta \diamond \Theta\rangle(s[1]) =!S.\mathsf{end}$.

Let $\Delta = s[1] : \mathsf{end}$ and $\Theta = s[1] : (S, \Pi)$. Then $\langle\Delta \diamond \Theta\rangle(s[1]) =!S.\mathsf{end}$.

Let $\Delta = s[1] :!S.\mathsf{end}$ and $\Theta = s[1] : (S', \Pi)$. Then $\langle\Delta \diamond \Theta\rangle(s[1]) =!S'.!S.\mathsf{end}$.

△

Generalized types may be projected on participants (notation $\mathcal{T} \upharpoonright \mathsf{q}$), as described in Fig. 10.16. These projections are *anonymous local types* (local types where sender names are omitted), defined as follows:

$$\tau ::= \mathsf{end} \mid !S.\tau \mid ?S.\tau \mid \mathtt{pause}.\tau \mid \mathtt{tick}.\tau \mid \langle\tau, \tau'\rangle^{ev} \mid \mu\mathbf{t}.\tau \mid \mathbf{t}$$

Intuitively, the projection of the generalized type $\langle\Delta \diamond \Theta\rangle(s[\mathsf{p}])$ on q, namely $\langle\Delta \diamond \Theta\rangle(s[\mathsf{p}]) \upharpoonright \mathsf{q}$, describes the part of p's contribution to $\langle\Delta \diamond \Theta\rangle$ that concerns q.

We may now define a duality relation $\bowtie$ between projections of generalized types. Informally, duality holds when the inputs offered by one side are matched by the

$$(!S.T) \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} !S.\,(T \upharpoonright \mathsf{q}) \qquad (?(\mathsf{p}, S).T) \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} \begin{cases} ?S.(T \upharpoonright \mathsf{q}) & \text{if } \mathsf{q} = \mathsf{p}, \\ T \upharpoonright \mathsf{q} & \text{otherwise.} \end{cases}$$

$$(\mathsf{pause}.T) \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} \mathsf{pause}.(T \upharpoonright \mathsf{q}) \qquad (\mathtt{tick}.T) \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} \mathtt{tick}.(T \upharpoonright \mathsf{q})$$
$$(\langle T_1, T_2 \rangle^{ev}) \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} \langle T_1 \upharpoonright \mathsf{q}, T_2 \upharpoonright \mathsf{q} \rangle^{ev}$$

$$(\mu \mathbf{t}.T) \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} \begin{cases} \mu \mathbf{t}.(T \upharpoonright \mathsf{q}) & \text{if q occurs in } T \vee \text{any type variable } \mathbf{t}' \text{ occurs on } T \\ \mathsf{end} & \text{otherwise} \end{cases}$$
$$\mathbf{t} \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} \mathbf{t} \qquad \mathsf{end} \upharpoonright \mathsf{q} \overset{\mathsf{def}}{=} \mathsf{end}$$

Figure 10.16: Projection of generalized types on participants.

outputs offered by the other side. The requirement is weaker in the other direction, since outputs do not need to be matched by inputs in a broadcast setting (the actual receivers may range from none to all participants except the sender). Hence, two types may be dual although not completely symmetric. In this respect, we depart from standard session calculi, where the requirement is symmetric for inputs and outputs. Dual types are expected to have matching explicit and implicit pauses, as well as matching watching statements, whose types are required to be pairwise dual.

Definition 10.30. The *duality relation* between projections of generalized types is the minimal symmetric relation which satisfies:

$$\mathsf{end} \bowtie \mathsf{end} \qquad \mathbf{t} \bowtie \mathbf{t} \qquad \tau \bowtie \tau' \Rightarrow \mu\mathbf{t}.\tau \bowtie \mu\mathbf{t}.\tau' \qquad \tau \bowtie \tau' \Rightarrow !S.\tau \bowtie ?S.\tau'$$
$$\tau \bowtie \tau' \Rightarrow !S.\tau \bowtie \tau' \qquad \tau \bowtie \tau' \Rightarrow \mathsf{pause}.\tau \bowtie \mathsf{pause}.\tau'$$
$$\tau \bowtie \tau' \Rightarrow \mathtt{tick}.\tau \bowtie \mathtt{tick}.\tau'$$
$$\langle \mathsf{end}, \tau \rangle^{ev} \bowtie \mathsf{end} \qquad \tau_1 \bowtie \tau_3 \text{ and } \tau_2 \bowtie \tau_4 \Rightarrow \langle \tau_1, \tau_2 \rangle^{ev} \bowtie \langle \tau_3, \tau_4 \rangle^{ev}$$
$$\mathsf{end} \bowtie \tau \Rightarrow \mathsf{end} \bowtie \mathsf{pause}.\tau \qquad \mathsf{end} \bowtie \tau \Rightarrow \mathsf{end} \bowtie \mathtt{tick}.\tau$$

Notice that terminated types may be dual to non terminated ones, due to the clause $T \bowtie T' \Rightarrow !S.T \bowtie T'$ and to the last two clauses of the definition. However, such non terminated types can only be sequences of send types or explicit/implicit pauses followed by send types, as for instance in $\mathsf{end} \bowtie \mathsf{pause}.!S.!S'.\mathsf{end}$.

Example 10.31 (Dual Projections of Generalized Types). We present some examples of dual generalized types:

(1) Let $\Delta = s[1] : !S.\,\mathsf{end}, s[2] : ?(1, S).\mathsf{end}$ and $\Theta = s[1] : \mathsf{void}, s[2] : \mathsf{void}$.
Then $\langle \Delta \diamond \Theta \rangle(s[1]) \upharpoonright 2 = !S.\mathsf{end} \;\; \bowtie \;\; ?S.\mathsf{end} = \langle \Delta \diamond \Theta \rangle(s[2]) \upharpoonright 1$

(2) Let $\Delta = s[1] : \mathsf{end}, s[2] : ?(1, S).\mathsf{end}$ and $\Theta = s[1] : (S, \emptyset), s[2] : \mathsf{void}$.
Then $\langle \Delta \diamond \Theta \rangle(s[1]) \upharpoonright 2 = !S.\mathsf{end} \;\; \bowtie \;\; ?S.\mathsf{end} = \langle \Delta \diamond \Theta \rangle(s[2]) \upharpoonright 1$

(3) Let $\Delta = s[1] : \mathsf{end}, s[2] : ?(1, S).\mathsf{end}$ and $\Theta = s[1] : (S, \{2\}), s[2] : \mathsf{void}$.
Then $\langle \Delta \diamond \Theta \rangle(s[1]) \upharpoonright 2 = !S.\mathsf{end} \;\; \bowtie \;\; ?S.\mathsf{end} = \langle \Delta \diamond \Theta \rangle(s[2]) \upharpoonright 1$

(4) Let $\Delta = s[1] : !S.\,\mathsf{end}, s[2] : ?(1, S).!S'.\,\mathsf{end}$ and $\Theta = s[1] : \mathtt{void}, s[2] : \mathtt{void}$.
Then $\langle\Delta \diamond \Theta\rangle(s[1]) \upharpoonright 2 = !S.\mathsf{end} \;\bowtie\; ?S.!S'.\mathsf{end} = \langle\Delta \diamond \Theta\rangle(s[2]) \upharpoonright 1$
Note that here the mutual projections are dual although not symmetric.

(5) Let

$$\begin{aligned}\Delta = {} & s[1] : \mathsf{pause}.!S_1.\,\mathsf{end}, s[2] : \mathsf{pause}.?(1, S_1).?(3, S_3).!S_2.\,\mathsf{end}, \\ & s[3] : ?(1, S_1).\mathsf{pause}.!S_3.\,\mathsf{end}\end{aligned}$$

and $\Theta = s[1] : (S_1, \{2\}), s[2] : (S_2, \emptyset), s[3] : (S_3, \emptyset)$. Then:

$$\begin{aligned}\langle\Delta \diamond \Theta\rangle(s[1]) \upharpoonright 2 &= !S_1.\mathsf{pause}.!S_1.\mathsf{end} \\ \langle\Delta \diamond \Theta\rangle(s[2]) \upharpoonright 1 &= !S_2.\,\mathsf{pause}.?S_1.!S_2.\,\mathsf{end} \\ \langle\Delta \diamond \Theta\rangle(s[1]) \upharpoonright 3 &= !S_1.\mathsf{pause}.!S_1.\mathsf{end} \\ \langle\Delta \diamond \Theta\rangle(s[3]) \upharpoonright 1 &= !S_3.?S_1.\mathsf{pause}.!S_3.\,\mathsf{end} \\ \langle\Delta \diamond \Theta\rangle(s[2]) \upharpoonright 3 &= !S_2.\mathsf{pause}.?S_3.!S_2.\,\mathsf{end} \\ \langle\Delta \diamond \Theta\rangle(s[3]) \upharpoonright 2 &= !S_3.\,\mathsf{pause}.!S_3.\,\mathsf{end}\end{aligned}$$

△

We are now ready to define *coherence* of configuration environments. Besides the usual compatibility condition between the types of participants, our notion of coherence also requires output persistence:

Definition 10.32 (Coherence). A configuration environment $\langle\Delta\diamond\Theta\rangle$ is *coherent*, written $Co\,\langle\Delta \diamond \Theta\rangle$, if:

1. For any $s[\mathsf{p}] \in dom(\langle\Delta \diamond \Theta\rangle)$, if $s[\mathsf{p}] \notin vdom(\Theta)$ then $s[\mathsf{p}] : T \in \Delta^{live}$ implies $\mathsf{OG}(T)$;
2. For any p, q, if $s[\mathsf{p}] \in dom(\Delta) \cup vdom(\Theta)$ and $s[\mathsf{q}] \in dom(\Delta) \cup vdom(\Theta)$ then:

$$\langle\Delta \diamond \Theta\rangle(s[\mathsf{p}]) \upharpoonright \mathsf{q} \;\bowtie\; \langle\Delta \diamond \Theta\rangle(s[\mathsf{q}]) \upharpoonright \mathsf{p}$$

In Def. 10.32, Condition 2. is the standard duality requirement for any pair of present participants in $\langle\Delta\diamond\Theta\rangle$ (i.e., participants whose session channel in s has some type in Δ or a non-void message type in Θ): it essentially requires that p and q make dual communications offers to each other. Note that any configuration environment whose domain is a singleton $\{s[\mathsf{p}]\}$ trivially satisfies this condition. Condition 1. is specific to our calculus and is meant to ensure output persistence: it says that if a participant has not yet sent a message in the current instant, then it better do so before the next suspension point is reached. The fact that the type of p is saturated, with all suspension points represented by explicit pauses, plays an essential role here.

It is easy to see that the first three configuration environments in Ex. 10.31 are not coherent, because they violate Condition 1. (while they satisfy Condition 2.). On the other hand, the last two configuration environments are coherent.

We prove now that any two projections of the same global type have dual mutual projections:

Lemma 10.33. *If* $(\mathsf{S}_{\mathcal{R}}(G, \mathcal{P}) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}}(G, \mathcal{P}) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$, $\mathcal{P} \subseteq \mathcal{P}'$, $\mathsf{Part}(G) \subseteq \mathcal{R} \subseteq \mathcal{R}'$, $\mathcal{P} \subseteq \mathcal{R}$ *and* $\mathcal{P}' \subseteq \mathcal{R}'$ *then* $(\mathsf{S}_{\mathcal{R}'}(G, \mathcal{P}') \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}'}(G, \mathcal{P}') \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$.

Proof. By induction on the structure of G. There are two base cases and five inductive cases. For details see App. F.2. ☐

Proposition 10.34. *Let G be a global type and* $\mathsf{p} \neq \mathsf{q}$. *Then* $(G\lfloor_{\mathsf{p}}) \upharpoonright \mathsf{q} \bowtie (G\lfloor_{\mathsf{q}}) \upharpoonright \mathsf{p}$.

Proof. By induction on G. For details see App. F.2. ☐

We are now finally ready to introduce the typing rules for configurations. First of all, we mention that our type system admits the following weakening and contraction rules:

$$\lfloor\textsc{Weak}\rfloor\ \frac{\Gamma \vdash C \triangleright \langle \Delta \diamond \Theta \rangle}{\Gamma \vdash C \triangleright \langle \Delta, c : \mathsf{end} \diamond \Theta \rangle} \qquad \lfloor\textsc{Contr}\rfloor\ \frac{\Gamma \vdash C \triangleright \langle \Delta, c : \mathsf{end} \diamond \Theta \rangle}{\Gamma \vdash C \triangleright \langle \Delta \diamond \Theta \rangle}$$

These structural rules will allow us to add and remove fresh channels with empty behaviors in the session environment. They are included to allow simplifications on some of the typing rules in Fig. 10.17 and Fig. 10.18, which we briefly describe below:

- Rule $\lfloor$RVar$\rfloor$ types a process variable inside a configuration. It asks that environment Γ contains the required process variable and that the current memory is typable.
- Rule $\lfloor$Inact$\rfloor$ types a terminated configuration with any session environment containing only elements of the form $c : \mathsf{end}$ or $c : \langle \mathsf{end}, T \rangle^{ev}$ (condition $\Delta = \Delta^{end}$).
- Rules $\lfloor$MInit$\rfloor$ and $\lfloor$MAcc$\rfloor$ require the standard environment to associate a global type G with the service identifier a. Moreover, the last premise of Rule $\lfloor$MAcc$\rfloor$ guarantees that the type of p's channel in P is obtained as the p-th saturated projection of G.
- Rule $\lfloor$Conc$\rfloor$ types the parallel composition of two processes with a given memory and set of events, provided both components are typable with such memory and set of events and the obtained session environments have disjoint domains.
- Rule $\lfloor$CRes$\rfloor$ types a restricted configuration with the empty configuration environment provided the session environment of its body is coherent. This is the only typing rule that requires coherence.
- Rule $\lfloor$Emit$\rfloor$ has no effect on types, as event emission only affects the communication behavior via the watching construct, and the event set is not typed.
- Rule $\lfloor$Pause$\rfloor$ types a paused configuration, checking that the reconditioned configuration is well-typed, and requiring that the reconditioned session environment is output-granting. The latter condition will be required in all rules for suspended configurations.
- Rule $\lfloor$Rec$\rfloor$ types recursion, requiring each call to be isolated in its own time instant. To this end, the type declaration $X : \mathsf{pause}.T$ is added to Γ.

$$\lfloor\textsc{RVar}\rfloor\ \dfrac{\Gamma\vdash X:\mathsf{pause}.T \quad \Gamma\vdash M\triangleright\Theta}{\Gamma\vdash\langle X,M,E\rangle\triangleright\langle c:\mathsf{pause}.T\diamond\Theta\rangle} \qquad \lfloor\textsc{Inact}\rfloor\ \dfrac{\Delta=\Delta^{end} \quad \Gamma\vdash M\triangleright\Theta}{\Gamma\vdash\langle\mathbf{0},M,E\rangle\triangleright\langle\Delta\diamond\Theta\rangle}$$

$$\lfloor\textsc{MInit}\rfloor\ \dfrac{\Gamma\vdash a\triangleright G \quad |\mathsf{Part}(G)|=n}{\Gamma\vdash\langle\bar{a}[n],\emptyset,\emptyset\rangle\triangleright\langle\emptyset\diamond\emptyset\rangle} \qquad \lfloor\textsc{MAcc}\rfloor\ \dfrac{\Gamma\vdash a\triangleright G \quad \Gamma\vdash\langle P\{s[\mathsf{p}]/\alpha\},M^{\emptyset}_{s},\emptyset\rangle\triangleright\langle s[\mathsf{p}]:G\lfloor_{\mathsf{p}}\diamond\Theta^{\emptyset}_{s}\rangle}{\Gamma\vdash\langle a[\mathsf{p}](\alpha).P,\emptyset,\emptyset\rangle\triangleright\langle\emptyset\diamond\emptyset\rangle}$$

$$\lfloor\textsc{Conc}\rfloor\ \dfrac{\Gamma\vdash\langle P_i,M,E\rangle\triangleright\langle\Delta_i\diamond\Theta\rangle,\ i=1,2}{\Gamma\vdash\langle P_1\mid P_2,M,E\rangle\triangleright\langle\Delta_1,\Delta_2\diamond\Theta\rangle} \qquad \lfloor\textsc{CRes}\rfloor\ \dfrac{\Gamma\vdash C\triangleright\langle\Delta\diamond\Theta\rangle \quad Co\,\langle\Delta\diamond\Theta\rangle}{\Gamma\vdash(\nu s)C\triangleright\langle\emptyset\diamond\emptyset\rangle}$$

$$\lfloor\textsc{Emit}\rfloor\ \dfrac{\Gamma\vdash\langle P,M,E\cup ev\rangle\triangleright\langle\Delta\diamond\Theta\rangle}{\Gamma\vdash\langle\mathtt{emit}\,ev.\,P,M,E\rangle\triangleright\langle\Delta\diamond\Theta\rangle} \qquad \lfloor\textsc{Pause}\rfloor\ \dfrac{\Gamma\vdash M\triangleright\Theta \quad \Gamma\vdash\langle P,M^{\emptyset},\emptyset\rangle\triangleright\langle c:T\diamond\Theta^{\emptyset}\rangle \quad \mathsf{OG}(T)}{\Gamma\vdash\langle\mathsf{pause}.\,P,M,E\rangle\triangleright\langle c:\mathsf{pause}.T\diamond\Theta\rangle}$$

$$\lfloor\textsc{Rec}\rfloor\ \dfrac{\begin{array}{c}\Gamma,X:\mathsf{pause}.T\vdash\langle P,M^{\emptyset},\emptyset\rangle\triangleright\langle c:T\diamond\Theta^{\emptyset}\rangle\\ \Gamma,X:\mathsf{pause}.T\vdash\langle P,M,E\rangle\triangleright\langle c:T\diamond\Theta\rangle\end{array}}{\Gamma\vdash\langle(\mathsf{rec}\,X\,.\,P),M,E\rangle\triangleright\langle c:T\diamond\Theta\rangle}$$

$$\lfloor\textsc{SendFirst}\rfloor\ \dfrac{\Gamma\vdash e:S \quad \Gamma\vdash\langle P,M\cup s[\mathsf{p}]:(d_S,\emptyset),E\rangle\triangleright\langle s[\mathsf{p}]:T\diamond\Theta,s[\mathsf{p}]:(S,\emptyset)\rangle}{\Gamma\vdash\langle s[\mathsf{p}]!\langle e\rangle.P,M\cup s[\mathsf{p}]:\epsilon,E\rangle\triangleright\langle s[\mathsf{p}]:!S.T\diamond\Theta,s[\mathsf{p}]:\mathsf{void}\rangle}$$

$$\lfloor\textsc{SendMore}\rfloor\ \dfrac{\begin{array}{c}\Gamma\vdash M\triangleright\Theta \quad \Gamma\vdash e:S \quad \Gamma\vdash v:S' \quad \mathsf{OG}(T)\\ \Gamma\vdash\langle P,M^{\emptyset}\cup s[\mathsf{p}]:(d_S,\emptyset),\emptyset\rangle\triangleright\langle s[\mathsf{p}]:T\diamond\Theta^{\emptyset},s[\mathsf{p}]:(S,\emptyset)\rangle\end{array}}{\Gamma\vdash\langle s[\mathsf{p}]!\langle e\rangle.P,M\cup s[\mathsf{p}]:(v,\Pi),E\rangle\triangleright\langle s[\mathsf{p}]:\mathtt{tick}.!S.T\diamond\Theta,s[\mathsf{p}]:(S',\Pi)\rangle}$$

$$\lfloor\textsc{RcvFirst}\rfloor\ \dfrac{\Gamma,x:S\vdash\langle P,M\cup s[\mathsf{p}]:(v,\Pi\cup\mathsf{q}),E\rangle\triangleright\langle s[\mathsf{q}]:T\diamond\Theta,s[\mathsf{p}]:(S,\Pi\cup\mathsf{q})\rangle}{\Gamma\vdash\langle s[\mathsf{q}]?(\mathsf{p},x).P,M\cup s[\mathsf{p}]:(v,\Pi),E\rangle\triangleright\langle s[\mathsf{q}]:?(\mathsf{p},S).T\diamond\Theta,s[\mathsf{p}]:(S,\Pi)\rangle}$$

$$\lfloor\textsc{RcvMore}\rfloor\ \dfrac{\begin{array}{c}\Gamma\vdash M\triangleright\Theta \quad \mathsf{q}\in\Pi \quad \Gamma\vdash v:S' \quad \mathsf{OG}(T)\\ \Gamma,x:S\vdash\langle P,M^{\emptyset}\cup s[\mathsf{p}]:(d_S,\{\mathsf{q}\}),\emptyset\rangle\triangleright\langle s[\mathsf{q}]:T\diamond\Theta^{\emptyset},s[\mathsf{p}]:(S,\{\mathsf{q}\})\rangle\end{array}}{\Gamma\vdash\langle s[\mathsf{q}]?(\mathsf{p},x).P,M\cup s[\mathsf{p}]:(v,\Pi),E\rangle\triangleright\langle s[\mathsf{q}]:\mathtt{tick}.?(\mathsf{p},S).T\diamond\Theta,s[\mathsf{p}]:(S',\Pi)\rangle}$$

$$\lfloor\textsc{RcvNext}\rfloor\ \dfrac{\Gamma,x:S\vdash\langle P,M\cup s[\mathsf{p}]:(v,\{\mathsf{q}\}),\emptyset\rangle\triangleright\langle s[\mathsf{q}]:T\diamond\Theta,s[\mathsf{p}]:(S,\{\mathsf{q}\})\rangle}{\Gamma\vdash\langle s[\mathsf{q}]?(\mathsf{p},x).P,M\cup s[\mathsf{p}]:\epsilon,E\rangle\triangleright\langle s[\mathsf{q}]:?(\mathsf{p},S).T\diamond\Theta,s[\mathsf{p}]:\mathsf{void}\rangle}$$

Figure 10.17: Typing rules for configurations (Part 1).

$$\lfloor\textsc{Watch}\rfloor\ \frac{\begin{array}{c}\Gamma \vdash \langle P, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_P \diamond \Theta\rangle \\ T_P = \mathsf{end} \vee (\Gamma \vdash \langle Q, M^{\emptyset}, \emptyset\rangle \triangleright \langle s[\mathsf{p}] : T_Q \diamond \Theta^{\emptyset}\rangle \wedge \mathsf{OG}(T_Q))\end{array}}{\Gamma \vdash \langle \mathtt{watch}\, ev\ \mathtt{do}\ P\{Q\}, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_P \star_{ev} T_Q \diamond \Theta\rangle}$$

$$\lfloor\textsc{If}\rfloor\ \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle \quad \Gamma \vdash \langle Q, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle}{\Gamma \vdash \langle \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle}$$

Figure 10.18: Typing rules for configurations (Part 2).

- Rule ⌊SendFirst⌋ types the first broadcast by participant p in the current instant (the fact that it is the first is indicated by the presence of $s[\mathsf{p}] : \epsilon$ in the memory). If S is the sort of the broadcast value, the continuation P is required to be typable in the memory obtained by replacing $s[\mathsf{p}] : \epsilon$ with $s[\mathsf{p}] : (d_S, \emptyset)$, where d_S is the default value of sort S. This amounts to say that the continuation P must be typable in the memory just after the broadcast.

- Rule ⌊SendMore⌋ types the broadcast of the value of an expression e when the sender p has already sent some value v in the current instant (as witnessed by the presence of $s[\mathsf{p}] : (v, \Pi)$ in the memory). In this case, we insert a `tick` in front of the send type of $s[\mathsf{p}]$, to indicate that the new broadcast will take place at the next instant. The continuation P must be typable in the refreshed memory $M^{\emptyset}$ updated with the new message sent by p, and the reconditioned environment must be output-granting.

- Rule ⌊RcvFirst⌋ is analogous to Rule ⌊SendFirst⌋: it types an input by receiver q from sender p if a message sent by p is present in the memory, and this message has not been read yet by q (namely, q is not in its set of Readers Π). The continuation P needs to be typable in the updated memory.

- Rule ⌊RcvMore⌋ is analogous to Rule ⌊SendMore⌋: it types an input by receiver q from sender p in case q has already read a message v from p in the current instant. In this case, a `tick` is inserted in front of the receive type of channel $s[\mathsf{q}]$. Again, the reconditioned environment must be output-granting.

- Rule ⌊RcvNext⌋ is used to type an input by receiver q from sender p at the start of a new instant, when no participant has sent any message yet. This rule is very similar to rule ⌊RcvFirst⌋, except that the output buffer of the sender is empty, and therefore it is typed with `void`.

- Rule ⌊Watch⌋ types the watching construct by creating a pair of types $\langle T, T'\rangle^{ev}$ for each participant. Since the alternative process Q may only be launched at the start of a new instant, its session environment is required to be output-granting.

- Rule ⌊If⌋ requires, as usual in session type systems, that the two branches of the conditional be typed with the same session environment Δ.

10.4.4 Properties of the Type System

The semantic properties P1 and P2 defined previously both rely on subject reduction (SR). To prove SR, we need some preliminary definitions and results. The first property we prove ensures that the typing of configurations ensures the typing of memories:

Lemma 10.35. *If* $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ *then* $\Gamma \vdash M \triangleright \Theta$.

Proof. By induction on the height of the typing derivation $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. For details see App. F.3

□

Before proving subject reduction we now formalize a notion of reduction for configuration environments. This is needed because because the configuration environment evolves along execution.

Definition 10.36 (Reduction of Configuration Environments). Let $\Rightarrow$ be the reflexive and transitive relation on configuration environments generated by:

1. $\langle \Delta, s[\mathsf{p}] :!S.T \diamond \Theta, s[\mathsf{p}] : \mathtt{void}\rangle \;\Rightarrow\; \langle \Delta, s[\mathsf{p}] : T \diamond \Theta, s[\mathsf{p}] : (S, \emptyset)\rangle$
2. $\langle \Delta, s[\mathsf{q}] :?(\mathsf{p}, S).T \diamond \Theta, s[\mathsf{p}] : (S, \Pi)\rangle \;\Rightarrow\; \langle \Delta, s[\mathsf{q}] : T \diamond \Theta, s[\mathsf{p}] : (S, \Pi \cup \{\mathsf{q}\})\rangle$ with $\mathsf{q} \notin \Pi$
3. $\langle \Delta \diamond \Theta\rangle \;\Rightarrow\; \langle \Delta' \diamond \Theta'\rangle$ implies $\langle \Delta \star_{ev} \Delta'' \diamond \Theta\rangle \;\Rightarrow\; \langle \Delta' \star_{ev} \Delta'' \diamond \Theta'\rangle$
4. $\langle \Delta \diamond \Theta\rangle \;\Rightarrow\; \langle \Delta' \diamond \Theta'\rangle$ implies $\langle \Delta, \Delta'' \diamond \Theta\rangle \;\Rightarrow\; \langle \Delta', \Delta'' \diamond \Theta'\rangle$

Similarly, we introduce a tick reduction for the configuration environments of suspended configurations. This reduction relation is parameterized by the set of events E which is used to recondition the watch types.

We now argue that it is important for us to distinguish between implicit and explicit pauses in our typing, specifically, when aiming to prove a subject reduction result. In particular, consider

$$C = \langle \mathcal{E}[s[\mathsf{p}]?(\mathsf{q}, x).s[\mathsf{p}]?(\mathsf{r}, y).P], M, E\rangle$$

with $s[\mathsf{r}] : (v, \Pi) \in M$, $\mathsf{p} \in \Pi$ and $(\nu s)C\ddagger$. Using the typing rules in Fig. 10.17 and Fig. 10.18, we can see that:

$$\Gamma \vdash C \triangleright \langle \Delta, s[\mathsf{p}] :?(\mathsf{q}, S).\mathtt{tick}.?(\mathsf{r}, S).T \diamond \Theta\rangle$$

Since C is suspended, the configuration reduces, via a tick transition (cf. Fig. 10.9), to $(\nu s)D$ with

$$D = \langle \mathcal{E}[s[\mathsf{p}]?(\mathsf{q}, x).s[\mathsf{p}]?(\mathsf{r}, y).P], M^{\emptyset}, \emptyset\rangle.$$

whose typing is:

$$\Gamma \vdash D \triangleright \langle \Delta', s[\mathsf{p}] :?(\mathsf{q}, S).?(\mathsf{r}, S).T \diamond \emptyset\rangle$$

It is clear that in the above typing judgment, the `tick` has been deleted. Thus, it is necessary for us to distinguish between implicit and explicit pauses when defining a reduction for typing environments, as some of the implicit pauses may be removed.

Formally, this situation happens whenever a suspension occurs because of rule (in_s) (cf. Fig. 10.7). We address the deletion of $\mathtt{tick}$, using the $\mathtt{trm}(\cdot)$ function defined in Fig. 10.19. Intuitively, $\mathtt{trm}(\cdot)$ parses the local type and recalculates where to put implicit pauses.

Definition 10.37 (Tick Reduction of Configuration Environments). Let $\curvearrowright_E$ be the parameterized relation on configuration environments generated by:

1. (Pause) $\langle \mathsf{pause}(\Delta) \diamond \Theta\rangle \curvearrowright_\emptyset \langle \Delta \diamond \Theta^\emptyset\rangle$
2. (Tick) $\langle \mathsf{tick}(\Delta) \diamond \Theta\rangle \curvearrowright_\emptyset \langle \Delta \diamond \Theta^\emptyset\rangle$
3. (In) $\Theta = \Theta', s[\mathsf{p}] : \mathtt{void}$ implies for any E: $\langle s[\mathsf{q}] :?(\mathsf{p}, S).T \diamond \Theta\rangle \curvearrowright_E \langle s[\mathsf{q}] :?(\mathsf{p}, S).\mathtt{trm}(T)^0_{\{\mathsf{p}\}} \diamond \Theta^\emptyset\rangle$ where $\mathtt{trm}(\cdot)$ is as in Fig. 10.19
4. (Par) $\langle \Delta_i \diamond \Theta\rangle \curvearrowright_E \langle \Delta'_i \diamond \Theta'\rangle$, $i = 1, 2$ implies $\langle \Delta_1, \Delta_2 \diamond \Theta\rangle \curvearrowright_E \langle \Delta'_1, \Delta'_2 \diamond \Theta'\rangle$.
5. (Restr) $\langle \Delta \diamond \Theta\rangle \curvearrowright_E \langle \Delta' \diamond \Theta'\rangle$ implies $\langle \Delta \setminus s \diamond \Theta \setminus s\rangle \curvearrowright_E \langle \Delta' \setminus s \diamond \Theta' \setminus s\rangle$.
6. (Watch)
   $\langle \Delta_1 \diamond \Theta\rangle \curvearrowright_E \langle \Delta'_1 \diamond \Theta'\rangle$ implies
$$\begin{array}{lll} \langle \Delta_1 \star_{ev} \Delta_2 \diamond \Theta\rangle \curvearrowright_E \langle \Delta_2 \diamond \Theta'\rangle & & \text{if } ev \in E \\ \langle \Delta_1 \star_{ev} \Delta_2 \diamond \Theta\rangle \curvearrowright_E \langle \Delta'_1 \star_{ev} \Delta_2 \diamond \Theta'\rangle & & \text{if } ev \notin E \end{array}$$

We also let $\langle \Delta \diamond \Theta\rangle \curvearrowright \langle \Delta' \diamond \Theta'\rangle$ if there exists E such that $\langle \Delta \diamond \Theta\rangle \curvearrowright_E \langle \Delta' \diamond \Theta'\rangle$. The predicate $\langle \Delta \diamond \Theta\rangle \curvearrowright_E$ is defined by $\langle \Delta \diamond \Theta\rangle \curvearrowright_E$ if there exist Δ', Θ such that $\langle \Delta \diamond \Theta\rangle \curvearrowright_E \langle \Delta' \diamond \Theta'\rangle$, and the predicate $\langle \Delta \diamond \Theta\rangle \curvearrowright$ is defined similarly.

Note that $\curvearrowright$ is defined also on the environments of suspended subconfigurations, which cannot perform a tick transition.

Just like processes, local types and session environments should be reconditioned to properly reflect the reconditioning of processes during tick reductions:

Definition 10.38 (Reconditioning of Local Types and Session Environments). Let E be a set of events, the *reconditioning* of a local type H (under E), written $[T]_E$, is defined as:

$$[T]_E \overset{\mathsf{def}}{=} \begin{cases} T' & \text{if } T = \mathsf{pause}.T' \text{ or } T = \mathtt{tick}.T' \\ T & \text{if } T = \mathsf{end} \text{ or } T = \mathbf{t} \\ ?(\mathsf{p}, S).\mathtt{trm}(T')^0_{\{\mathsf{p}\}} & \text{if } T =?(\mathsf{p}, S).T' \\ T_2 & \text{if } T = \langle T_1, T_2\rangle^{ev} \text{ and } ev \in E \\ \langle [T_1]_E, T_2\rangle^{ev} & \text{if } T = \langle T_1, T_2\rangle^{ev} \text{ and } ev \notin E \end{cases}$$

where $\mathtt{trm}(\cdot)$ is defined in Fig. 10.19. Given Δ, its reconditioning under E is defined as $[\Delta]_E = \{c : [T]_E \bigm| c : T \in \Delta\}$.

The following proposition is easy to show, by induction on Def. 10.37:

Proposition 10.39. *If* $\langle \Delta \diamond \Theta\rangle \curvearrowright_E \langle \Delta' \diamond \Theta'\rangle$ *then* $\Delta' = [\Delta]_E$ *and* $\Theta' = \Theta^\emptyset$.

$$\mathtt{trm}(T)^0_\Pi \overset{\mathsf{def}}{=} \begin{cases} !S.T' & \text{if } T = \mathtt{tick}.!S.T' \\ !S.\mathtt{trm}(T')^1_\Pi & \text{if } T = !S.T' \\ \mathtt{tick}.?(\mathsf{p}, S).T' & \text{if } T = \mathtt{tick}.?(\mathsf{p}, S).T' \text{ and } \mathsf{p} \in \Pi \\ ?(\mathsf{p}, S).T' & \text{if } T = \mathtt{tick}.?(\mathsf{p}, S).T' \text{ and } \mathsf{p} \notin \Pi \\ ?(\mathsf{p}, S).\mathtt{trm}(T')^0_{\Pi\cup\{\mathsf{p}\}} & \text{if } T = ?(\mathsf{p}, S).T' \text{ and } \mathsf{p} \notin \Pi \\ \langle \mathtt{trm}(T_1)^0_\Pi, T_2\rangle^{ev} & \text{if } T = \langle T_1, T_2\rangle^{ev} \\ \mu\mathbf{t}.T' & \text{if } T = \mu\mathbf{t}.T' \\ \mathsf{pause}.T' & \text{if } T = \mathsf{pause}.T' \\ \mathsf{end} & \text{if } T = \mathsf{end} \end{cases}$$

$$\mathtt{trm}(T)^1_\Pi \overset{\mathsf{def}}{=} \begin{cases} \mathtt{tick}.!S.T' & \text{if } T = \mathtt{tick}.!S.T' \\ \mathtt{tick}.?(\mathsf{p}, S).T' & \text{if } T = \mathtt{tick}.?(\mathsf{p}, S).T' \text{ and } \mathsf{p} \in \Pi \\ ?(\mathsf{p}, S).T' & \text{if } T = \mathtt{tick}.?(\mathsf{p}, S).T' \text{ and } \mathsf{p} \notin \Pi \\ ?(\mathsf{p}, S).\mathtt{trm}(T')^1_{\Pi\cup\{\mathsf{p}\}} & \text{if } T = ?(\mathsf{p}, S).T' \text{ and } \mathsf{p} \notin \Pi \\ \langle \mathtt{trm}(T_1)^1_\Pi, T_2\rangle^{ev} & \text{if } T = \langle T_1, T_2\rangle^{ev} \\ \mu\mathbf{t}.T' & \text{if } T = \mu\mathbf{t}.T' \\ \mathsf{pause}.T' & \text{if } T = \mathsf{pause}.T' \\ \mathsf{end} & \text{if } T = \mathsf{end} \end{cases}$$

Figure 10.19: $\mathtt{trm}()$ function

Definition 10.40 (Mixed Reduction of Configuration Environments). Let $\rightsquigarrow$ stand for either $\Rightarrow$ or $\curvearrowright$, and let $\rightsquigarrow^*$ be its reflexive and transitive closure.

Hence $\langle \Delta \diamond \Theta\rangle \rightsquigarrow^* \langle \Delta' \diamond \Theta'\rangle$ when $\langle \Delta \diamond \Theta\rangle$ can reduce to $\langle \Delta' \diamond \Theta'\rangle$ via a mixed sequence of reductions. We prove now that types are preserved under $\equiv$ (i.e., *subject congruence*) and substitution. Notice that Rule $\lfloor$WEAK$\rfloor$ is necessary for subject congruence, as it allows to add ended sessions to Δ.

Lemma 10.41 (Subject Congruence). *If* $\Gamma \vdash C \triangleright \langle \Delta \diamond \Theta\rangle$ *and* $C \equiv C'$ *then* $\Gamma \vdash C' \triangleright \langle \Delta \diamond \Theta\rangle$.

Proof. By induction on the definition of structural congruence (cf. Fig. 10.5). For details see App. F.3. □

Lemma 10.42 (Substitution Lemma). *If* $\Gamma, x : S \vdash C \triangleright \langle \Delta \diamond \Theta\rangle$ *and* $\Gamma \vdash v : S$ *then* $\Gamma \vdash C\{^v/_x\} \triangleright \langle \Delta \diamond \Theta\rangle$.

Proof. By induction on the height of the type derivation. The base cases are rules $\lfloor$INACT$\rfloor$ and $\lfloor$MINIT$\rfloor$. The thesis is easily derived observing that x does not occur free neither in **0** nor in the initiator. See App. F.3 for details on the inductive cases. □

Lemma 10.43 (Reduction Lemma). *Let* $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ *and* $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$ *via some reduction rule different from* [Cont] *and* [Struct]. *Then*

$$\langle \Delta \diamond \Theta\rangle \;\Rightarrow\; \langle \Delta' \diamond \Theta'\rangle$$

and $\Gamma \vdash \langle P', M', E' \rangle \triangleright \langle \Delta' \diamond \Theta' \rangle$. *Moreover, if* $\langle \Delta, \Delta_0 \diamond \Theta, \Theta_0 \rangle$ *is coherent, then* $\langle \Delta', \Delta_0 \diamond \Theta', \Theta_0 \rangle$ *is coherent.*

Proof. By induction on length of the reduction $\langle P, M, E \rangle \longrightarrow \langle P', M', E' \rangle$, with a case analysis on the last applied rule. For details see App. F.3. $\square$

To prove subject reduction, we first need to prove that tick reduction for configuration environments preserves duality and that suspension preserves typing. This is useful for ensuring that the semantics given to environment configurations works correctly.

Lemma 10.44 (Tick Reduction of Configuration Environments Preserves Duality). *If* $\langle \Delta \diamond \Theta \rangle \curvearrowright_E \langle \Delta' \diamond \Theta' \rangle$ *then* $\Delta' = [\Delta]_E$, $dom(\Delta') \subseteq dom(\Delta)$ *and* $\Theta' = \Theta^{\emptyset}$. *Moreover, if* $\langle \Delta \diamond \Theta \rangle$ *satisfies duality then also* $\langle \Delta' \diamond \Theta' \rangle$ *satisfies duality and for any* $s[\mathsf{p}], s[\mathsf{q}] \in dom(\Delta')$, *if* $s[\mathsf{p}] : T_{\mathsf{p}} \in \Delta$ *and* $s[\mathsf{q}] : T_{\mathsf{q}} \in \Delta$, *then* $\langle \Delta \diamond \Theta \rangle(s[\mathsf{p}]) \upharpoonright \mathsf{q} \bowtie \langle \Delta \diamond \Theta \rangle(s[\mathsf{q}]) \upharpoonright \mathsf{p}$ *if and only if* $[T_{\mathsf{p}}]_E \upharpoonright \mathsf{q} \bowtie [T_{\mathsf{q}}]_E \upharpoonright \mathsf{p}$.

Proof. By induction on the definition of $\curvearrowright$. There is only one basic case, corresponding to Rule (Pause). For details see App. F.3. $\square$

Lemma 10.45 (Suspension Lemma). *Let* $\langle P, M, E \rangle \ddagger$ *and* $\Gamma \vdash \langle P, M, E \rangle \triangleright \langle \Delta \diamond \Theta \rangle$. *Then we have that* $\langle \Delta \diamond \Theta \rangle \curvearrowright \langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$ *and* $\Gamma \vdash \langle [P]_E, M^{\emptyset}, \emptyset \rangle \triangleright \langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$. *Moreover, if* $\langle \Delta \diamond \Theta \rangle$ *is coherent then* $\langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$ *is coherent.*

Proof. By induction on the definition of $\langle P, M, E \rangle \ddagger$. The basic cases correspond to the suspension rules $(pause)$, (out_s), (in_s) and (in_s^2), and the inductive cases to rules (par_s), $(watch_s)$ and (rec_s). For details see App. F.3. $\square$

We can now finally prove the subject reduction theorem. As most of our previous results, this theorem deals only with reachable configurations. Let $\rightsquigarrow^+$ denote the transitive closure of the relation $\rightsquigarrow$.

Theorem 10.46 (Subject Reduction). *Let* C *be a reachable configuration and* $C \rightsquigarrow^+ C'$. *If* $\Gamma \vdash C \triangleright \langle \Delta \diamond \Theta \rangle$ *then* $\Gamma \vdash C' \triangleright \langle \Delta' \diamond \Theta' \rangle$ *and* $\langle \Delta \diamond \Theta \rangle \Rrightarrow^* \langle \Delta' \diamond \Theta' \rangle$ *for some* Δ', Θ'. *Moreover if* $\langle \Delta \diamond \Theta \rangle$ *is coherent then* $\langle \Delta' \diamond \Theta' \rangle$ *is coherent.*

Proof. By induction on the length n of the reduction sequence $\rightsquigarrow^+$. For detail see App. F.3. $\square$

10.5 Time-Related Properties

In this section we prove that our type system enforces some desirable "real-time" properties. More precisely, we prove that any configuration that is reachable from an initial configuration and complies with a global type G (the notion of G-compliance will be made precise below) satisfies the following properties:

P1. *Output persistence*: Every participant broadcasts exactly once during every instant;

P2. *Input timeliness*: Every unguarded input is matched by an output during the current instant, if not preceded by another input with equal source and target, or during the next instant, if not preempted.

To formalise Property P1, we first define the auxiliary property of *output readiness* for reachable configurations of the form $C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$, which states that all participants occurring in P perform a broadcast during the current instant. Intuitively, a reachable configuration $C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$ represents the state of a running session at the start of an instant (or more precisely, before any message exchange or event emission within an instant).

Definition 10.47 (Output Readiness). A reachable configuration $C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$ is *output-ready* if whenever $C \Downarrow (\nu s)\langle P', M', E'\rangle$, then $s[\mathsf{p}] \in vdom(M')$ for every $s[\mathsf{p}] \in nm(P)$.

Note that only the participants p whose behavior is nonterminated at the beginning of the instant (condition $s[\mathsf{p}] \in nm(P)$) are required to perform an output before the end of the instant (condition $s[\mathsf{p}] \in vdom(M')$). Then, output persistence for such configurations essentially amounts to requiring output readiness at every instant. For an initial configuration $\langle Q, \emptyset, \emptyset\rangle$, output persistence amounts to requiring output readiness for any configuration $(\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$ such that $\langle Q, \emptyset, \emptyset\rangle \rightsquigarrow^{+} (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$.

Definition 10.48 (Output Persistence). A reachable configuration C is *output-persistent if whenever* $C \rightsquigarrow^{*} (\nu s)\langle P, M, E\rangle \Downarrow (\nu s)\langle P', M', E'\rangle$, then $s[\mathsf{p}] \in vdom(M')$ for every $s[\mathsf{p}] \in nm(P)$.

Again, only nonterminated participants p at the beginning of the last instant (condition $s[\mathsf{p}] \in nm(P)$) are required to perform an output. For instance, our auction protocol (cf. § 10.2) satisfies output persistence although the terminated participant *Forwarder* does not output anything during the last instant.

We now formalize Property P2, which again rests on an auxiliary property defined only on reachable configurations of the form $C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$, called input readiness.

Definition 10.49 (Input Readiness). A reachable configuration $C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$ is *input ready* if whenever $C \Downarrow (\nu s)\langle \mathcal{E}[s[\mathsf{q}]?(\mathsf{p}, x).P'], M', E'\rangle$, then $s[\mathsf{p}] \in vdom(M')$.

Observe that $s[\mathsf{p}] \in vdom(M')$ implies $s[\mathsf{p}] : (v, \Pi \cup \{q\}) \in M'$ for some v, Π. Namely, q must have read a message from p in the current instant, otherwise the final configuration would not be suspended.

Input timeliness amounts to input readiness at every instant. For an initial configuration C, input timeliness amounts to requiring input readiness for the configurations derivable from C.

Definition 10.50 (Input Timeliness). A reachable configuration C is considered *input timely* if whenever $C \rightsquigarrow^{*} (\nu s)\langle \mathcal{E}[s[\mathsf{q}]?(\mathsf{p}, x).P], M, E\rangle \Downarrow (\nu s)\langle P', M', E'\rangle$ then $s[\mathsf{p}] \in vdom(M')$.

Example 10.51.

$C = \langle (\nu s)(s[1]?(2, x).s[1]?(2, y).\mathbf{0} \mid s[2]!\langle v\rangle.s[2]!\langle v\rangle.\mathbf{0}), M_s, \emptyset\rangle$ is input timely

$C' = \langle (\nu s)(s[1]?(2, x).\mathbf{0} \mid \mathsf{pause}.\ s[2]!\langle v\rangle.\mathbf{0}), M_s, \emptyset\rangle$ is not satisfy input timely

Indeed, in C the expectations of the two participants are dual and "well-timed": their first communication takes place in the first instant and their second communication takes place in the second instant.

In C', on the other hand, participant 1 is ready to receive a message from participant 2 in the first instant, and entitled to do so because she hasn't read previously from participant 2, but there is no available message from participant 2 in the first instant. In fact, C is G-compliant whereas C' is not even typable. $\triangle$

Observe that if $\Gamma \vdash C = \langle P, M^{\emptyset}, \emptyset\rangle \triangleright \langle \Delta \diamond \Theta\rangle$, the coherence of the configuration environment $\langle \Delta \diamond \Theta\rangle$ is not sufficient to ensure input timeliness of $(\nu\tilde{s})C$, because of the possibility of circular dependencies. Indeed, duality is a binary property which does not exclude n-ary circular dependencies such as that of the following process, which is reminiscent of the dining philosophers deadlock:

$$R = s[1]?(2,x).s[1]!\langle v_1\rangle.\mathbf{0} \mid s[2]?(3,y).s[2]!\langle v_2\rangle.\mathbf{0} \mid s[3]?(1,z).s[3]!\langle v_3\rangle.\mathbf{0} \qquad (10.2)$$

Note that the configuration $(\nu s)C = \nu s\langle R, M^{\emptyset}_{s}, \emptyset\rangle$ is not deadlocked but *livelocked* in MRS. Indeed, as shown in § 10.3.3, MRS is deadlock-free: thanks to our semantic rule (in_s) (cf. Fig. 10.7), all potential deadlocks due to missing messages are turned into livelocks. In fact, it is easy to see that $(\nu s)C = \nu s\langle R, M^{\emptyset}_{s}, \emptyset\rangle$ does not satisfy input timeliness, because all participants are waiting for each other. Since the configuration is reconditioned to itself at the following instant, it gives rise to a livelock.

In order to obtain input timeliness (and thus livelock-freedom), we need to require that the types of all participants be *projections of the same saturated* global type in the initial configuration of the session. We first define what it means for a configuration to implement a session specified by a saturated global type G. From now on, we will consider only saturated global types. As a consequence we will use the standard projection $\upharpoonright$ defined in Fig. 10.12 and not the saturated projection $\lfloor$.

Definition 10.52 (G-Compliant Configuration). Let G be a saturated global type. A reachable configuration $C = (\nu s)\langle P, M, E\rangle$ is *G-compliant* if there exist Δ, Θ such that $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ and $\langle \Delta \diamond \Theta\rangle(s[\mathsf{p}]) = G \upharpoonright \mathsf{p}$ for every $\mathsf{p} \in \mathsf{Part}(G)$.

The key property for proving both output persistence and input timeliness is the absence of circular dependencies in G-compliant configurations. The proof makes use of two *flattening* functions $\Phi(G)$ and $\Phi(T)$ on saturated global and local types. These functions extract from a type the sequence of I/O communications occurring in the first instant, forgetting about recursion and selecting the principal behavior in watch types.

Definition 10.53 (Flattening). The *flattening functions* $\Phi(G)$ and $\Phi(T)$ are defined in Fig. 10.20.

Since $\Phi(G)$ is again a global type, it can be projected on participants. It is easy to see that:

Lemma 10.54. $\Phi(G) \upharpoonright \mathsf{p} = \Phi(G \upharpoonright \mathsf{p})$

Proof. By induction on G. $\square$

We can now prove that G-compliant configurations are free of circular dependencies such as that exhibited by the three-philosopher process (cf. Ex. 10.2).

$$\begin{aligned}
&\Phi(\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.G) \overset{\mathsf{def}}{=} \mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.\Phi(G)\\
&\Phi(\mathsf{pause}.G) \overset{\mathsf{def}}{=} \Phi(\mathtt{tick}.G) \overset{\mathsf{def}}{=} \Phi(\mathbf{t}) \overset{\mathsf{def}}{=} \Phi(\mathsf{end}) \overset{\mathsf{def}}{=} \mathsf{end}\\
&\Phi(\mathtt{watch}\ ev\ \mathsf{do}\ G\ \mathtt{else}\ G') \overset{\mathsf{def}}{=} \Phi(\mu\mathbf{t}.G) \overset{\mathsf{def}}{=} \Phi(G)\\
&\Phi(!S.T) \overset{\mathsf{def}}{=} !S.\Phi(T)\\
&\Phi(?(\mathsf{p},S).T) \overset{\mathsf{def}}{=} ?(\mathsf{p},S).\Phi(T)\\
&\Phi(\mathsf{pause}.T) \overset{\mathsf{def}}{=} \Phi(\mathtt{tick}.T) \overset{\mathsf{def}}{=} \Phi(\mathbf{t}) \overset{\mathsf{def}}{=} \Phi(\mathsf{end}) \overset{\mathsf{def}}{=} \mathsf{end}\\
&\Phi(\langle T,T'\rangle^{ev}) \overset{\mathsf{def}}{=} \Phi(\mu\mathbf{t}.T) \overset{\mathsf{def}}{=} T
\end{aligned}$$

Figure 10.20: Flattening of saturated global and local types.

Lemma 10.55 (Absence of Circular Dependencies). *Let $C = (\nu s)\langle P, M, E\rangle$ be a G-compliant configuration, and let $\{\mathsf{p}_1,\dots,\mathsf{p}_n\} \subseteq \mathsf{Part}(G)$, with $n \geq 2$. If $s[\mathsf{p}_i] \notin vdom(M)$ for all $i \in \{1,\dots,n\}$, then there cannot exist $\mathcal{E}_1,\dots,\mathcal{E}_n$ such that:*

$$\begin{array}{c}
C \equiv \langle \mathcal{E}_1[s[\mathsf{p}_1]?(\mathsf{p}_2,x_1).P_1], M^{\emptyset}, \emptyset\rangle\\
C \equiv \langle \mathcal{E}_2[s[\mathsf{p}_2]?(\mathsf{p}_3,x_2).P_2], M^{\emptyset}, \emptyset\rangle\\
\vdots\\
C \equiv \langle \mathcal{E}_n[s[\mathsf{p}_n]?(\mathsf{p}_1,x_n).P_n], M^{\emptyset}, \emptyset\rangle
\end{array} \tag{10.3}$$

Proof. By contradiction. Assume the situation described in (10.3). By hypothesis C is G-compliant, hence $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ and $\langle \Delta \diamond \Theta\rangle(s[\mathsf{p}_i]) = G \upharpoonright \mathsf{p}_i$ for every p_i. From $s[\mathsf{p}_i] \notin vdom(M)$ it follows that $s[\mathsf{p}_i] \notin vdom(\Theta)$, thus $\langle \Delta \diamond \Theta\rangle(s[\mathsf{p}_i] = \Delta(s[\mathsf{p}_i])$. Therefore for all $i \in \{1,\dots,n\}$ we have:

$$\Delta(s[\mathsf{p}_i]) = G \upharpoonright \mathsf{p}_i \tag{10.4}$$

Observe now that $C \equiv \langle \mathcal{E}_i[s[\mathsf{p}_i]?(\mathsf{p}_{(i+1)\ mod\ n}, x_i).P_i], M, E\rangle$ implies that for some T_i

$$\Phi(\Delta(s[\mathsf{p}_i])) = ?(\mathsf{p}_{(i+1)\ mod\ n}, S_i).\Phi(T_i) \tag{10.5}$$

Pick now an arbitrary $k \in \{1,\dots,n\}$. It follows that

$$\begin{aligned}
\Phi(G) \upharpoonright \mathsf{p}_k &= \Phi(G \upharpoonright \mathsf{p}_k) && \text{(by Lem. 10.54)}\\
&= \Phi(\Delta(s[\mathsf{p}_k])) && \text{(by (10.4))}\\
&= ?(\mathsf{p}_{(k+1)\ mod\ n}, S_k).\Phi(T_i) && \text{(by (10.5))}
\end{aligned}$$

This means that $\Phi(G)$ is of the form:

$$\Phi(G) = \sigma_k.\mathsf{p}_{(k+1)\ mod\ n}\!\uparrow\!\langle S_k, \Pi_k\rangle.\Phi(G_k) \tag{10.6}$$

where σ_k is a possibly empty sequence of communications not involving p_k and not involving $\mathsf{p}_{(k+1)\ mod\ n}$ as a sender, and $\mathsf{p}_k \in \Pi_k$. This implies:

$$\Phi(G) \upharpoonright \mathsf{p}_{(k+1)\ mod\ n} = \sigma'_k.!S_k.T$$

where σ'_k is the projection of σ_k on $\mathsf{p}_{(k+1)\ mod\ n}$ Now there are two possible cases:

(1) σ'_k is the empty sequence. Then

$$\begin{aligned}\Phi(G) \restriction \mathsf{p}_{(k+1)\ mod\ n} &= !S_k.T \\ &\neq ?(\mathsf{p}_{(k+2)\ mod\ n}, S_{k+1}).T' && \text{(by (10.5))} \\ &= \Phi(\Delta(s[\mathsf{p}_{(k+1)\ mod\ n}])) && \text{(by Lem. 10.4)} \\ &= \Phi(G \restriction \mathsf{p}_{(k+1)\ mod\ n}) && \text{(by Lem. 10.54)}\end{aligned}$$

This inequality contradicts Lem. 10.54.

(2) σ'_k starts with $?(\mathsf{p}_{(k+2)\ mod\ n}, S_{k+1}).T'$. Then we iterate the reasoning until we reach the p_k we started with (which we are sure to reach since the number of participants is finite), and at this point we have a contradiction since by hypothesis σ_k does not contain p_k in (10.6).

□

Corollary 10.56. *Let $C = (\nu s)\langle P, M, E\rangle$ be a G-compliant configuration such that $C\ddagger$. Then $P = \mathcal{E}[s[\mathsf{q}]?(\mathsf{p}, x).P']$ implies $s[\mathsf{p}] \in vdom(M)$.*

Proof. By contradiction. Suppose that $s[\mathsf{p}] \notin vdom(M)$. Since C is G-compliant, we know that $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle\Delta \diamond \Theta\rangle$ and $\langle\Delta \diamond \Theta\rangle(s[\mathsf{r}]) = G \restriction \mathsf{r}$ for all $\mathsf{r} \in \mathsf{Part}(G)$. From $s[\mathsf{p}] \notin vdom(M)$ it follows that $s[\mathsf{p}] \notin vdom(\Theta)$, hence $\langle\Delta \diamond \Theta\rangle(s[\mathsf{p}] = \Delta(s[\mathsf{p}])$. This means that $\Delta(s[\mathsf{p}]) = G \restriction \mathsf{p}$. On the other hand, given the shape of P, we know that it must be $\Phi(G \restriction \mathsf{q}) = \Phi(\langle\Delta \diamond \Theta\rangle(s[\mathsf{q}])) = \{!S_\mathsf{q}.\}?(\mathsf{p}, S).\Phi(T)$ for some S_q, S and T, where the notation $\{!S_\mathsf{q}.\}$ means that the initial output is possibly missing (in case $s[\mathsf{q}] \notin vdom(M)$). By Lem. 10.54 $\Phi(G \restriction \mathsf{q}) = \Phi(G) \restriction \mathsf{q}$. This means that $\Phi(G)$ is of the form:

$$\Phi(G) = \sigma.\mathsf{p}\uparrow\langle S, \Pi\rangle.\Phi(G') \tag{10.7}$$

where σ is a possibly empty sequence of communications not involving q as a receiver nor p as a sender, and $\mathsf{q} \in \Pi$. Now, consider the projection $\Phi(G) \restriction \mathsf{p}$ of $\Phi(G)$ on p. Note that the projection of σ on p cannot be empty, because in this case we would have $\Phi(G) \restriction \mathsf{p} = !S.T_\mathsf{p}$ for some T_p, contradicting the fact that C is suspended. Then σ must consist of inputs by p. This means that $P = \mathcal{E}[s[\mathsf{p}]?(\mathsf{r}, x).R]$. Since C is suspended, we know that either $s[\mathsf{r}] \notin vdom(M)$ or $s[\mathsf{r}] : (v, \Pi \cup \{p\}) \in M$ for some v, Π. In the latter case, by Rule $\lfloor$RcvMore$\rfloor$ we would have $\Delta(s[\mathsf{p}]) = \mathsf{pause}.?(\mathsf{r}, S').\Phi(T') = G \restriction \mathsf{p}$ for some S' and T'. Then $\Phi(G \restriction \mathsf{p}) = \Phi(G) \restriction \mathsf{p}$ would be end, and thus so would $\Phi(G)$, contradicting equation (10.7), where σ is supposed to consist only of communications. Therefore it must be $s[\mathsf{r}] \notin vdom(M)$. But now we have for participant r the same hypotheses that we had for p in the beginning. Hence we can iterate the reasoning and since the number of participants is finite this leads us to the circular situation (10.3), which by Lemma 10.55 is impossible. □

We will prove now that the above lemma entails input readiness, and as a consequence, also input timeliness. We first show that compliance with a global type is preserved along execution.

Lemma 10.57. *Let C be a G-compliant configuration and $\langle\Delta \diamond \Theta\rangle$ be the corresponding configuration environment. If $C \longrightarrow C'$ (resp., $C \hookrightarrow_E C'$), then there exists G' with corresponding configuration environment $\langle\Delta' \diamond \Theta'\rangle$ such that C' is G'-compliant and $\langle\Delta \diamond \Theta\rangle \Rightarrow \langle\Delta' \diamond \Theta'\rangle$ (resp., $\langle\Delta \diamond \Theta\rangle \curvearrowright_E \langle\Delta' \diamond \Theta'\rangle$).*

Proof. By induction on the inference of $\longrightarrow$ (resp., $\hookrightarrow_E$). □

Lemma 10.58 (**G-Compliance Implies Input Readiness**)**.** *Every G-compliant configuration*

$$C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$$

satisfies input readiness.

Proof. Let $C \Downarrow (\nu s)\langle P', M', E'\rangle = C'$, where $P' = \mathcal{E}[s[\mathsf{q}]?(\mathsf{p}, x).P'']$. By Lemma 10.57, C' is G'-compliant for some G'. Then $s[\mathsf{p}] \in vdom(M')$ by Corollary 10.56. □

Theorem 10.59 (**G-Compliance Implies Input Timeliness**)**.** *Every G-compliant configuration*

$$C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$$

satisfies input timeliness.

Proof. Suppose $C \leadsto^* (\nu s)\langle \mathcal{E}[s[\mathsf{q}]?(\mathsf{p}, x).P'], M, E\rangle = C'$ and $C'\ddagger$. The proof is by induction on the number n of $\hookrightarrow_E$ steps in the execution $\leadsto^*$. If $n = 0$ then we are in the case $C \Downarrow C'$ and we have the result by Lemma 10.58.

If $n > 0$, the result follows again by Lemma 10.58, observing that by Lemma 10.57 G-compliance is preserved by execution, and that every $\hookrightarrow_E$ step gives rise again to a configuration of the form $(\nu s)\langle Q, M^{\emptyset}, \emptyset\rangle$. □

We proceed now to prove output persistence.

Lemma 10.60 (**G-Compliance Implies Output Readiness**)**.** *Let $C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle$ be a G-compliant configuration. If $C \Downarrow (\nu s)\langle P', M', E\rangle$, then $s[\mathsf{p}] \in vdom(M')$ for every $s[\mathsf{p}] \in nm(P)$.*

Proof. Let $C' = (\nu s)\langle P', M', E'\rangle$. Since C is G-compliant, by Lemma 10.57 C' is G'-compliant for some G'. Therefore there exist Δ, Θ such that $\Gamma \vdash \langle P', M', E'\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ and $\langle \Delta \diamond \Theta\rangle(s[\mathsf{p}]) = G' \upharpoonright \mathsf{p}$ for every $\mathsf{p} \in \mathsf{Part}(G')$. Now, the statement $(\nu s)\langle P', M', E'\rangle\ddagger$ is deduced by Rule $(restr_s)$ from $\langle P', M', E'\rangle\ddagger$.

The proof then proceeds by contradiction. Suppose there exists a $s[\mathsf{p}] \in nm(P)$ such that $s[\mathsf{p}] \notin vdom(M')$. We have $\langle \Delta \diamond \Theta\rangle(s[\mathsf{p}]) = (\Delta(s[\mathsf{p}])) = T$ and by coherence $\mathsf{OG}(T)$. This means that

$$\Phi(T) = \sigma.!S.T'$$

for some σ, S, T' and there are three possible cases for σ:

(1) σ is empty, so T is derived using rule $\lfloor\textsc{Sendfirst}\rfloor$:

$$\Gamma \vdash \langle s[\mathsf{p}]!\langle e\rangle.R, M'' \cup \{s[\mathsf{p}] : \epsilon\}, E\rangle \triangleright \langle \Delta, s[\mathsf{p}] :!S.T \diamond \Theta,\ s[\mathsf{p}] : \mathtt{void}\rangle$$

with $P' \equiv \mathcal{E}[s[\mathsf{p}]!\langle e\rangle.R]$ and $M' = M'' \cup \{s[\mathsf{p}] : \epsilon\}$ but this is a contradiction since $\langle s[\mathsf{p}]!\langle e\rangle.R, M'' \cup \{s[\mathsf{p}] : \epsilon\}, E\rangle$ cannot be suspended.

(2) σ is a non empty sequence of inputs and T is derived using rule $\lfloor\textsc{Rcvfirst}\rfloor$:

$$\Gamma \vdash \langle s[\mathsf{p}]?(\mathsf{q}, x).R, M'' \cup \{s[\mathsf{q}] : (v, \Pi)\}, E\rangle \triangleright \langle \Delta, s[\mathsf{p}] :?(\mathsf{q}, S).T \diamond \Theta,\ s[\mathsf{q}] : (S, \Pi)\rangle$$

with $P' \equiv \mathcal{E}[s[\mathsf{p}]?(\mathsf{q}, x).R]$ and $M' = M'' \cup \{s[\mathsf{q}] : (v, \Pi)\}$ with $\mathsf{p} \notin \Pi$, but this is a contradiction since $\langle s[\mathsf{p}]?(\mathsf{q}, x).R, M'' \cup \{s[\mathsf{q}] : (v, \Pi)\}, E\rangle$ cannot be suspended.

(3) σ is a non empty sequence of inputs and T is derived using rule $\lfloor$RCVNEXT$\rfloor$:

$$\Gamma \vdash \langle s[\mathsf{p}]?(\mathsf{q}, x).R, M'' \cup \{s[\mathsf{q}] : \epsilon\}, E\rangle \triangleright \langle \Delta, s[\mathsf{p}] :?(\mathsf{q}, S).T \diamond \Theta,\ s[\mathsf{q}] : \mathtt{void}\rangle$$

with $P' \equiv \mathcal{E}[s[\mathsf{p}]?(\mathsf{q}, x).R]$ and $M' = M'' \cup \{s[\mathsf{q}] : \epsilon\}$. This is a contradiction since by Cor. 10.56 we should have $s[\mathsf{q}] \in vdom(M')$.

Hence in all cases we reach a contradiction, concluding the proof. □

Theorem 10.61 (G-Compliance Implies Output persistence). *Let C be a G-compliant configuration. If $C \rightsquigarrow^* (\nu s)\langle P_0, M_0, E_0\rangle \Downarrow (\nu s)\langle P', M', E'\rangle$, then $s[\mathsf{p}] \in vdom(M')$ for every $s[\mathsf{p}] \in nm(P_0)$.*

Proof. Let $C = (\nu s)\langle P, M^{\emptyset}, \emptyset\rangle \rightsquigarrow^* (\nu s)\langle P_0, M_0, E_0\rangle \Downarrow (\nu s)\langle P', M', E'\rangle$. We want to show that $s[\mathsf{p}] \in vdom(M')$ for every $s[\mathsf{p}] \in nm(P_0)$. The proof is by induction on the number n of $\hookrightarrow_E$ steps in the execution sequence $\rightsquigarrow^*$. If $n = 0$ then we are in the case $C \Downarrow C'$ and we have the result by Lem. 10.60. If $n > 0$, the result follows again by Lem. 10.60, since by Lem. 10.57 G-compliance is preserved by execution and every $\hookrightarrow_E$ step gives rise again to a configuration of the form $(\nu s)\langle Q, M^{\emptyset}, \emptyset\rangle$. □

10.6 Timed Patterns Revisited: MRS

In this section we discuss how global types in MRS (cf. Fig. 10.10) can be used to enforce the timed patterns presented in § 1.6. Contrary to the translations in Ch. 4 and Ch. 7, global types in MRS can be used to enforce the timing constraints, rather than represent processes that execute the pattern's behavior.

Remark 10.62. Notice that we do not write saturated global types in the examples below (cf. Fig. 10.11). This is not a problem because the timing patterns presented in § 1.6 concern communications between two (or more) participants, rather than the outputs without receptors added by saturation.

Example 10.63 (Request-Response Timeout Pattern). Let us recall the request-response timeout pattern from § 1.6 as it is observed from the server side:

(a) *Server side:* After receiving a message REQ from A, B must send the acknowledgment ACK within t time units.

To represent this pattern in MRS we can use the developments presented in § 7.4.2: using preemption constructs to simulate the timeout signal. For the global types, we assume two participants: C, which represents the client and S, which represents the server. The global type below captures the pattern:

$$G = \mathsf{C}\!\uparrow\!\langle \mathtt{REQ}, \{\mathsf{S}\}\rangle.\mathtt{watch}\ t\ \mathtt{do}\ \mathsf{S}\!\uparrow\!\langle \mathtt{ACK}, \{\mathsf{C}\}\rangle.\mathtt{end}\ \mathtt{else}\ G_f$$

Above, event t represents the timeout signal which activates the alternative behavior G_f if s does not send the message ACK within the appropriate time-frame. The alternative behavior G_f can be, for example, a recovery protocol to deal with the failure.

To understand the relation between global type G and the MRS calculus, we shall show a possible implementation for the server. First, consider the saturated version of G (cf. Fig. 10.11). Notice that we assume that $\mathsf{Part}(G) = \{\mathsf{C}, \mathsf{S}\}$:

$$\mathsf{S}_{\mathsf{Part(G)}}(G, \emptyset) = \mathsf{C}{\uparrow}\langle \mathsf{REQ}, \{\mathsf{S}\}\rangle.\mathtt{watch}\ t\ \mathtt{do}\ \mathsf{S}{\uparrow}\langle \mathsf{ACK}, \{\mathsf{C}\}\rangle.\mathtt{end}\ \mathtt{else}\ \mathsf{S}_{\mathsf{Part(G_f)}}(G_f, \emptyset) = G'$$

Moreover, we consider the projection $G' \upharpoonright \mathsf{S}$ to obtain the local type corresponding to the server (cf. Fig. 10.12):

$$G' \upharpoonright \mathsf{S} = ?(\mathsf{C}, \mathsf{REQ}).\langle !\mathsf{ACK}.\mathsf{end}, \mathsf{S}_{\mathsf{Part(G_f)}}(G_f, \emptyset) \upharpoonright \mathsf{S}\rangle^t = T_\mathsf{S}$$

It can then be shown that T_S types the following process (assuming that the session between S and C is called s and ack is of type ACK):

$$P_\mathsf{S} = s[\mathsf{S}]?(s[\mathsf{C}], x_3).\mathtt{watch}\, t\ \mathtt{do}\ s[\mathsf{S}]!\langle ack\rangle.\mathbf{0}\{P_f\}$$

where P_f must be typable with $\mathsf{S}_{\mathsf{Part(G_f)}}(G_f, \emptyset) \upharpoonright \mathsf{S}$ and represents a process which may be executed in case of failure. $\triangle$

Example 10.64 (Messages in a Time-Frame Pattern). We recall the variants of this pattern presented in § 1.6:

(a) *Interval:* A is allowed to send B at most k messages, and at time intervals of at least t and at most r time units.

(b) *Overall time-frame:* A is allowed to send B at most k messages in the overall time-frame of at least t and at most r time units.

For this pattern we take advantage of both the time units induced by pauses in MRS and events. We present the MRS global types below and explain some of the choices made for the representation:

(a) For the interval pattern we assume that there is a number k of messages to be sent and that the lower and upper bound of the time-frame are t and r, respectively. Then, the global type follows:

$$\begin{aligned} G &= \mathtt{watch}\ end\ \mathtt{do}\ G(k)\ \mathtt{else}\ \mathtt{end} \\ G(k) &= \mathtt{pause}_t.\mathtt{watch}\ r\ \mathtt{do}\ \mathsf{A}{\uparrow}\langle \mathsf{M}_k, \{B\}\rangle.G(k-1)\ \mathtt{else}\ G_f \end{aligned}$$

In the protocol above, type $\mathtt{pause}_t$ represents a sequence $\mathtt{pause}.\cdots.\mathtt{pause}$ with t pauses representing the wait for the time-frame. Thanks to this sequence of pauses we can represent the necessary interval between messages. The upper bound of the timing constraint is implemented by using event r. As in Ex. 10.63, we use G_f to denote the protocol used in case of failure.

(b) Using a similar strategy as with the pattern above, we model the overall time-frame pattern:

$$\begin{aligned} G &= \mathtt{watch}\ r\ \mathtt{do}\ (\mathtt{watch}\ end\ \mathtt{do}\ G(k)\ \mathtt{else}\ \mathtt{end})\ \mathtt{else}\ G_f \\ G(k) &= \mathtt{pause}_t.\mathsf{A}{\uparrow}\langle \mathsf{M}_k, \{B\}\rangle.G(k-1) \end{aligned}$$

The protocol above is very similar to the interval pattern. The only difference is that we only check for the overall timeout r, instead of every individual one.

△

Example 10.65 (Action Duration Pattern). We recall the description of the pattern in § 1.6:

(a) The time elapsed between two actions of the same participant A must not exceed t time units.

Similarly to the request-response timeout pattern, we use the preemption construct for types to check the timing of the communications:

$$G = \mathtt{watch}\ t\ \mathtt{do}\ (\mathsf{p}\uparrow\langle\mathsf{M}_1, \Pi\rangle.\mathsf{A}\uparrow\langle\mathsf{M}_1, \{B\}\rangle.G')\ \mathtt{else}\ G_f$$

Above, event t represents a timeout event whose presence implies that the timing constraint was violated. Similarly, to the previous examples, G_f represents the alternative protocol in case of failure. △

Example 10.66 (Repeated Constraint Pattern). Below we recall the description of the repeated constraint pattern in § 1.6:

(a) A must send (and unbounded number of) messages to B every t time units.

This requirement can be represented in MRS using recursion and preemption:

$$G = \mathtt{watch}\ t\ \mathtt{do}\ (\mu\mathbf{t}.\mathsf{p}\uparrow\langle\mathsf{M}, \Pi\rangle.\mathbf{t})\ \mathtt{else}\ G_f$$

Above, t represents upper bound of the time-frame and G_f the alternative protocol in case of failure. △

11 Conclusions and Related Work

In this chapter we present the conclusions and related work of Ch. 10. In § 11.1 we explain some of our design choices and in § 11.2, we discuss further related work.

11.1 Concluding Remarks

We have developed a typed framework for multiparty sessions by building upon MRS, a new process calculus that integrates constructs from session-based concurrency with constructs from synchronous reactive programming (SRP). The calculus MRS accounts for broadcast communication, logical instants, and preemption – all of which are hard to represent in existing process languages for sessions, usually based on the π-calculus. For instance, a session π-calculus with broadcasting has been studied in [KGG14], but it does not support time-dependent interactions nor reactivity. Indeed, there are useful interaction patterns, such as *hot-service replacement* [MBC07], which are representable in SRP but not in asynchronous calculi. The semantics of MRS ensures typical properties of SRP, such as deadlock-freedom and reactivity.

Our type system for MRS crucially relies on *saturation* for global types, a notion that we developed to address the subtle distinction between explicit and implicit pauses, and to capture the "timing" of a protocol interaction within the global type itself. Another salient feature of our static analysis is a new notion of duality, suited to our broadcast setting, in which outputs do not need to be matched by inputs. Our notion of duality is also "time-aware" in that it requires dual participants to have matching pauses. The benefits of our integration reflect also in the semantic properties enforced by our type system: besides classical session safety properties, our static analysis guarantees two new time-related properties that seem to be desirable for sessions in a reactive setting: input timeliness and output persistence. We now discuss some design choices regarding our calculus:

- In our calculus MRS, the properties of deadlock-freedom and (bounded) reactivity are enforced by the operational semantics, while the classical properties of sessions, as well as the new time-related properties (input timeliness and output persistence), are enforced by our specific session type system. It could be argued that with a type system at hand, a more liberal semantics could have been used for the calculus, letting the type system take care also of the deadlock-freedom and reactivity properties. The reason for our choice is that deadlock-freedom and reactivity are essential properties of SRP: they are required for the synchronous reactive model to make sense. Hence they should be an integral part of the calculus itself. This is the case also for real SRP languages such as ReactiveML.

- Saturation could have been conceived as a well-formedness property of types rather than as an operation on them. In other words, our type system could have been designed so as to enforce the explicit separation of instants rather than relying on it. For instance, we could have defined a "well-timedness" predicate on global types requiring that subsequent broadcasts from the same sender and recursion unfoldings be separated by pauses. This way, we could have omitted some suspension rules ((out_s) and (in_s^2)) from the semantics, and the "More" typing rules from the type system. However, we chose the latter solution to avoid blurring the readability of real-world programs with the presence of too many pauses. The former solution would become an attractive option if combined with a "pause inference" mechanism.

11.2 Related Work

Most related work has been already mentioned in the introduction and throughout the chapter. Here we briefly discuss other relevant literature. In the realm of (multiparty) session types, forms of reactive behaviors have been addressed via constructs for exceptions, events, and run-time adaptation. This way, e.g., interactional exceptions for binary sessions were developed in [CHY08]; an associated type system ensures consistent handling of exceptions. The work [CGY16] extends this approach to multiparty sessions.

In general, there is a tension between forms of (interactional) exceptions and behavioral type systems, mainly due to the linearity enforced by such systems, which conflicts with the possibility of not (fully) consuming behaviors abstracted by types. The works [MV14, PG15] address this tension for binary sessions by appealing to *affinity* rather than to linearity. A similar approach is adopted in [FLMD19] for a functional programming language. The recent work [CP17] gives an alternative treatment of non-determinism and control effects within a Curry-Howard interpretation of binary session types; the proposed framework allows to represent exceptions. Concerning events and adaptation, the work [KYHH16] is the first to integrate events within a session-typed framework, supported by dynamic type inspection (a typecase construct). This work, however, is limited to binary sessions; the work [DP16] extends this framework to the multiparty setting, with the aim of handling run-time adaptation of choreographies. None of these works supports declarative or timed conditions, which are naturally expressible using synchronous programming constructs. To our knowledge, the only prior work that considers (unreliable) broadcasting in session types is [KGG14], which focuses on binary sessions. The work in [DHH^{+}15] develops run-time verification techniques for interruptible, multiparty conversations. Also, the work [CVB^{+}16] proposes protocol types for handling partial failures and ensuring absence of orphan messages and deadlocks, among other properties.

The most closely related work is that in [CAP17], which encodes of a binary session π-calculus into the synchronous reactive language ReactiveML. Using the duality between the two partners in binary sessions, this encoding simulates messages-over-channels in the session π-calculus by values-over-events in ReactiveML, slicing every session into a sequence of atomic instants: each instant corresponds to exactly one step in the protocol of the session. In contrast, instead of encoding a multiparty session calculus into RML, here we pursue a different goal: devise a minimal extension of a multiparty session calculus that accommodates reactive features, and provide a session type system that ensures the usual session properties together with some new semantic properties of interest: output persistence, input lock-freedom, safe event handling.

PART V

CLOSING REMARKS AND FUTURE PERSPECTIVES

12
Closing Remarks and Future Perspectives

12.1 Closing Remarks

Ensuring the *correctness* of message-passing programs is a difficult problem in Computer Science. The complexity of specifying these systems stems from the interplay of heterogeneous components that depend on a myriad of features, such as communication protocols, timed requirements, reactive behavior, and partial information. These features introduce a number of difficult issues that must be address to ensure correctness, including the following:

- Components specified in different languages may have potentially conflicting requirements.
- A component specification may describe features that are not expressible in the specifications of other components.
- Languages may be equipped with *reasoning techniques* that can only be used to analyze individual components and do not scale to the complete system.

The role that features such as partial information and timed behavior play in a specification largely depends on the *view* from which the system is analyzed. We have focused on two natural and commonly used views for message-passing systems: *operational* and *declarative*. In this dissertation, we have assumed that the operational view is concerned with *how* the system executes, whereas the declarative view is concerned with *what* the system should execute. In this sense, languages that fall within the operational view (i.e., *operational languages*) describe explicitly the instructions that govern the system's execution, while languages that fall within the declarative view (i.e., *declarative languages*) specify the conditions that govern the system's execution. In this dissertation we considered the following operational process calculi for message-passing concurrency:

- $\pi^{\lightning}_{\mathsf{OR}}$, a sub-class of π (proposed by Vasconcelos [Vas12], § 2.2) without output races (cf. § 3.1.1).
- π_{E}, an extension of $\pi^{\lightning}_{\mathsf{OR}}$ with session establishment constructs (cf. § 3.1.3).
- $\pi^{\lightning}_{\mathsf{R}}$, a sub-class of π without races (cf. § 3.1.2).
- $\mathsf{a}\pi$, a queue-based session π-calculus (cf. § 3.2).

and the following declarative languages, which are able to specify *partial information*, *timed behavior*, and *reactive behavior* in message-passing programs:

- `lcc`, a constraint language inspired by linear logic (cf. § 2.3).
- $\mathtt{lcc}^{\mathsf{p}}$, an extension of `lcc` with private information (cf. § 3.3).
- RML, a synchronous reactive programming language built upon OCaml (cf. § 2.4).
- RMLq, an extension of RML with queues and explicit states (cf. § 3.4).

Considering the previous context, we believe that a *unified view* is needed to address the issues that arise from the nature of message-passing programs. Our work has shown that this unified view can be used to obtain robust and comprehensive specifications of these programs. In particular, we showed that unified specifications allow us to investigate the interplay between both declarative and operational views by enabling a uniform analysis of the system's components. A key insight from our work is that declarative languages can provide a foundation for this unified view.

As a necessary step to develop the unified view we advocate, we also analyzed mechanisms to relate operational and declarative languages. In particular, we have investigated how *relative expressiveness* techniques can be used to cope with the issues that the heterogeneity of message-passing programs induce. In this regard, we have developed *encodings* that allow us to correctly translate (typed) operational languages for specifying message-passing programs into declarative languages (cf. Parts II and III). Moreover, to further investigate the relation between operational and declarative views, we have also developed a "hybrid" process calculus in which message-passing programs can be specified with timed and reactive constructs (cf. Part IV). Our results show that our approach provides a uniform setting for analyzing message-passing programs. Furthermore, we have also established that it is feasible to analyze declarative features in operational specifications by using our encodings—see § 4.4, § 5.4, and § 7.4 for concrete illustration.

12.1.1 Advantages

The core of our argument is that the correctness properties of encodings enable a unified view of message-passing programs. Indeed, encodings allow to translate the specifications of individual components into a common specific language which captures all the requirements of the program. There are two main advantages to our approach:

Uniform Setting for Message-Passing Programs: Our encodings allowed us to use the target language to analyze source processes. Thanks to the correctness properties of these encodings (i.e., *encodability criteria*) we were able to use the reasoning techniques from the target language to verify properties of translated specifications. Indeed, the encodability criteria we considered ensure that encoded target specifications preserve both the structure and behavior of source specifications. In this sense, an important advantage arises: if the target language is carefully selected, using encodings may enable the use of different reasoning techniques which may not be available in the source language. There are two specific examples of this in our work. The first one was presented in § 4.4 and § 7.4, where we used the encodings in Ch. 4 and Ch. 7 to specify the timed patterns for communication protocols in § 1.6. The second one refers to the security property proven for the encoding presented in Ch. 5 (cf. Thm. 5.51). This property ensures that the translations of well-typed networks in π_{E} (cf. § 3.1.3) are well-typed with respect to the type system for $\mathtt{lcc}^{\mathsf{p}}$ (cf. § 3.3). Hence, we guarantee that translations of well-typed networks respect the local information contained in $\mathtt{lcc}^{\mathsf{p}}$ abstractions.

Well-Behaved Message-Passing Specifications: We have shown that by focusing on specific classes of source processes we can reduce the gap between the source and target languages while obtaining strong correctness properties. This was possible because we focused on meaningful source specifications; in the setting of session-based concurrency this means *well-typed programs*. This property ensures that well-typed session programs never reduce to errors. Although our translations are defined over all possible source terms, our correctness properties are delimited to the translations of well-typed programs. This allowed us to prove encodability criteria that are stronger when compared to the properties we would have obtained by considering arbitrary source terms. Throughout this dissertation we used different type systems to narrow down the class of source programs considered for our encodings. In this sense, the type systems defined for $\pi^{\lightning}_{\mathsf{OR}}$ (cf. § 3.1.1) and $\pi^{\lightning}_{\mathsf{R}}$ (cf. § 3.1.2) should be highlighted, as they sharply define the domains of our translations. The relation between the classes of well-typed processes in $\pi^{\lightning}_{\mathsf{OR}}$, $\pi^{\lightning}_{\mathsf{OR}}$, and $\pi^{\lightning}_{\mathsf{R}}$ can be seen in Fig. 12.1.

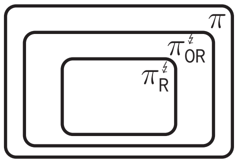

π
$\pi^{\lightning}_{\mathsf{OR}}$
$\pi^{\lightning}_{\mathsf{R}}$

Figure 12.1: Classes and sub-classes of well-typed processes in π, $\pi^{\lightning}_{\mathsf{OR}}$, and $\pi^{\lightning}_{\mathsf{R}}$.

The biggest rectangle represents the class of well-typed process in π. Observe that the class of well-typed processes in $\pi^{\lightning}_{\mathsf{OR}}$ is contained within the class of π since output races are allowed. Similarly, the class of well-typed $\pi^{\lightning}_{\mathsf{R}}$ processes is contained within the class of well-typed $\pi^{\lightning}_{\mathsf{OR}}$ programs, which allows input races. Using these sub-classes and the *operational correspondence* criterion (cf. Def. 2.3(3,4)) we ensure that the encodings of $\pi^{\lightning}_{\mathsf{OR}}$ into $\mathtt{lcc}$ (cf. Ch. 4) and of

$\pi^{\lightning}_{\mathsf{R}}$ into RML (cf. Ch. 7) produced translations that do not reduce to errors—in the sense of the source language. Indeed, the operational correspondence criterion guarantees that the behavior of source terms is preserved (i.e., *operational completeness*) and that the encoding does not introduce extraneous behaviors (i.e., *operational soundness*). Even with the valuable guidance given by the well-typedness of source specifications, this latter property proved to be rather challenging to establish in all our encodings.

12.1.2 Discussion

We believe that our work raises insightful points regarding the use of relative expressiveness for relating operational and declarative languages:

Language Differences: Relative expressiveness techniques have been traditionally used to compare the expressive power of relatively similar formal languages. For example, the work presented in [FG96, Bor98, Ama00, Mer00, Gor10, PN12, PN16, vG18] focuses on comparing process calculi such as the π-calculus and its many variants. In contrast, our work required the development of translations between very different languages. Indeed, our source languages are all message-passing and session-based, while our target languages exhibit differences that range from the concurrency model they adhere to (e.g., `lcc` is a shared-memory language) to their semantics (e.g., RML has a synchronous semantics). All these differences must be reconciled to develop meaningful encodings that enjoy strong correctness properties. Specific examples of how these differences manifested in our work and how we addressed them follow. First, consider the encoding of $\pi^{\lightning}_{\mathsf{OR}}$ into `lcc`, presented in Ch. 4. This encoding had to be developed in a way that enabled `lcc` to cope with the sequentiality present in $\pi^{\lightning}_{\mathsf{OR}}$, as `lcc` has no sequential composition operator. Notice that even with such mechanisms, `lcc` cannot encode all possible behaviors in $\pi^{\lightning}_{\mathsf{OR}}$. Still, `lcc` can capture the essence of session-based programs by focusing on well-typed $\pi^{\lightning}_{\mathsf{OR}}$ programs—see § 3.1.1. Second, in the encoding presented in Ch. 7 the differences are even more prominent since the semantics of RML induces *time instants* for RML programs. Moreover, RML uses a synchronous big-step semantics that allows multiple simultaneous synchronizations between event emissions and awaiting processes. This behavior contrasts with the semantics of $\pi^{\lightning}_{\mathsf{R}}$, in which a single-synchronization is executed in each reduction. We addressed this issue by equipping $\pi^{\lightning}_{\mathsf{R}}$ with a big-step semantics where a single big-step reduction represents several reduction steps and by developing a type system that disallows races in $\pi^{\lightning}_{\mathsf{R}}$—see § 3.1.2.

Determinism: A notorious difference that appeared in developing a unified view for message-passing programs is the (non-)determinism present in the various operational and declarative languages studied. For instance, while π (cf. § 2.2) induces rich forms of non-deterministic behaviors via unrestricted types, `lcc` (cf. § 2.3, [Hae11]) allows only for guarded non-deterministic choices and ReactiveML (cf. § 2.4, [MP05]) forbids it altogether. In concurrent constraint programming, non-determinism is known to heavily influence the definition of theoretical foundations such as denotational semantics [Ten76]—see, e.g., [dBPP95].

Similarly, the lack of non-determinism in ReactiveML is inherited from the synchronous programming paradigm, where properties such as determinism and confluence are essential in proving real-time programs correct [Hal98]. In our work we addressed these discrepancies by using specialized type systems for π: by narrowing down the forms of non-determinism in the source languages, the type systems we developed for $\pi_{\mathsf{0R}}^{\lightning}$ (cf. § 3.1.1) and for $\pi_{\mathsf{R}}^{\lightning}$ (cf. § 3.1.2) were instrumental to ensure that our encodings correctly capture the intended behavior of source terms.

Encodability Criteria: The previously mentioned differences between operational and declarative languages must be carefully considered when selecting the encodability criteria to guarantee the correctness of our encodings. This is because it is crucial for our unified view of message-passing programs to ensure that translated specifications preserve the correctness properties inherited from the source specifications. In particular, we wanted our translations to preserve the communication safety guarantees provided by session types in well-typed source processes. These guarantees often manifest themselves as conditions over the syntactic structure of processes (see, e.g., Def. 3.14, Def. 3.27). Therefore, we decided that our encodings should satisfy two *static criteria*: *name invariance* (cf. Def. 2.3(1)) and *compositionality* (cf. Def. 2.3(2)). Similarly, since we also wanted our translated specifications to preserve the behavior of well-typed session-based programs, we needed to consider the operational correspondence criterion already mentioned above. We believe that these criteria provide the minimal requirements needed to establish the foundations for our unified view. Given the differences between the languages covered in our work, it is unsurprising that the encodability criteria may require adjustments to capture the peculiarities of the languages involved and to induce the desired correctness properties in the encodings. As an example of this, the analysis of the translation of $\pi_{\mathsf{R}}^{\lightning}$ into RML (cf. Ch. 7) required the introduction of *refined encodings* (cf. Def. 2.6), which include a *coarser* form of operational completeness that accounts for the big-step semantics of RML.

Although a great deal of our work focused on encodings and their correctness, the work presented in Part IV offers a different kind of insights about the interplay of synchronous reactive programming and session-based concurrency. Since MRS is a multiparty calculus equipped with reactive constructs and whose communication model follows SRP, we have been able to analyze phenomena that, to the best of our knowledge, have not been studied in a uniform setting. In particular, MRS allowed us to investigate the interplay between broadcast, events, timed behavior, and reactive behavior without having to analyze each feature independently. Another interesting aspect of MRS is types not only statically check communication correctness, but also *timed properties* such as *output persistence*, which guarantees that processes execute an output action at least once during an instant, and *input timeliness*, which guarantees that every input is satisfied either in the current or the next instant. Also, we showed that the multiparty session types in MRS are expressive enough to specify and enforce the timed patterns presented in § 1.6 (cf. § 10.6).

12.2 Future Work

We now discuss possible directions for future work derived from the results presented here.

Session-Based Concurrency and Concurrent Constraint Programming: We would like to use the encodings presented in Part II to transfer reasoning techniques between session-based calculi and `lcc`. Two concrete examples of this kind of transference are the characterization of a class of deadlock-free `lcc` processes by encoding the session calculus in [CP10] and the use of the phase semantics presented in [FRS01] to prove *safety properties* for session π-calculi. We also would like to extend our encodings to consider the session π-calculus with asynchronous (queue-based), eventful semantics defined in [KYH11] as we believe `lcc` is a natural language to analyze event-driven behavior. Furthermore, since here we have focused on π-calculus implementations for *binary* session types, we think that extending our `lcc` encodings to address multiparty session processes [HYC16, BHTY10] would allow us to have rich declarative interpretations of communication scenarios with more than two participants. We also want to investigate how compositional is the translation in § 5 in terms of the session establishment protocols used. Hence, rather than the NSL protocol, we would like to analyze different session initiation protocols. In this way, we could investigate the security features that appear in different security protocols by using `lcc`. Finally, we would like to extend our results with notions such as nested locations and security levels. Indeed, we believe we can reason about security and space-related notions in session-based concurrency by using techniques adopted from recent constraint-based declarative languages [KPPV12, GHP$^+$17].

Session-Based Concurrency and Synchronous Reactive Programming: We plan to explore more in depth the limits of the runnable encodings presented in Part III. In particular, we want to develop a session type library in ReactiveML, similar to the one created by Padovani in OCaml [Pad, Pad17]. Moreover, it should be possible to obtain similar encodability results by using variants of session π-calculi such as the ones presented in by di Giusto and Pérez, and Castellani et al. in [DP16, CDP16]. Results on this line would clarify the role of the synchronous reactive paradigm in issues such as runtime adaptation and deadlock resolution. Furthermore, we believe that extending our work on reactive session-based concurrency to consider notions of security, following the approach presented in [MBC07] would help us clarify notions such as *non-interference* in reactive languages. Such extension would also require the development of behavioral equivalences for ReactiveML. Lastly, we propose the use of the *Globally Asynchronous Locally Synchronous* (GALS) model [BCE$^+$03] to relax the synchrony assumption imposed by the synchronous programming paradigm in our encodings. This would allow us to obtain more realistic specifications for session-based concurrency. On a more foundational level, we would like to investigate possible translations from reactive synchronous calculi such as the ones in [Ama07, ABBC06] into the π-calculus. Such results would allow us to compare the expressive power of reactive constructs such as, e.g., the preemption operator.

A Synchronous Reactive Session-Based Calculus: Directions for future work based on the calculus presented in Part IV include the extension of MRS with a runtime monitoring mechanism such as the ones in [DP16, FPS18]. We believe that work in this direction would allow us to naturally check events, as they are intrinsically *runtime structures*, not static entities. Moreover, we believe that using runtime verification would help us clarify the role that preemption and events play in failure handling and interruption mechanisms [DHH$^+$15, APN17]. We also think that the preemption construct in MRS provides a fertile ground for analyzing runtime adaptation and deadlock resolution as presented in [CDP16, DP16]. Moreover, we would like to explore whether ReactiveML is an acceptable host language for implementing MRS. Similarly to the work with encodings on Part III, we want to extend MRS to consider the GALS paradigm. In this way we would be able to relax the synchrony requirements for processes, allowing them to have different clocks and inhabit different localities. Yet another line of future work for MRS consists in the addition of selection and branching constructs. This would allow us to make closer comparisons of this calculus with respect to MPST.

Bibliography

[ABB$^+$16] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

[ABBC06] Roberto M. Amadio, Gérard Boudol, Frédéric Boussinot, and Ilaria Castellani. Reactive concurrent programming revisited. *Electr. Notes Theor. Comput. Sci.*, 162:49–60, 2006.

[ACP16] Jaime Arias, Mauricio Cano, and Jorge A. Pérez. Towards A practical model of reactive communication-centric software. In *Proc. of the Italian Conference on Theoretical Computer Science (ICTCS).*, volume 1720 of *CEUR Workshop Proceedings*, pages 227–233. CEUR-WS.org, 2016.

[AGPV06] Jesús Aranda, Cinzia Di Giusto, Catuscia Palamidessi, and Frank D. Valencia. On recursion, replication and scope mechanisms in process calculi. In *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, pages 185–206, 2006.

[Ama] Amazon. Amazon web services website. `https://aws.amazon.com/`. Accessed: 2019-13-08.

[Ama00] Roberto M. Amadio. On modelling mobility. *Theoretical Computer Science*, 240(1):147–176, 2000.

[Ama07] Roberto M. Amadio. A synchronous pi-calculus. *Inf. Comput.*, 205(9):1470–1490, 2007.

[APN17] Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, pages 1–16, 2017.

[APV19] Alen Arslanagic, Jorge A. Pérez, and Erik Voogd. Minimal session types (pearl). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, pages 23:1–23:28, 2019.

[Arb16] Farhad Arbab. Proper protocol. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 65–87, 2016.

[Ari15] Jaime Arias. *Formal Semantics and Automatic Verification of Hierarchical Multimedia Scenarios with Interactive Choices.* (*Sémantique Formelle et Vérification Automatique de Scénarios Hiérarchiques Multimédia avec des Choix Interactifs*). PhD thesis, University of Bordeaux, France, 2015.

[BBK87] Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. On the consistency of koomen's fair abstraction rule. *Theor. Comput. Sci.*, 51:129–176, 1987.

[BCDM11] Maria Grazia Buscemi, Mario Coppo, Mariangiola Dezani-Ciancaglini, and Ugo Montanari. Constraints for service contracts. In *Trustworthy Global Computing - 6th International Symposium, TGC 2011, Aachen, Germany, June 9-10, 2011. Revised Selected Papers*, pages 104–120, 2011.

[BCE$^+$03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BCM$^+$15] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In *FORTE*, volume 9039 of *LNCS*, pages 161–177. Springer, 2015.

[BdBP97] Eike Best, Frank S. de Boer, and Catuscia Palamidessi. Partial order and SOS semantics for linear constraint programs. In *Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings*, pages 256–273, 1997.

[BDGK14] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. In *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy*, pages 51–66, 2014.

[BdS96] Frédéric Boussinot and Robert de Simone. The SL synchronous language. *IEEE Trans. Software Eng.*, 22(4):256–266, 1996.

[BFM98] Howard Bowman, Giorgio P. Faconti, and Mieke Massink. Specification and verification of media constraints using UPAAL. In *Design, Specification and Verification of Interactive Systems'98, Proceedings of the Fifth International Eurographics Workshop, June 3-5, 1998, Abingdon, United Kingdom, Volume 1*, pages 261–277, 1998.

[BG92] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[BHTY10] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 162–176. Springer - Verlag, 2010.

[BJPV11] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011.

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BM07a] Maria Grazia Buscemi and Hernán C. Melgratti. Transactional service level agreement. In *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, pages 124–139, 2007.

[BM07b] Maria Grazia Buscemi and Ugo Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *ESOP 2007*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.

[BM08] Maria Grazia Buscemi and Ugo Montanari. Open bisimulation for the concurrent constraint pi-calculus. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 254–268, 2008.

[BM11] Maria Grazia Buscemi and Ugo Montanari. Cc-pi: A constraint language for service negotiation and composition. In *Results of the SENSORIA Project*, volume 6582 of *LNCS*, pages 262–281. Springer, 2011.

[BMP13] Guillaume Baudart, Louis Mandel, and Marc Pouzet. Programming mixed music in ReactiveML. In *ACM SIGPLAN Workshop on Functional Art, Music, Modeling and Design (FARM'13)*, Boston, USA, September 2013. Workshop ICFP 2013.

[BMS15] Frédéric Boussinot, Bernard Monasse, and Jean-Ferdy Susini. Reactive programming of simulations in physics. *International Journal of Modern Physics C*, 26(12):1550132, 2015.

[BMVY19] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Asynchronous timed session types - from duality to time-sensitive processes. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Prague, Czech Republic, Proceedings*, pages 583–610, 2019.

[Bor98] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998.

[Bou91] Frédéric Boussinot. Reactive C: an extension of C to program reactive systems. *Softw., Pract. Exper.*, 21(4):401–428, 1991.

[Bou92] Gérard Boudol. Asynchrony and the Pi-calculus. Research Report RR-1702, INRIA, 1992.

[Bou04] Gérard Boudol. ULM: A core programming model for global computing: (extended abstract). In *13th European Symposium on Programming, ESOP*, pages 234–248, 2004.

[Bou06] Frédéric Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, 2006.

[BP07] Luca Bortolussi and Alberto Policriti. Stochastic concurrent constraint programming and differential equations. *Electr. Notes Theor. Comput. Sci.*, 190(3):27–42, 2007.

[BS00] Frédéric Boussinot and Jean-Ferdy Susini. Java threads and sugarcubes. *Softw., Pract. Exper.*, 30(5):545–566, 2000.

[BTZ12] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. Contract-oriented computing in CO2. *Sci. Ann. Comp. Sci.*, 22(1):5–60, 2012.

[BYY14] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *Proc. of CONCUR'14*, volume 8704, pages 419–434. Springer, 2014.

[BZ09] Massimo Bartoletti and Roberto Zunino. A calculus of contracting processes. Technical Report DISI-09-056, University of Trento, 2009.

[BZ10] Massimo Bartoletti and Roberto Zunino. A calculus of contracting processes. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 332–341, 2010.

[CAP16] Mauricio Cano, Jaime Arias, and Jorge A. Pérez. A reactive interpretation session-based concurrency. Workshop on Reactive and Event-based Languages & Systems (REBLS), co-located with the ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH), 2016.

[CAP17] Mauricio Cano, Jaime Arias, and Jorge A. Pérez. Session-based concurrency, reactively. In *Proc. of the Int. Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 74–91, 2017.

[CCDGP19] Mauricio Cano, Ilaria Castellani, Cinzia Di Giusto, and Jorge A. Pérez. Multiparty Reactive Sessions. Research Report 9270, INRIA, April 2019.

[CD09] Mario Coppo and Mariangiola Dezani-Ciancaglini. Structured communications with concurrent constraints. In *Proc. of TGC 2008*, volume 5474 of *LNCS*, pages 104–125. Springer, 2009.

[CDP16] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Asp. Comput.*, 28(4):669–696, 2016.

[CDPY15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, pages 146–178, 2015.

[CDV15] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Self-adaptive multiparty sessions. *Service Oriented Computing and Applications*, 9(3-4):249–268, 2015.

[CGHL10] Marco Carbone, Davide Grohmann, Thomas T. Hildebrandt, and Hugo A. López. A logic for choreographies. In *Proc. of PLACES 2010*, pages 29–43, 2010.

[CGY16] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):156–205, 2016.

[CHY08] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR'8, Toronto, Canada, August 19-22, 2008*, pages 402–417, 2008.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21th International Conference, CONCUR 2010, Paris, France, 2010. Proceedings*, pages 222–236, 2010.

[CP17] Luís Caires and Jorge A. Pérez. Linearity, control effects, and behavioral types. In *ESOP'17, ETAPS'17, Uppsala, Sweden, April 22-29, 2017*, pages 229–259, 2017.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL 1987, Proceedings*, pages 178–188, 1987.

[CPS09] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 33–37, 2009.

[CRLP15] Mauricio Cano, Camilo Rueda, Hugo A. López, and Jorge A. Pérez. Declarative interpretations of session-based concurrency. In *Proc. of the Int. Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 67–78. ACM, 2015.

[CRS18] Rance Cleaveland, A. W. Roscoe, and Scott A. Smolka. Process algebra and model checking. In *Handbook of Model Checking.*, pages 1149–1195. Springer, 2018.

[CVB$^+$16] Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. A type theory for robust failure handling in distributed systems. In

FORTE'16, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, pages 96–113, 2016.

[dBGM00] Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000.

[dBPP95] Frank S. de Boer, Alessandra Di Pierro, and Catuscia Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theor. Comput. Sci.*, 151(1):37–78, 1995.

[DGS12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proc. of PPDP'12*, pages 139–150, 2012.

[DHH$^+$15] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods in System Design*, 46(3):197–225, 2015.

[DP16] Cinzia Di Giusto and Jorge A. Pérez. Event-based run-time adaptation in communication-centric systems. *Formal Asp. Comput.*, 28(4):531–566, 2016.

[DRV98] Juan F. Díaz, Camilo Rueda, and Frank D. Valencia. Pi+- calculus: A calculus for concurrent processes with constraints. *CLEI Electron. J.*, 1(2), 1998.

[dS84] Robert de Simone. On meije and SCCS: infinite sum operators VS. non-guarded definitions. *Theor. Comput. Sci.*, 30:133–138, 1984.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 372–385, 1996.

[FH93] Thom W. Frühwirth and Philipp Hanschke. Terminological reasoning with constraint handling rules. In *Principles and Practice of Constraint Programming, PPCP 1993, Newport, Rhode Island*, pages 80–89, 1993.

[FLMD19] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types Without Tiers. In *Proc. POPL'19*, pages 1–29, New York, NY, USA, 2019. ACM.

[FMR$^+$09] Dirk Fahland, Jan Mendling, Hajo A. Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus imperative process modeling languages: The issue of maintainability. In *Business Process Management Workshops, BPM 2009 International Workshops, Ulm, Germany, September 7, 2009. Revised Papers*, pages 477–488, 2009.

[Fok09] Wan Fokkink. Process algebra: An algebraic theory of concurrency. In *Algebraic Informatics, Third International Conference, CAI 2009, Thessaloniki, Greece, May 19-22, 2009, Proceedings*, pages 47–77, 2009.

[Fow16] Simon Fowler. An erlang implementation of multiparty session actors. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, pages 36–50, 2016.

[FPS18] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime verification for decentralised and distributed systems. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, pages 176–210. Springer, 2018.

[FRS01] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Inf. Comput.*, 165(1):14–41, 2001.

[FV13] Juliana Franco and Vasco Thudichum Vasconcelos. A concurrent programming language with refined session types. In *Software Engineering and Formal Methods - SEFM 2013, Revised Selected Papers*, pages 15–28, 2013.

[GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[GHP$^+$17] Michell Guzmán, Stefan Haar, Salim Perchy, Camilo Rueda, and Frank D. Valencia. Belief, knowledge, lies and other utterances in an algebra for space and extrusion. *J. Log. Algebr. Meth. Program.*, 86(1):107–133, 2017.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[GLGB87] Thierry Gautier, Paul Le Guernic, and Löic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, UK, 1987. Springer-Verlag.

[Gor10] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[GV10] Marco Giunti and Vasco Thudichum Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR*, LNCS, pages 432–446. Springer, 2010.

[Hae11] Rémy Haemmerlé. Observational equivalences for linear logic concurrent constraint languages. *TPLP*, 11(4-5):469–485, 2011.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 1–16, 1998.

[Har] Robert Harper. What, if anything, is a declarative language? Last Accessed On: September 2019. URL: `https://existentialtype.wordpress.com/2013/07/18/what-if-anything-is-a-declarative-language/`.

[HBS73] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245, 1973.

[Hen07] Matthew Hennessy. *A distributed Pi-calculus*. Cambridge University Press, 2007.

[HL09] Thomas T. Hildebrandt and Hugo A. López. Types for secure pattern matching with local knowledge in universal concurrent constraint programming. In *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, pages 417–431, 2009.

[HM10] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010.*, pages 59–73, 2010.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, pages 509–523, 1993.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of ESOP'98*, volume 1381, pages 122–138. Springer, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proc. of POPL'08*, pages 273–284. ACM, 2008.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

[KCD$^+$09] Slim Kallel, Anis Charfi, Tom Dinkelaker, Mira Mezini, and Mohamed Jmaiel. Specifying and monitoring temporal properties in web services compositions. In *Seventh IEEE European Conference on Web Services (ECOWS 2009), 9-11 November 2009, Eindhoven, The Netherlands*, pages 148–157, 2009.

[KDPG18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Sci. Comput. Program.*, 155:52–75, 2018.

[KGG14] Dimitrios Kouzapas, Ramunas Gutkovas, and Simon J. Gay. Session types for broadcasting. In *PLACES 2014*, volume 155 of *EPTCS*, pages 25–31, 2014.

[Kle08] J. Klensin. Simple mail transfer protocol. Last Accessed: July, 2019. URL: `https://tools.ietf.org/html/rfc5321`, October 2008.

[Kow79] Robert A. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[Kow88] Robert A. Kowalski. The early years of logic programming. *Commun. ACM*, 31(1):38–43, 1988.

[KPPV12] Sophia Knight, Catuscia Palamidessi, Prakash Panangaden, and Frank D. Valencia. Spatial and epistemic modalities in constraint-based process calculi. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, pages 317–332, 2012.

[KYH11] Dimitrios Kouzapas, Nobuko Yoshida, and Kohei Honda. On asynchronous session semantics. In *Proc. of FORTE'11*, volume 6722 of *LNCS*, pages 228–243. Springer, 2011.

[KYHH16] Dimitrios Kouzapas, Nobuko Yoshida, Raymond Hu, and Kohei Honda. On asynchronous eventful session semantics. *Mathematical Structures in Computer Science*, 26(2):303–364, 2016.

[LOP09] Hugo A. López, Carlos Olarte, and Jorge A. Pérez. Towards a unified framework for declarative structured communications. In *PLACES 2009, York, UK, 22nd March 2009.*, volume 17 of *EPTCS*, pages 1–15, 2009.

[Low96] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Software - Concepts and Tools*, 17(3):93–102, 1996.

[MB05] Louis Mandel and Farid Benbadis. Simulation of mobile ad hoc network protocols in ReactiveML. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05)*, Edinburgh, Scotland, April 2005. Electronic Notes in Theoretical Computer Science.

[MBC07] Ana Almeida Matos, Gérard Boudol, and Ilaria Castellani. Typing noninterference for reactive programs. *J. Log. Algebr. Program.*, 72(2):124–156, 2007.

[Mer00] Massimo Merro. Locality and polyadicity in asynchronous name-passing calculi. In *Foundations of Software Science and Computation Structures, Third International Conference, FOSSACS 2000, Held as Part of the Joint European Conferences on Theory and Practice of Software,ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, pages 238–251, 2000.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, LFCS, University of Edinburgh, 1991.

[Mil99] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[MM07] Lionel Morel and Louis Mandel. Executable contracts for incremental prototypes of embedded systems. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'07)*, March 2007.

[MM12] Rubén Monjaraz and Julio Mariño. From the π-calculus to flat GHC. In *Proc. of PPDP'12*, pages 163–172. ACM, 2012.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[MP05] Louis Mandel and Marc Pouzet. ReactiveML: a reactive extension to ML. In *Proc. of PPDP'05*, pages 82–93. ACM, 2005.

[MP14] Louis Mandel and Cédric Pasteur. Reactivity of Cooperative Systems - Application to ReactiveML. In *21st International Symposium, SAS 2014, Munich, Germany, 2014.*, pages 219–236, 2014.

[MPP15a] Louis Mandel, Cédric Pasteur, and Marc Pouzet. ReactiveML, ten years later. In *Proc. of PPDP 2015*, pages 6–17, 2015.

[MPP15b] Louis Mandel, Cédric Pasteur, and Marc Pouzet. Time refinement in a functional synchronous language. *Sci. Comput. Program.*, 111:190–211, 2015.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[MPYZ19] Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. Motion session types for robotic interactions (brave new idea paper). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom.*, pages 28:1–28:27, 2019.

[MT90] Faron Moller and Chris M. N. Tofts. A temporal calculus of communicating systems. In *CONCUR '90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings*, pages 401–415, 1990.

[MV14] Dimitris Mostrous and Vasco Thudichum Vasconcelos. Affine sessions. In *COORDINATION'14, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 115–130, 2014.

[NBY14] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. In *Proc. of BEAT'14*, volume 162, pages 19–26, 2014.

[NBY17] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.

[NPV02] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.

[NT04] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL 2004, USA, June 18-19, 2004, Proceedings*, pages 56–70, 2004.

[NYH12] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, pages 202–218, 2012.

[OV08a] Carlos Olarte and Frank D. Valencia. The expressivity of universal timed CCP: undecidability of monadic FLTL and closure operators for security. In *Proc. of PPDP'08*, pages 8–19. ACM, 2008.

[OV08b] Carlos Olarte and Frank D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 145–150, 2008.

[OY16] Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL 2016, USA, 2016*, pages 568–581, 2016.

[Pad] Luca Padovani. FuSe - A simple library implementation of binary sessions. URL: `http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html`.

[Pad17] Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.

[Par00] Joachim Parrow. Trios in concert. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 623–638, 2000.

[Par08] Joachim Parrow. Expressiveness of process algebras. *Electr. Notes Theor. Comput. Sci.*, 209:173–186, 2008.

[PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, 2007.

[Pér10] Jorge A. Pérez. *Higher-Order Concurrency: Expressiveness and Decidability Results*. PhD thesis, University of Bologna, Italy, 2010.

[Pet62] C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*, pages 386–390, 1962.

[Pet12] Kirstin Peters. *Translational Expressiveness. Comparing Process Calculi using Encodings*. PhD thesis, Berlin Institute of Technology, 2012.

[PG15] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 3–22, 2015.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[PN12] Kirstin Peters and Uwe Nestmann. Is it a "good" encoding of mixed choice? In *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 210–224, 2012.

[PN16] Kirstin Peters and Uwe Nestmann. Breaking symmetries. *Mathematical Structures in Computer Science*, 26(6):1054–1106, 2016.

[PSVV06] Catuscia Palamidessi, Vijay A. Saraswat, Frank D. Valencia, and Björn Victor. On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 59–68, 2006.

[PT08] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proc. of Symposium on Haskell'08*, pages 25–36. ACM, 2008.

[PvG15] Kirstin Peters and Rob J. van Glabbeek. Analysing and comparing encodability criteria for process calculi. *Archive of Formal Proofs*, 2015, 2015.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[San09] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral dissertation awards. MIT Press, 1993.

[SJG94] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 71–80, 1994.

[SL92] Vijay Saraswat and Patrick Lincoln. Higher-order, linear, concurrent constraint programming. Technical report, Xerox Parc, 1992.

[SLS$^+$18] Rodrigo C. M. Santos, Guilherme F. Lima, Francisco Sant'Anna, Roberto Ierusalimschy, and Edward Hermann Haeusler. A memory-bounded, deterministic and terminating semantics for the synchronous programming language céu. In *Proceedings of LCTES 2018*, pages 1–18. ACM, 2018.

[SMMM06] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. GLONEMO: Global and accurate formal models for the analysis of ad hoc sensor networks. In *Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)*, Nice, France, May 2006.

[SY16] Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP 2016*, LIPIcs. Dagstuhl, 2016.

[Ten76] Robert D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, 1976.

[TS05] Olivier Tardieu and Robert de Simone. Loops in Esterel. *ACM Trans. Embed. Comput. Syst.*, 4(4):708–750, November 2005.

[Uni] Uppsala University. Uppaal tool website. `http://www.uppaal.org/`. Accessed: 2019-13-08.

[Vas12] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.

[vG18] Rob van Glabbeek. A theory of encodings and expressiveness (extended abstract) - (extended abstract). In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 183–202, 2018.

[Wad14] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

[YHNN13] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, pages 22–41, 2013.

[YV07] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

A
Chapter 3

A.1 Proofs for $\pi_{\mathsf{OR}}^{\lightning}$

Lemma 3.7 (Basic Properties for Types). *Let T be a $\pi_{\mathsf{OR}}^{\lightning}$ type such that $\overline{T}$ is defined. Then, all of the following holds:*

1. *If $\mathsf{un}(T)$ then one of the following holds:*
   (*a*) *If $T = \mathsf{end}$ or $T = \mathsf{a}$ then $\neg\mathtt{out}(\overline{T})$ holds.*
   (*b*) *If $T = \mu\mathsf{a}.T$ or $T = qp$ then $\mathtt{out}(\overline{T})$ holds.*
2. *If $\mathtt{lin}(T)$ then $\mathtt{lin}(\overline{T})$ holds.*

Proof (*see Page 69*). The proof of (2) proceeds by induction on the structure of T. All the cases are immediate by Def. 2.15. The most interesting proof is for (1). By assumption, $\mathsf{un}(T)$ and $\neg\mathtt{out}(T)$ are true. We proceed by induction on the structure of T. The base cases are $T = \mathsf{end}$ and $T = \mathsf{a}$; they are immediate and fall on Item (a) (i.e., $\neg\mathtt{out}(\mathsf{a})$ and $\neg\mathtt{out}(\mathsf{end})$ hold). For the inductive step, we consider two cases: (1) whenever $T = \mu\mathsf{a}.T$, and (2) whenever $T = qp$. Case (1) is immediate by applying the IH. We detail Case (2): by the definition of $\mathsf{un}(\cdot)$, it must be the case that $\neg\mathtt{out}(T)$ holds and $q = \mathsf{un}$. The previous implies that $T = ?T_1.T_2$ with $\neg\mathtt{out}(T_2)$ holds or $T = \&\{l_i : T_i\}_{i\in I}$ with $\neg\mathtt{out}(T_i)$ true, for every $i \in I$. For each of these cases we have that, by Def. 2.15, $\mathtt{out}(\overline{T})$ is true. □

Lemma 3.8 (Properties of Typing Environments). *Let $\Gamma = \Gamma_1 \circ \Gamma_2$. Then all of the following hold:*

1. *$\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$.*
2. *Suppose that $x : qp \in \Gamma \wedge (q = \mathtt{lin} \vee (q = \mathsf{un} \wedge \mathtt{out}(p) = \mathtt{tt}))$. Then, either $x : qp \in \Gamma_1$ and $x \notin dom(\Gamma_2)$ or $x : qp \in \Gamma_2$ and $x \notin dom(\Gamma_1)$.*

3. $\Gamma = \Gamma_2 \circ \Gamma_1$.

4. *If* $\Gamma_1 = \Delta_1 \circ \Delta_2$ *then* $\Delta = \Delta_2 \circ \Gamma_2$ *and* $\Gamma = \Delta_1 \circ \Delta$.

Proof (*see Page 69*). Every item is proven by induction on the structure of Γ and by using the definition of splitting and predicates $\mathsf{un}(\cdot)$ and $\mathtt{lin}(\cdot)$ appropriately:

1. The base case is $\Gamma = \emptyset$. Then, $\Gamma_1 = \emptyset$ and $\Gamma_2 = \emptyset$. Moreover, $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2) = \emptyset$. For the inductive step, consider $\Gamma = \Gamma', x : T$. There are two subcases, depending on whether $\mathsf{un}(T)$ or $\mathtt{lin}(T)$. In the former case, we have that $\Gamma', x : T = \Gamma_1 \circ \Gamma_2$, and by Def. 2.16, $x : T \in \Gamma_1$ and $x : T \in \Gamma_2$. By IH, $\Gamma' = \Gamma_1' \circ \Gamma_2'$ and $\mathcal{U}(\Gamma') = \mathcal{U}(\Gamma_1') = \mathcal{U}(\Gamma_2')$. Since $\mathsf{un}(T)$, then $x : T \in \mathcal{U}(\Gamma', x : T)$, $x : T \in \mathcal{U}(\Gamma_1)$, and $x : T \in \mathcal{U}(\Gamma_2)$. Thus, $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$. The latter case is immediate by IH since $x : T \notin \mathcal{U}(\Gamma)$.

2. The base case is $\Gamma = \emptyset$. It is immediate as the empty environment does not contain any elements. For the inductive step, assume $\Gamma = \Gamma, x : qp$. There are two cases: if $q = \mathtt{lin}$ then the proof is immediate by Def. 2.16; if $q = \mathsf{un}$ then, by assumption, $\mathsf{out}(p) = \mathtt{tt}$. This implies that $\mathsf{un}(qp) = \mathtt{ff}$; therefore, since $\mathtt{lin}(qp)$ holds, we can conclude the proof by Def. 2.16.

3. Immediate by commutativity of the ',' for environments.

4. Immediate by associativity of the '$\circ$' operation (cf. Def. 2.16).

$\square$

Lemma 3.11 (Subject Congruence). *If* $\Gamma \vdash P$ *and* $P \equiv_{\mathsf{S}} Q$ *then* $\Gamma \vdash Q$.

Proof (*see Page 70*). By a case analysis on the typing derivation for each member of each axiom for $\equiv_{\mathsf{S}}$. The most interesting ones are: (1) $P \,|\, \mathbf{0} \equiv_{\mathsf{S}} P$, and (2) $(\boldsymbol{\nu} xy)(P \,|\, Q)$ if $x, y \notin \mathsf{fv}_\pi(Q)$.

Case $P \mid \mathbf{0} \equiv_{\mathsf{S}} P$**:**

$\Rightarrow$)	(1)	$\Gamma \vdash P \mid \mathbf{0}$	(Assumption)
	(2)	$\Gamma = \Gamma_1 \circ \Gamma_2$	(Inv. on Rule (T:Par), (1))
	(3)	$\Gamma_1 \vdash P$	(Inv. on Rule (T:Par), (2))
	(4)	$\Gamma_2 \vdash \mathbf{0}$	(Inv. on Rule (T:Par), (2))
	(5)	$\mathsf{un}(\Gamma_2)$	(Inv. on Rule (T:Nil) to (4))
	(6)	$\Gamma_1 \circ \Gamma_2 \vdash P$	(Lem. 3.9 to (3), (5))

$\Leftarrow$)	(1)	$\Gamma \vdash P$	(Assumption)
	(2)	$\emptyset \vdash \mathbf{0}$	(Rule (T:Nil) to $\emptyset$ - $\mathsf{un}(\emptyset)$ is true)
	(3)	$\Gamma \vdash P \mid \mathbf{0}$	(Rule (T:Par) to (1), (2))

Case $(\boldsymbol{\nu} xy)P \mid Q \equiv_{\mathsf{S}} (\boldsymbol{\nu} xy)(P \mid Q)$ **with** $x, y \notin \mathsf{fv}_\pi(Q)$**:**

$\Rightarrow$) By assumption we have that $\Gamma \vdash (\boldsymbol{\nu} xy)P \mid Q$ and $x, y \notin \mathsf{fv}_\pi(Q)$. Then:

(1)	$\Gamma = \Gamma_1 \circ \Gamma_2$	(Inv. on Rule (T:Par), Assumption)
(2)	$\Gamma_1 \vdash (\boldsymbol{\nu} xy)P$	(Inv. on Rule (T:Par), (1))
(3)	$\Gamma_2 \vdash Q$	(Inv. on Rule (T:Par), (1))
(4)	$\Gamma_1, x : T, y : \overline{T} \vdash P$	(Inv. on Rule (T:Res) on (2))

We now distinguish two cases. The first and most interesting one corresponds to whenever $\mathsf{un}(T)$ is true. The second one corresponds to whenever $\mathsf{un}(T)$ is false (i.e., which groups the remaining possibilities):

$\mathsf{un}(T)$ **is true:** In this case we distinguish two further sub-cases depending on whether the $\mathsf{un}(\overline{T})$ holds or not:

$\mathsf{un}(\overline{T})$ **is true:** If this is the case, then we apply Lem. 3.9 twice in (6) to obtain (7) $\Gamma_2, x : T, y : \overline{T} \vdash Q$ and conclude by applying (T:Par) and (T:Res) on (5) and (7).

$\mathsf{un}(\overline{T})$ **is false:** If this is the case, then we only apply weakening once (cf. Lem. 3.9) in (6) to obtain (7) $\Gamma_2, x : T \vdash Q$ and conclude by applying (T:Par) and (T:Res) to (5) and (7).

$\mathsf{un}(T)$ **is false:** In this case, we have two cases depending on $\mathsf{un}(T)$. Each case proceeds similarly as above.

$\Leftarrow$)

(1)	$\Gamma \vdash (\boldsymbol{\nu} xy)(P \mid Q)$	(Assumption)
(2)	$x, y \notin \mathsf{fv}_\pi(Q)$	(Assumption)
(3)	$\Gamma, x : T, y : \overline{T} \vdash P \mid Q$	(Inv. on Rule (T:Res), (1))

We distinguish two further cases depending on whether $\mathsf{un}(T)$ is true or false:

$\mathsf{un}(T)$ **is true:** In this case we distinguish two further sub-cases depending on whether the $\mathsf{un}(\overline{T})$ holds or not:

$\mathsf{un}(\overline{T})$ **is true:** If this is the case, then by inversion on (T:Par) on (3), we have that (4) $\Gamma, x : T, y : \overline{T} \vdash P$ and (5) $\Gamma_2, x : T, y : \overline{T} \vdash Q$. Then, we apply Rule (T:Res) on (4) to obtain (6) $\Gamma_1 \vdash (\boldsymbol{\nu} xy)P$ and we apply Lem. 3.10(2) to (5), obtaining (7) $\Gamma_2 \vdash Q$. We conclude by applying Rule (T:Par) to (6) and (7).

$\mathsf{un}(\overline{T})$ **is false:** If this is the case, then by inversion on (T:Par) on (3), we have that (4) $\Gamma, x : T, y : \overline{T} \vdash P$ and (5) $\Gamma_2 \vdash Q$. Moreover, by Lem. 3.10(1) applied to (2) and (5), it must be the case that $y : \overline{T} \notin \Gamma_2$. Therefore, we apply Lem. 3.10(2) once to remove $x : T$ from Γ_2. Then, we apply Rule (T:Res) in (4) to obtain (6) $\Gamma \vdash (\boldsymbol{\nu} xy)P$ and conclude by applying Rule (T:Par).

$\mathsf{un}(T)$ **is false:** In this case, we have two cases depending on $\mathsf{un}(T)$. Each case proceeds similarly as above.

□

Lemma 3.12 (Substitution). *If* $\Gamma_1 \vdash v : T$ *and* $\Gamma_2, x : T \vdash P$ *then* $\Gamma \vdash P\{^v\!/x\}$, *with* $\Gamma = \Gamma_1 \circ \Gamma_2$.

Proof (see Page 70). By induction on the structure of P. There are nine cases: one base case and eight inductive cases.

Base Case: $P = \mathbf{0}$. By inversion, $\mathsf{un}(\Gamma_2, x : T)$, which implies $\mathsf{un}(T)$. By inversion on the rules for values, we also have that $\mathsf{un}(\Gamma_1)$. Thus, $\mathsf{un}(\Gamma)$, with $\Gamma = \Gamma_1 \circ \Gamma_2$ holds. Since $\mathbf{0}\{^v\!/x\} = \mathbf{0}$, the proof concludes by applying Rule (T:Nil).

Inductive Step: The proofs for $P = Q_1 \mid Q_2$, $P = (\boldsymbol{\nu} yz)Q$, and $P = u?\,(Q_1):(Q_2)$ follow by applying the IH. The other five cases proceed similarly. We only detail the case for $P = y\langle u\rangle.Q$.

$P = y\langle u\rangle.Q$**:** We distinguish four sub-cases: (1) $y = x$ and $\neg\mathsf{un}(T)$, (2) $y = x$ and $\mathsf{un}(T)$, (3) $u = x$ and $\neg\mathsf{un}(T)$, (4) $u = x$ and $\mathsf{un}(T)$. Notice that sub-case (2) is not possible since by assumption $\Gamma, x : T \vdash P$, which implies that the judgment proceeds with Rule (T:Out) and therefore, by inversion $T = \mathsf{un}!U.U'$, which can never satisfy $\mathsf{un}(T)$. We only detail sub-cases (1) and (4), as sub-case (3) proceeds similarly.

1. By assumption, $P = x\langle u\rangle.P$ and $\neg\mathsf{un}(T)$. Moreover, since the judgment $\Gamma_2 \vdash P$ can only be obtained with Rule (T:Out), it must be the case that $T = q!U.U'$. Thus, by inversion on Rule (T:Out), (1) $\Gamma_2 = \Delta_1 \circ \Delta_2 \circ \Delta_3$, (2) $\Delta_1 \vdash x : q!U.U'$, (3) $\Delta_2 \vdash u : U$, and (4) $\Delta_3 + x : U' \vdash P$. We distinguish two further cases depending on whether $q = \mathtt{lin}$ or $q = \mathsf{un}$. We assume the latter, as the former is similar. By (2), it must be the case that $x : T \in \Delta_1$. Moreover, by inversion on (2) and (3), we have that $\mathsf{un}(\Delta_1')$ holds with (5) $\Delta_1' = \Delta_1 \setminus x : T$, and that (6) $\mathsf{un}(\Delta_2)$ holds. By Lem. 3.10(1), we also have that $x : T \notin \Delta_2$ and $x : T \notin \Delta_3$. Moreover, by Lem. 3.8(1), $\Delta_3 = \Gamma_2$, which implies that $\Delta_3 + x : U' = \Gamma_2, x : U'$. By applying the IH, $\Gamma_1 \circ (\Gamma_2, v : U') \vdash Q\{{}^{v}\!/x\}$. We then distinguish two further cases depending on whether $\mathsf{un}(U)$ or $\neg\mathsf{un}(U)$. In both cases we proceed similarly: we add all the missing hypothesis applying Lem. 3.9 and conclude by reapplying Rule (T:Out).

4. By assumption, $P = y\langle x\rangle.P$ and $\mathsf{un}(T)$. Moreover, since the judgment $\Gamma_2 \vdash P$ can only be obtained with Rule (T:Out), it must be the case that $T = q!U.U'$. We distinguish cases depending on whether $\mathsf{un}(U)$ or $\neg\mathsf{un}(U)$ are true. Both cases proceed similarly, so we only detail the latter. If $\mathsf{un}(U)$ holds, we have that (1) $\Gamma_2 = \Delta_1 \circ \Delta_2 \circ \Delta_3$, (2) $\Delta_1 \vdash x : q!U.U'$, (3) $\Delta_2 \vdash x : U$, and (4) $\Delta_3 + y : U' \vdash P$, and $x : U \in \Delta_1$, $x : U \in \Delta_2$, and $x : U \in \Delta_3$. Following a similar line of reasoning as the one above, we can conclude that $\Delta_3 = \Gamma_2$, and that $\Delta_1 = \Delta_2$. Moreover, by assumption $\Gamma_1 \vdash v : U$ and by IH, we then have that $\Gamma_1 \circ (\Gamma_2, v : U, y : U') \vdash P'\{{}^{v}\!/x\}$. We then distinguish two further cases depending on whether $\mathsf{un}(U')$ or $\neg\mathsf{un}(U')$. In both cases we conclude similarly as above.

□

Theorem 3.13 (Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \longrightarrow^* Q$ *then* $\Gamma \vdash Q$.

Proof (see Page 71). By induction on the reduction $P \longrightarrow^* Q$ with a case analysis on the last applied rule. The most interesting ones are for Rules $\lfloor$Com$\rfloor$ and $\lfloor$Rep$\rfloor$, as the others proceed similarly (or in the case of Rule $\lfloor$Str$\rfloor$, by the IH and Lem. 3.11). We show them both:

Case $\lfloor\textsc{Com}\rfloor$**:**

(1)	$P = (\boldsymbol{\nu} xy)(x\langle v\rangle.Q_1 \mid y(z).Q_2 \mid Q_3)$	(Assumption)
(2)	$P \longrightarrow (\boldsymbol{\nu} xy)(Q_1 \mid Q_2\{v/z\} \mid Q_3)$	(Assumption)
(3)	$\Gamma \vdash P$	(Assumption)
(4)	$\Gamma, x : T, y : \overline{T} \vdash x\langle v\rangle.Q_1 \mid y(z).Q_2 \mid Q_3$	(Inv. on Rule (T:Res), (1))
(5)	$\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$	(Inv. on Rule (T:Par), (3))
(6)	$\Gamma_1, x : T \vdash x\langle v\rangle.Q_1$	(Inv. on Rule (T:Par), (3))
(7)	$\Gamma_2, y : \overline{T} \vdash y(z).Q_2$	(Inv. on Rule (T:Par), (3))
(8)	$\Gamma_3 \vdash Q_3$	(Inv. on Rule (T:Par), (3))
(9)	$\Gamma_1 = \Gamma_1' \circ \Gamma_1'' \circ \Gamma_1'''$	(Inv. on Rule (T:Out), (6))
(10)	$\Gamma_1', x : T \vdash x : \mathsf{un}!U.U'$	(Inv. on Rule (T:Out), (6))
(11)	$\Gamma_1'' \vdash v : U$	(Inv. on Rule (T:Out), (6))
(12)	$\Gamma_1''' + x : U' \vdash Q_1$	(Inv. on Rule (T:Out), (6))
(13)	$\mathsf{un}(\Gamma_1') \wedge \mathsf{un}(\Gamma_1'')$ hold	(Inv. on Val. Rules, (10), (11))
(14)	$\Gamma_1''' = \Gamma_1$	(Lem. 3.8 to (13), (12))
(15)	$\Gamma_1 + x : U' \vdash Q_1$	((14), (12))
(16)	$(\Gamma_2, z : U') + y : \overline{U'} \vdash Q_2$	(Derived similarly to (15))

We now distinguish cases depending the nature of T and $\overline{T}$. Below we consider all the possible combinations:

$\neg\mathsf{un}(T) \wedge \neg\mathsf{un}(\overline{T})$**:** In this case, we have that (a) $\Gamma_1, x : U' \vdash Q_1$, by (15) and the definition of $+$; (b) $\Gamma_2, z : U, y : \overline{U'} \vdash Q_2$, by (16) and the definition of $+$; (c) $\Gamma_3 \vdash Q_3$, by (8), and (d) $\Gamma_1'' \vdash v : U$, by (11). By Lem. 3.12 applied to (d) and (b), we have (e) $\Gamma_2, y : \overline{U'} \vdash Q_2$, and we can finish the proof by applying rules (T:Par), (T:Par), (T:Res).

$\neg\mathsf{un}(T) \wedge \mathsf{un}(\overline{T})$**:** In this case, we have that (a) $\Gamma_1, x : U' \vdash Q_1$, by (15) and the definition of $+$; $(\Gamma_2, z : U') + y : \overline{U'} \vdash Q_2$ and $y : \mathsf{un}?U.\overline{U'} \in \Gamma_2$, by (16) and Definition of $\mathsf{un}(\cdot)$. Thus, $\overline{U'} = \mathsf{un}?U.\overline{U'}$ and thus, (b) $\Gamma_2, z : U', y : \overline{T} \vdash Q_2$. Similarly as above, we also have (c) $\Gamma_3 \vdash Q_3$, by (8); and (d) $\Gamma_1'' \vdash v : U$, by (11). By Lem. 3.9 on (a), we have (e) $\Gamma_1, x : U', y : \overline{T} \vdash Q_1$, and we can conclude by applying Lem. 3.12 and rules (T:Par), (T:Par), (T:Res) as above.

Other Cases: Notice that cases (i) $\mathsf{un}(T) \wedge \mathsf{un}(\overline{T})$ and (ii) $\mathsf{un}(T) \wedge \neg\mathsf{un}(\overline{T})$ are not possible, because $T = \mathsf{un}!U.U'$. Therefore $\mathtt{out}(T) = \mathtt{tt}$, and so our definition of $\mathsf{un}(T)$ does not hold.

Case $\lfloor\textsc{Rep}\rfloor$**:** Assuming that $P = (\boldsymbol{\nu} xy)(x\langle v\rangle.Q_1 \mid *\, y(z).Q_2 \mid Q_3)$, we proceed similarly as above. By inversion on Rule (T:Res):

$$\Gamma, x : T, y : \overline{T} \vdash x\langle v\rangle.Q_1 \mid *\, y(z).Q_2 \mid Q_3)$$

with $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$. Following a similar derivation as above, we can conclude that $T = q!U.U'$, and $\overline{T} = q?U.\overline{U'}$. Then, we can deduce that:

(a) $\Gamma_1 = \Gamma_1' \circ \Gamma_1'' \circ \Gamma_1'''$;

(b) $\Gamma_1''' + x : U' \vdash Q_1$ by inversion Rule (T:Out);

(c) $\Gamma_1'', x : T \vdash x : q!U.U'$ by inversion Rule (T:Out);

(d) $\mathsf{un}(\Gamma_1'')$ by inversion Rule (T:VAR);
(e) $\Gamma' \vdash v : U$ by inversion Rule (T:OUT);
(f) $\mathsf{un}(\Gamma_1')$ by inversion Rule (T:VAR) (or (T:BOOL));
(g) $(\Gamma_2, y : \overline{T}, z : U') + y : \overline{U'} \vdash Q_2$ by inversion Rule (T:RIN);
(h) $\mathsf{un}(\Gamma_2, y : \overline{T})$ by inversion Rule (T:RIN);
(i) $\Gamma_3 \vdash Q_3$ by inversion Rule (T:PAR).

By (h), we have that $\mathsf{un}(\overline{T})$ holds, which implies that $\neg\mathtt{out}(\overline{T})$ holds, and that $\overline{T} = U'$; hence T is a recursive type. Moreover, by Lem. 3.7(1b), we have that $\mathsf{un}(T)$ does not hold. Hence, we have that: $x : T \notin \Gamma_1'''$, $x : T \notin \Gamma_1'$. Then, by Lem. 3.8(1) to (a), (d), and (f), we have that $\Gamma_1''' = \Gamma_1$. Then, by applying Lem. 3.12 and Lem. 3.9 to (e) and (c), we have that $\Gamma_2, y : \overline{T} \circ \Gamma_1' \circ \Gamma_1'' \vdash Q_2\{v/z\}$. We then apply Lem. 3.9 to (i) to obtain $(\Gamma_3, y : \overline{T}) \circ \Gamma_1' \circ \Gamma_1'' \vdash Q_3$. Notice that we also know, by Lem. 3.9, that $\Gamma_1', \Gamma_1'' \circ (\Gamma_2, y : \overline{T}) \vdash *\, y(z).Q_2$. We then apply Rule (T:PAR) (which is applicable, since $\mathsf{un}(\overline{T})$ holds) to the previous hypotheses to obtain:

$$\Gamma, x : T, y : \overline{T} \vdash Q_1 \mid Q_2 \mid *\, y(z).Q_2 \mid Q_3$$

We then conclude by applying Rule (T:RES).

□

Theorem 3.15 (Type Safety). *If $\vdash P$ then P is well-formed.*

Proof (see Page 71). For the sake of contradiction, assume that $\vdash P$ and that P is not well-formed (i.e., it does not satisfy any of the items in Def. 3.14). We show that for every item that is not satisfied we reach a contradiction. If $\vdash P$, then there exists a derivation $\vdash (\boldsymbol{\nu} x_1 \ldots x_n y_1 \ldots y_n)(Q_1 \mid Q_2 \mid R)$, with $n \geq 1$ and therefore, by inversion applied n times, there exists an environment $x_1 : T_n, \ldots, x_n : T_n, y_1 : \overline{T}_1, \ldots, y_n : \overline{T}_n = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash Q_1 \mid Q_2 \mid R$. We now show that if $Q_1 \mid Q_2 \mid R$ does not satisfy any item in Def. 3.14, we reach a contradiction:

1. *If $Q_1 = v?\,(Q_1) : (Q_2)$ then $v \in \{\mathtt{tt}, \mathtt{ff}\}$:* Assume that does not satisfy this condition. Then, the derivation $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash Q_1 \mid Q_2 \mid R$ is not possible as Rule (T:IF) requires $v : \mathtt{bool}$.

2. *There does not exist Q_1 and Q_2 such that they are both prefixed with the same variable:* Assume there exists Q_1 and Q_2 that are prefixed in the same variable. Then there are two cases: (i) if the prefixes are of different nature, we reach a contradiction, since it is not possible for $x : T$ and $x : T'$ with $T \neq T'$ to appear in a typing derivation in the same environment; (ii) if the prefixes are of the same nature, we just need to notice that our splitting operation does not allow for session types that satisfy $\mathtt{out}(\cdot)$ to be shared among environments. Thus, only unrestricted input and branching types can be shared.

3. *If Q_1 is prefixed on $x_1 \in \widetilde{x}$ and Q_2 is prefixed on $y_1 \in \widetilde{y}$ then $Q_1 \mid Q_2$ is a redex:* Suppose that this does not hold (i.e., $Q_1 \mid Q_2$ is not a redex). Then, the typing derivation is not possible, since Rule (T:RES) requires the types of two covariables to be dual, thus reaching a contradiction.

□

A.2 Proofs for $\pi_{\mathsf{R}}^{\lightning}$

Lemma 3.21 (Properties of Typing Environments). *Let $\Gamma = \Gamma_1 \circ \Gamma_2$. Then all of the following hold:*

1. *$\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$.*
2. *Suppose that $x : qp \in \Gamma$. Then, either $x : qp \in \Gamma_1$ and $x \notin dom(\Gamma_2)$ or $x : qp \in \Gamma_2$ and $x \notin dom(\Gamma_1)$.*
3. *$\Gamma = \Gamma_2 \circ \Gamma_1$.*
4. *If $\Gamma_1 = \Delta_1 \circ \Delta_2$ then $\Delta = \Delta_2 \circ \Gamma_2$ and $\Gamma = \Delta_1 \circ \Delta$.*

Proof (see Page 75). Every item is proven by induction on the structure of Γ and by using the definition of splitting and predicates $\mathsf{un}(\cdot)$ and $\mathtt{lin}(\cdot)$ appropriately:

1. The base case is $\Gamma = \emptyset$. Then, $\Gamma_1 = \emptyset$ and $\Gamma_2 = \emptyset$. Moreover, $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2) = \emptyset$. For the inductive step, consider $\Gamma = \Gamma', x : T$. There are two subcases, depending on whether $\mathsf{un}(T)$ or $\mathtt{lin}(T)$. In the former case, we have that $\Gamma', x : T = \Gamma_1 \circ \Gamma_2$, and by Def. 2.16, $x : T \in \Gamma_1$ and $x : T \in \Gamma_2$. By IH, $\Gamma' = \Gamma'_1 \circ \Gamma'_2$ and $\mathcal{U}(\Gamma') = \mathcal{U}(\Gamma'_1) = \mathcal{U}(\Gamma'_2)$. Since $\mathsf{un}(T)$, then $x : T \in \mathcal{U}(\Gamma', x : T)$, $x : T \in \mathcal{U}(\Gamma_1)$, and $x : T\mathcal{U}(\Gamma_2)$. Thus, $\mathcal{U}(\Gamma) = \mathcal{U}(\Gamma_1) = \mathcal{U}(\Gamma_2)$. The latter case is immediate by IH since $x : T \notin \mathcal{U}(\Gamma)$.
2. The base case is $\Gamma = \emptyset$. It is immediate as the empty environment does not contain any elements. For the inductive step, assume $\Gamma = \Gamma, x : qp$. There are two cases: (1) $q = \mathtt{lin}$ and (2) $q = \mathsf{un}$. Both cases are immediate by Def. 2.16 since $\mathsf{un}(q)$ is false for both qualifiers.
3. Immediate by commutativity of the ',' for environments.
4. Immediate by associativity of the '$\circ$' operation (cf. Def. 2.16).

$\square$

Lemma 3.25 (Substitution). *If $\Gamma_1 \vdash v : T$ and $\Gamma_2, x : T \vdash P$ then $\Gamma \vdash P\{v/x\}$, with $\Gamma = \Gamma_1 \circ \Gamma_2$.*

Proof (see Page 76). By induction on the structure of P. There are nine cases. The most interesting case is whenever $P = *\,y(z).Q$ and $y = x$ (the case whenever $y \neq x$ is immediate by IH). We detail this case below. By assumption, (1) $\Gamma_1 \vdash v : T$ and (2) $\Gamma_2, x : T \vdash P = *\,x(y).Q$. By inversion on Rule (T:VAR) and (1), we have that $\mathsf{un}(\Gamma_1)$ holds. Moreover, by inversion on Rule (T:RIN), we have that: (1) $\mathsf{un}(\Gamma_2)$ holds, (2) $x : T \vdash x : \mathsf{un}?T'.U$, and (3) $\Gamma_2, z : T' \vdash Q$. By IH, we have that (4) $\Gamma_1 \circ (\Gamma_2, z : T') \vdash Q\{v/x\} = Q$ (notice that $x \notin \mathsf{fv}_\pi(Q)$). Moreover, we have that (5) $v : T \vdash v : T$ by applying Rule (T:VAR), and we have that (6) $\mathsf{un}(\Gamma_1 \setminus v : T)$ holds, as no qualified pre-typed satisfies $\mathsf{un}(\cdot)$. Then, we apply Rule (T:RIN) to (4), (5), and (6) to conclude the proof. $\square$

Theorem 3.26 (Subject Reduction). *If $\Gamma \vdash P$ and $P \longrightarrow^* Q$ then $\Gamma \vdash Q$.*

Proof (*see Page 76*). By induction on the reduction $P \longrightarrow^* Q$ with a case analysis on the last applied rule. The most interesting ones are for Rules $\lfloor \text{Com} \rfloor$ and $\lfloor \text{Rep} \rfloor$. We only show the case for Rule $\lfloor \text{Rep} \rfloor$ as the case for Rule $\lfloor \text{Com} \rfloor$ can be obtained similarly.

Case $\lfloor$Rep$\rfloor$: Assume that $P = (\boldsymbol{\nu} xy)(x\langle v\rangle.Q_1 \mid *\, y(z).Q_2 \mid Q_3)$By inversion on Rule (T:Res), $\Gamma, x : T, y : \overline{T} \vdash x\langle v\rangle.Q_1 \mid *\, y(z).Q_2 \mid Q_3)$, with $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$. Considering Def. 3.18 and Lem. 3.21 we can conclude that $T = q!U.U'$, and $\overline{T} = q?U.\overline{U'}$. Then, we can deduce that:

(a) $\Gamma_1 = \Gamma_1' \circ \Gamma_1'' \circ \Gamma_1'''$;

(b) $\Gamma_1''' + x : U' \vdash Q_1$ by inversion Rule (T:Out);

(c) $\Gamma_1'', x : T \vdash x : q!U.U'$ by inversion Rule (T:Out);

(d) $\mathsf{un}(\Gamma_1'')$ by inversion Rule (T:Var);

(e) $\Gamma' \vdash v : U$ by inversion Rule (T:Out);

(f) $\mathsf{un}(\Gamma_1')$ by inversion Rule (T:Var) (or (T:Bool));

(g) $(\Gamma_2, z : U) \vdash Q_2$ by inversion Rule (T:RIn);

(h) $\mathsf{un}(\Gamma_2)$ by inversion Rule (T:RIn);

(i) $\Gamma_3 \vdash Q_3$ by inversion Rule (T:Par).

Then, qualified pre-types cannot be shared (i.e., $\mathsf{un}(T)$ and $\mathsf{un}(\overline{T})$ are false), we have that: $x : T \notin \Gamma_1'''$, $x : T \notin \Gamma_1'$. Then, by Lem. 3.21(1) to (a), (d), and (f), we have that $\Gamma_1''' = \Gamma_1$. Then, by applying Lem. 3.25 and Lem. 3.22 to (e) and (c), we have that $\Gamma_2 \circ \Gamma_1' \circ \Gamma_1'' \vdash Q_2\{{}^{v}\!/\!{}_{z}\}$. We then apply Lem. 3.22 to (i) to obtain $\Gamma_3 \circ \Gamma_1' \circ \Gamma_1'' \vdash Q_3$. Notice that Lem. 3.22 ensures that $\Gamma_1', \Gamma_1'' \circ (\Gamma_2, y : \overline{T}) \vdash *\, y(z).Q_2$. We then apply Rule (T:Par) to the previous hypotheses to obtain:

$$\Gamma, x : T, y : \overline{T} \vdash Q_1 \mid Q_2 \mid *\, y(z).Q_2 \mid Q_3$$

We then conclude by applying Rule (T:Res).

□

Theorem 3.28 (Type Safety). *If $\vdash P$ then P is well-formed.*

Proof (*see Page 76*). For the sake of contradiction, assume that $\vdash P$ and that P is not well-formed (i.e., it does not satisfy any of the items in Def. 3.27). We show that for every item that is not satisfied we reach a contradiction. If $\vdash P$, then there exists a derivation $\vdash (\boldsymbol{\nu} x_1 \ldots x_n y_1 \ldots y_n)(Q_1 \mid Q_2 \mid R)$, with $n \geq 1$ and therefore, by inversion applied n times, there exists an environment $x_1 : T_n, \ldots, x_n : T_n, y_1 : \overline{T}_1, \ldots, y_n : \overline{T}_n = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash Q_1 \mid Q_2 \mid R$. We now show that if $Q_1 \mid Q_2 \mid R$ does not satisfy any item in Def. 3.27, we reach a contradiction:

1. *If $Q_1 = v?\,(Q_1) : (Q_2)$ then $v \in \{\mathtt{tt}, \mathtt{ff}\}$:* Assume that does not satisfy this condition. Then, the derivation $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash Q_1 \mid Q_2 \mid R$ is not possible as Rule (T:If) requires $v : \mathtt{bool}$.

2. *There does not exist processes P and Q that are prefixed on the same variable:* Assume there exists Q_1 and Q_2 that are prefixed in the same variable. Then there are two cases: (i) if the prefixes are of different nature, we reach a contradiction, since it is not possible for $x : T$ and $x : T'$ with $T \neq T'$ to appear in a typing derivation in the same environment; (ii) if the prefixes are of the same nature, we just need to notice that our splitting operation does not allow for any qualified pre-type to be shared among environments, leading to a contradiction.

3. *If Q_1 is prefixed on $x_1 \in \widetilde{x}$ and Q_2 is prefixed on $y_1 \in \widetilde{y}$ then $Q_1 \mid Q_2$ is a redex:* Suppose that this does not hold (i.e., $Q_1 \mid Q_2$ is not a redex). Then, the typing derivation is not possible, since Rule (T:Res) requires the types of two co-variables to be dual, thus reaching a contradiction.

□

Lemma 3.35 (Semantic Correspondence). *For every well-typed $\pi_{\mathsf{R}}^{\lightning}$ program P the following holds:*

1. *If $P \hookrightarrow\!\!\!\!\!\rightarrow Q$ then $P \longrightarrow^* Q$.*

2. *$P \longrightarrow^* Q$ then there exists Q' such that $P \hookrightarrow\!\!\!\!\!\rightarrow^* Q'$ and $Q \longrightarrow^* Q'$.*

Proof (*see Page 78*). We prove each item.

1. By induction on the number of parallel sub-processes of P.

2. By induction on the length k_0 of the sequence of reductions $P \longrightarrow^* Q$. The base case is immediate, since $P \longrightarrow^* P$ implies that $P \hookrightarrow\!\!\!\!\!\rightarrow^* P$. For the inductive step, we state the IH:

   IH: If $P \longrightarrow^* Q_0 \longrightarrow Q$ with $P \longrightarrow^* Q_0$ in $k \leq k_0$ steps. Then, there exists Q_0' such that $P \hookrightarrow\!\!\!\!\!\rightarrow^* Q_0'$ and $Q_0 \longrightarrow^* Q_0'$.

   We shall use the marked arrow introduced in page 78 (i.e., $\xrightarrow{\bullet}$). Let us enumerate our assumptions: (1) $\vdash P$ and (2) $P \longrightarrow^* Q$. We then proceed by induction on the length k_0 of the reduction $P \longrightarrow^* Q$. The base case is whenever $P \longrightarrow^* P$ and is immediate, as $P \hookrightarrow\!\!\!\!\!\rightarrow^* P$. For the inductive step, we consider the IH above. Then, we need to prove that the property holds for for $Q_0 \xrightarrow{\bullet} Q$. By Cor. 3.34:

   $$P = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ldots \mid P_n)$$

   with $n \geq 1$ and every P_i, $1 \leq i \leq n$ a pre-redex or conditional. By IH, $P \longrightarrow^* Q_0$. Moreover, by Thm. 3.26 applied to $P \longrightarrow^* Q_0$ and Assumption (1), $\vdash Q_0$. Next, by Thm. 3.28 applied to Q_0, Q_0 is well-formed (cf. Def. 3.27). Finally, by Cor. 3.34:

   $$Q_0 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1' \mid \ldots \mid P_m')$$

   with $m \geq 1$ and every P_i', $1 \leq i \leq m$ a pre-redex or conditional.

   Next, by IH, $Q_0 \longrightarrow^* Q_0'$, thus, we distinguish two cases: (1) whenever reduction $Q_0 \xrightarrow{\bullet} Q$ is in the sequence $Q_0 \longrightarrow^* Q_0'$ (i.e., $Q_0 \xrightarrow{\bullet}^* Q_0'$) and (2) whenever reduction $Q_0 \xrightarrow{\bullet} Q$ is not in the sequence $Q_0 \longrightarrow^* Q_0'$.

Case (1): Let $Q' = Q'_0$. Then, the case follows immediately by IH: If $Q' = Q'_0$, then by IH, $P \lhook\joinrel\twoheadrightarrow^* Q'$ and $Q_0 \xrightarrow{\bullet}{}^* Q'$. Moreover, by Rule $\lfloor$Big-Step$\rfloor$ in Def. 3.31 and since it must be the case that $Q_0 \xrightarrow{\bullet} Q$ is an outermost reduction (as it is included in $Q_0 \xrightarrow{\bullet}{}^* Q'$), it is possible to rearrange the sequence as $Q_0 \xrightarrow{\bullet} Q \longrightarrow^* Q'$, finishing the proof.

Case (2): We analyze the nature of reduction $Q_0 \xrightarrow{\bullet} Q$, distinguishing two cases: (1) when $Q_0 \xrightarrow{\bullet} Q$ comes from a conditional or (2) when $Q_0 \xrightarrow{\bullet} Q$ comes from a synchronization. Case (1) assumes that there exists P'_j, $1 \leq j \leq m$ such that:

$$Q = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid P''_j \mid \ldots \mid P'_m) \tag{1}$$

Case (2) assumes that there exist P_j and P_p, $1 \leq j, p \leq m$, $j \neq p$ such that:

$$Q = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid P''_j \mid \ldots \mid P''_p \mid \ldots \mid P'_m) \tag{2}$$

We detail Case (1) as the other proceeds similarly:

Case (1) Assume $Q = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid P''_j \mid \ldots \mid P'_m)$. Also, assume that $Q_0 \xrightarrow{\bullet} Q$. Then, by IH, there exists Q'_0 such that:

$$P \lhook\joinrel\twoheadrightarrow^* Q'_0 \wedge Q_0 \longrightarrow^* Q'_0$$

By Rule $\lfloor$Big-Step$\rfloor$ in Def. 3.31:

$$Q'_0 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(R'_1 \mid \ldots \mid R'_r \mid R_{r+1} \mid \ldots \mid R_s)$$

such that $R'_1, \ldots, R'_r$ are π processes and $R_{r+1}, \ldots, R_s$ are pre-redexes or conditional processes that were not in an outermost reduction. Next, by Cor. 3.34 and renaming the R_i processes to S_i $(r+1 \leq i \leq s)$ we have:

$$Q'_0 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(S_1 \mid \ldots \mid S_t \mid S_{r+1} \mid \ldots \mid S_s)$$

with $t \geq r$, $s \geq 1$ and every S_i, $1 \leq i \leq t$ is a pre-redex or conditional process.
Notice that the fact that reduction $Q_0 \xrightarrow{\bullet} Q$ is not included in $Q_0 \longrightarrow^* Q'_0$ together with the IH (i.e., $P \lhook\joinrel\twoheadrightarrow^* Q'_0$) imply that reduction $Q_0 \xrightarrow{\bullet} Q$ is not an outermost reduction in any of the processes of the sequence $P \lhook\joinrel\twoheadrightarrow^* Q'_0$. Therefore, there must exist an S_j $(j \in \{1, \ldots, t, r+1, \ldots, s\})$ such that:

$$Q'_0 \xrightarrow{\bullet} (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(S_1 \mid \ldots \mid S_t \mid \ldots \mid S'_j \mid \ldots \mid S_{r+1} \mid \ldots \mid S_s) = Q''_0$$

Then, applying Rule $\lfloor$Big-Step$\rfloor$ to Q'_0 we know there exist processes $S_1, \ldots, S_j, \ldots, S_a$ (with $a \leq s$) and processes $S_{a+1}, \ldots, S_b$ (with $b \leq s$) such that:

$$Q'_0 \lhook\joinrel\twoheadrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(S'_1 \mid \ldots \mid S'_j \mid \ldots \mid S'_a \mid S_{a+1} \mid \ldots \mid S_b) = Q' \tag{A.1}$$

By Lem. 3.35(1), $Q'_0 \longrightarrow^* Q'$. Furthermore, by Def. 3.31, $Q_0 \xrightarrow{\bullet} Q$ must be included in $Q'_0 \longrightarrow^* Q'$; thus, $Q'_0 \xrightarrow{\bullet}{}^* Q'$. Moreover, $Q_0 \longrightarrow^*$

Q_0' by IH and therefore, $Q_0 \longrightarrow^* Q_0' \overset{\bullet}{\longrightarrow}^* Q'$. Next, observe that $Q_0 \overset{\bullet}{\longrightarrow} Q$ and $Q_0' \overset{\bullet}{\longrightarrow} Q_0''$ implies that $Q_0 \overset{\bullet}{\longrightarrow} Q \longrightarrow^* Q'$, which implies that $Q \longrightarrow^* Q'$. Finally, by IH, we have that $P \hookrightarrow\!\!\!\!\rightarrow^* Q_0'$ and by (A.1) above, $P \hookrightarrow\!\!\!\!\rightarrow^* Q_0' \hookrightarrow\!\!\!\!\rightarrow Q'$ and thus, $P \hookrightarrow\!\!\!\!\rightarrow^* Q'$, finishing the proof.

□

A.3 Proofs for π_{E}

Lemma 3.37 (Subject Congruence for π_{E}). *If $\Phi \vdash_{\mathsf{N}} N$ and $N \equiv_{\mathsf{S}} M$ then $\Phi \vdash_{\mathsf{N}} M$.*

Proof (see Page 82). Using a case analysis on all the rules for $\equiv_{\mathsf{S}}$. We show four cases as all the other proceed similarly:

Case $N \mid \mathbf{0} \equiv_{\mathsf{S}} N$**:** We show both directions:

$\Rightarrow$) We proceed as follows:

(1) $\Phi \vdash_{\mathsf{N}} N \mid \mathbf{0}$ by Assumption.

(2) $\Phi \vdash_{\mathsf{N}} N$ by Inversion on Rule (T:NPar).

$\Leftarrow$) We proceed as follows:

(1) $\Phi \vdash_{\mathsf{N}} N$ by Assumption.

(2) $\Phi \vdash_{\mathsf{N}} \mathbf{0}$ by Rule (T:NNil).

(3) $\Phi \vdash_{\mathsf{N}} N \mid \mathbf{0}$ by (T:NPar) on (1) and (2).

Case $N \mid M \equiv_{\mathsf{S}} M \mid N$**:** We prove one direction, as the other proceeds in the same way:

$\Rightarrow$): We proceed as follows:

(1) $\Phi \vdash_{\mathsf{N}} N \mid M$ by Assumption.

(2) $\Phi \vdash_{\mathsf{N}} N$ by Inversion on Rule (T:NPar) and (1).

(3) $\Phi \vdash_{\mathsf{N}} M$ by Inversion on Rule (T:NPar) and (1).

(4) $\Phi \vdash_{\mathsf{N}} M \mid N$ by applyinh Rule (T:NPar) to (3) and (2).

Case $(\boldsymbol{\nu} xy)\mathbf{0} \equiv_{\mathsf{S}} \mathbf{0}$**:** We show both directions:

$\Rightarrow$) We proceed as follows:

(a) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)\mathbf{0}$ by Assumption.

(b) $\Phi \vdash_{\mathsf{N}} \mathbf{0}$ by applying Rule (T:NNil) to (1).

$\Leftarrow$) (1) $\Phi \vdash_{\mathsf{N}} \mathbf{0}$ (Assumption).

(2) $x : \mathsf{end}, y : \mathsf{end} \vdash \mathbf{0}$ by the application of Lem. 3.9 and Rule (T:Nil) from Fig. 2.3.

(3) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)\mathbf{0}$ by (T:Sess) and (2).

Case $(\boldsymbol{\nu} xy)P \mid (\boldsymbol{\nu} wz)Q \equiv_{\mathsf{S}} (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(P \mid Q)$**:** We show one side, as the other proceeds similarly:

$\Rightarrow$) We proceed as follows. Notice that $x, y \notin \mathsf{fv}_\pi(Q)$ and $w, z \notin \mathsf{fv}_\pi(P)$:

(1) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)P \mid (\boldsymbol{\nu} wz)Q$ by Assumption.
(2) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)P$ by Inversion on Rule (T:NPar) and (1)).
(3) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} wz)Q$ by Inversion on Rule (T:NPar) and (1).
(4) $x : T_1, y : \overline{T_1} \vdash P$ by Inversion on (T:Sess) and (2).
(5) $w : T_2, w : \overline{T_2} \vdash Q$ by Inversion on (T:Sess) and (3).
(6) $\vdash (\boldsymbol{\nu} xy)P$ by the application of Rule (T:Res) from Fig. 2.3 to (4).
(7) $\vdash (\boldsymbol{\nu} wz)Q$ by the application of Rule (T:Res) from Fig. 2.3 to (5).
(8) $\vdash (\boldsymbol{\nu} xy)P \mid (\boldsymbol{\nu} wz)Q$ by the application of Rule (T:Par) from Fig. 2.3 to (6) and (7).
(9) $\vdash (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(P \mid Q)$ by Lem. 3.11 and $\equiv_{\mathsf{S}}$ in π_{E}.
(10) $x : T_1, y : \overline{T_1} \vdash (\boldsymbol{\nu} wz)(P \mid Q)$ by Inversion on Rule (T:Res) from Fig. 2.3 and (9).
(11) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)(\boldsymbol{\nu} wz)(P \mid Q)$ by the application of Rule (T:Sess) to (10).

□

Theorem 3.38 (Subject Reduction for π_{E}). *If $\Phi, \Gamma \vdash_{\mathsf{N}} N$ and $N \longrightarrow_{\mathsf{N}}^{*} N'$ then $\Phi, \Gamma \vdash_{\mathsf{N}} N'$.*

Proof (see Page 83). By induction on k, the length of the reduction, followed by a case analysis on the last applied rule. Notice that Case $\lfloor$NStr$\rfloor$, concludes by IH and Lem. 3.37.

Base Case: Immediate, as it is the case where $k = 0$. Thus, $N \longrightarrow_{\mathsf{N}}^{*} N$.

Inductive Step: We proceed by a case analysis on the last applied rule:

Case Rule $\lfloor$SEstR$\rfloor$:

(1) $N = \underbrace{\left[\overline{a}^{m}\langle x\rangle.P_1\right]^{n}}_{M_1} \mid \underbrace{\left[* \, a^{\rho}(y).P_2\right]^{m}}_{M_2}$ by Assumption.
(2) $N' = (\boldsymbol{\nu} xy)(P_1 \mid P_2) \mid \left[* \, a^{\rho}(y).P_2\right]^{m}$ (Assumption).
(3) $\Phi \vdash_{\mathsf{N}} N$ by Assumption.
(4) $\Phi \vdash_{\mathsf{N}} M_1$ by (1),(3), and Inversion on Rule (T:NPar) in Fig. 3.4.
(5) $\Phi \vdash_{\mathsf{N}} M_2$ by (1),(3), Inversion on Rule (T:NPar) in Fig. 3.4.
(6) $\Phi \vdash_{\mathsf{N}} a : \langle T\rangle$ by (4) and Inversion on Rule (T:Req) in Fig. 3.4.
(7) $x : \overline{T} \vdash P_1$ by (4) and Inversion on Rule (T:Req) in Fig. 3.4.
(8) $\Phi \vdash_{\mathsf{N}} a : \langle T\rangle$ by (5) and Inversion on Rule (T:RAcc) in Fig. 3.4.
(9) $y : T \vdash P_2$ by (5) and Inversion on Rule (T:RAcc) in Fig. 3.4.
(10) $x : \overline{T}, y : T \vdash P_1 \mid P_2$ by the application of Rule (T:Par) in Fig. 2.3 to (7) and (9).
(11) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)(P_1 \mid P_2)$ by the application of Rule (T:Sess) in Fig. 3.4 to (10).
(12) $\Phi \vdash_{\mathsf{N}} \left[* \, a^{\rho}(y).P_2\right]^{m}$ by applying Rule (T:RAcc) to (8) and (9).
(13) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)(P_1 \mid P_2) \mid \left[* \, a^{\rho}(y).P_2\right]^{m}$ by applying Rule (T:NPar) to (11) and (12).

Case Rule $\lfloor$NPar$\rfloor$**:** Follows from the IH.

Case Rule $\lfloor$SRed$\rfloor$**:** Follows by inversion on Rule (T:Sess), Thm. 3.13 and applying Rule (T:Sess).

□

Lemma 3.41 (Type Safety for Runtime Networks). *If* $N \equiv_{\mathsf{S}} (\boldsymbol{\nu} xy)P \mid M$ *and* $\Phi \vdash_{\mathsf{N}} N$ *then* $(\boldsymbol{\nu} xy)P$ *is well-formed (cf. Def. 3.14).*

Proof (see Page 84). The proof can be derived as follows from the assumptions:

(1) $N \equiv_{\mathsf{S}} (\boldsymbol{\nu} xy)P \mid M$ by Assumption.

(2) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)P \mid M$ by Assumption.

(3) $\Phi \vdash_{\mathsf{N}} (\boldsymbol{\nu} xy)P$ by Inversion on (2) and Rule (T:NPar).

(4) $\Phi \vdash_{\mathsf{N}} M$ by Inversion on (2) and Rule (T:NPar).

(5) $x : T, y : \overline{T} \vdash P$ by Inversion on (3) and Rule (T:sess).

(6) $\vdash P$ by applying Rule (T:Res) to (5).

(7) P is well-formed by applying Thm. 3.15 to (6).

□

A.4 Proofs for aπ

Theorem 3.67 (Subject Congruence for aπ). *If* $\Gamma \vdash_{\Sigma} P \triangleright \Delta$ *and* $P \equiv_{\mathsf{A}} Q$ *then* $\Gamma \vdash_{\Sigma} Q \triangleright \Delta$.

Proof (see Page 97). By a case analysis on the rules of $\equiv_{\mathsf{A}}$ (cf. Def. 3.44). For each case we need to prove both directions:

(1) $P \mid \mathbf{0} \equiv_{\mathsf{A}} P$:

$\Rightarrow$): By assumption and inversion on Rule (T:QConc), $\Gamma \vdash_{\Sigma} P \triangleright \Delta_1$ and $\Gamma \vdash_{\emptyset} \mathbf{0} \triangleright \Delta_2$. By applying inversion twice, we have that $\mathsf{end}(\Delta_2)$ holds, Thus, we can conclude by adding Δ_2 to Δ_1, by applying Lem. 3.61.

$\Leftarrow$): $\Delta' = \emptyset$ is a terminated linear environment and therefore, by Rule (T:Nil), $\Gamma \vdash_{\emptyset} \mathbf{0} \triangleright \emptyset$. Finally, we can apply Rule (T:QConc) with the assumption to conclude the proof.

(2) $P \mid Q \equiv_{\mathsf{A}} Q \mid P$: Both directions are immediate by commutativity of $\asymp$.

(3) $P \equiv_{\mathsf{A}} Q$ if $P \equiv_{\alpha} Q$: Immediate by applying Lem. 3.63.

(4) $(P \mid Q) \mid R \equiv_{\mathsf{A}} P \mid (Q \mid R)$: Both directions are immediate by associativity of $\asymp$.

(5) $(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P \equiv_{\mathsf{A}} (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P$: Both directions are immediate by the commutativity of Δ.

(6) $(\boldsymbol{\nu}x)\mathbf{0} \equiv_{\mathsf{A}} \mathbf{0}$:

$\Rightarrow$): By assumption, (1) $\Gamma \vdash_{\Sigma} (\boldsymbol{\nu}x)\mathbf{0} \triangleright \Delta$. Since $(\boldsymbol{\nu}x)\mathbf{0}$ is a$\pi^\star$ process then $\Sigma = \emptyset$ and (1) is deduced with Rule (T:Res). Thus, by inversion, (2) $\Gamma \vdash_{\Sigma,x,\overline{x}} \mathbf{0} \triangleright \Delta, x : \mathsf{end}, \overline{x} : \mathsf{end}$. By the syntactic structure of $\mathbf{0}$, it can be deduced that this judgment was derived with Rule (T:WkS). Thus, by inversion, (3) $\Gamma \vdash_{\Sigma} \mathbf{0} \triangleright \Delta, x : \mathsf{end}, \overline{x} : \mathsf{end}$. Then, it can be deduced $\Sigma = \emptyset$. Thus, by inversion, (4) $\Gamma \vdash \mathbf{0} \triangleright \Delta, x : \mathsf{end}, \overline{x} : \mathsf{end}$. Finally, we have that this judgment must have been deduced with Rule (T:Nil) and hence, by inversion: $\Delta, x : \mathsf{end}, \overline{x} : \mathsf{end}$. Then, by Lem. 3.62, we can conclude that $\Gamma \vdash \mathbf{0} \triangleright \Delta$. Finally, we can apply Rule (T:WkS) twice to obtain $\Gamma \vdash_{x,\overline{x}} \mathbf{0} \triangleright \Delta$, finishing the proof.

$\Leftarrow$): By assumption, (1) $\Gamma \vdash_{\Sigma} (\boldsymbol{\nu}x)\mathbf{0} \triangleright \Delta$ and by applying inversion twice we get that $\Sigma = \emptyset$ and that $\mathsf{end}(\Delta)$ holds. Then, we can apply Lem. 3.62 to obtain $\Gamma \vdash \mathbf{0} \triangleright \Delta, x : \mathsf{end}, \overline{x} : \mathsf{end}$. Next, using Rule (T:WkS) twice, we deduce $\Gamma \vdash_{x,\overline{x}} \mathbf{0} \triangleright \Delta, x : \mathsf{end}, \overline{x} : \mathsf{end}$. Finally, we can apply Rule (T:Res) to conclude $\Gamma \vdash_{\Sigma} (\boldsymbol{\nu}x)\mathbf{0} \triangleright \Delta$, finishing the proof.

(7) $(\boldsymbol{\nu}k)(k[i:\epsilon; o:\epsilon] \mid \overline{k}[i:\epsilon; o:\epsilon]) \equiv_{\mathsf{A}} \mathbf{0}$:

$\Rightarrow$): By inversion on Rule (T:Res), $\Gamma \vdash_{\Sigma,k,\overline{k}} k[i:\epsilon; o:\epsilon] \mid \overline{k}[i:\epsilon; o:\epsilon] \triangleright \Delta, x : T \asymp M_1, \overline{x} : T' \asymp M_2$ with $T \perp T'$. This judgment was deduced with Rule (T:QConc). Thus, by inversion: (1) $\Gamma \vdash_{\Sigma_1,k} k[i:\epsilon; o:\epsilon] \triangleright \Delta_1, x : M_1, \overline{x} : T'$ and (2) $\Gamma \vdash_{\Sigma_2,\overline{k}} \overline{s}[i:\epsilon; o:\epsilon] \triangleright \Delta_2, x : T, \overline{x} : M_2$. Then, (1) and (2) could only have been deduced wit Rules (T:OQEnd) or (T:IQEnd). Let us assume that both were deduced with Rule (T:OQEnd). All the other cases proceed similarly. By inversion on Rule (T:OQEnd) on (1) and (2), we have that $\Sigma_1 = \Sigma_2 = \emptyset$, which implies that $\Sigma = \emptyset$. Also, $\mathsf{end}(\Delta_1, x : M_1, \overline{x} : T')$ and $\mathsf{end}(\Delta_2, x : T)$ hold. Thus, $T = T' = \mathsf{end}$. We can then conclude by applying Rule (T:Nil) with $\Delta = \Delta_1, \Delta_2, x : T, \overline{x} : T'$.

$\Leftarrow$): We proceed similarly to the case $\mathbf{0} \equiv_{\mathsf{A}} (\boldsymbol{\nu}x)\mathbf{0}$. The only caveat is that we have to consider that $\mathsf{end} \asymp \emptyset = \mathsf{end}$, to obtain the desired typing.

(8) $(\boldsymbol{\nu}x)P \mid Q \equiv_{\mathsf{A}} (\boldsymbol{\nu}x)(P \mid Q)$ if $x \notin \mathsf{fv}_{\pi}(Q)$: Both directions are immediate by inversion and considering that x cannot appear in the linear environment typing Q.

□

Theorem 3.69 (Subject Reduction for aπ). *If $\Gamma \vdash_{\Sigma} P \triangleright \Delta$ with Δ well-configured (cf. Def. 3.58) and $P \longrightarrow_{\mathsf{A}}^{*} Q$ then $\Gamma \vdash_{\Sigma} Q \triangleright \Delta'$ with $\Delta \rightharpoonup^{*} \Delta'$ and Δ' is well-configured.*

Proof (see Page 97). By induction on the length of reduction $P \longrightarrow_{\mathsf{A}}^{*} Q$. The base case is immediate. We show the inductive step:

Rule $\lfloor$Send$\rfloor$: In here we distinguish two cases depending on whether v is a boolean value or another channel (delegation).

Case $v : \mathtt{bool}$ By assumption, (1) $\Gamma \vdash_{\Sigma} k\langle v\rangle.Q \mid k[i:\widetilde{h_1}, o:\widetilde{h_2}] \triangleright \Delta', k :!U.(T \asymp M)$ and $\Delta = \Delta', x :!U.T$ is well-configured. By using inversion on Rule (T:QConc), we have that (2) $\Gamma \vdash_{\Sigma_1} k\langle v\rangle.Q \triangleright \Delta'_1, k :!U.T$ and (3) $\Gamma \vdash_{\Sigma_2,k}$

$k[i : \widetilde{h_1}, o : \widetilde{h_2}] \triangleright \Delta'_2, k : M$. By inversion applied on (2) (Rule (T:Send)), we have that (4) $\Sigma_1 = \emptyset$, (5) $\Gamma \vdash v : U$ and (6) $\Gamma \vdash Q \triangleright \Delta'_1, T$. Similarly, for (3), we distinguish cases depending on whether $\widetilde{h_2} = \epsilon$ or not. Without loss of generality, we only consider the case when $\widetilde{h_2} \neq \epsilon$. The other case is similar.

In this case, we have that $\widetilde{h_2} = v_1 \cdot \ldots \cdot v_n$, $n \geq 1$. Thus, by applying inversion on Rule (T:QOut) we have that (7) $\Sigma_2 = \emptyset$ and (8) $\mathsf{end}(\Delta'_2)$ holds. Then, we have that $k\langle v\rangle.Q \mid k[i : \widetilde{h_1}, o : \widetilde{h_2}] \longrightarrow_{\mathsf{A}} Q \mid k[i : \widetilde{h_1}, o : \widetilde{h_2} \cdot v]$. By applying Rule (T:QOut) with (5) we have that (9) $\Gamma \vdash_{\Sigma_2, k} k[i : \widetilde{h_1}, o : \widetilde{h_2} \cdot v] \triangleright \Delta'_2, k : M'$, where $M' = m_1 \ldots . m_n .!U$ and for every $i \in \{1, \ldots, n\}$, $m_i = \oplus l_j$ for some j or $m_i = !U'$ for some U'.

Then, by applying Rule (T:QConc) to (6) and (9) yields:

$$\Gamma \vdash_{\Sigma} k\langle v\rangle.Q \mid k[i : \widetilde{h_1}, o : \widetilde{h_2}] \triangleright \Delta', x : T \asymp M'$$

Now, notice that since Δ is well-configured (Def. 3.58), $\Delta = \Delta_0, k : !U.T, \overline{k} : ?U.\overline{T}$. Moreover, the merging $T \asymp M'$ reconstructs the type $!U.(T \asymp M)$ given that session types are reconstructed in the reverse order, as how the messages were sent. Thus, $\Delta', x : T \asymp M' = \Delta$ and thus, $\Delta \rightharpoonup^* \Delta$, finishing the proof.

Case $v : T$**:** If the value is a channel (an hence its type is a session type), then the proof proceeds mostly as the case above, but we assume that the judgment is deduced using Rule (T:Del), rather than (T:Send).

Rule $\lfloor$**Sel**$\rfloor$**:** Similar to the case above, but we make use of Rule (T:Sub) to find the correct typing for the selection operator.

Rule $\lfloor$**Com**$\rfloor$**:** This case proceeds immediately form the IH. Notice that the environment Δ reduces with Rule $\lfloor$E:Send$\rfloor$. Namely, when we move an output message from an output queue to the input queue of the complementary endpoint, the message is consumed and thus, the environment changes.

Rule $\lfloor$**Recv**$\rfloor$**:** Similarly to $\lfloor$Send$\rfloor$. We distinguish two cases depending on whether the input process was typed using Rule (T:Rcv) or (T:DRcv), in both cases we proceed as in the case for the semantics Rule $\lfloor$Send$\rfloor$.

Other Cases: Case $\lfloor$Bra$\rfloor$ is similar to the case above. Cases for rules $\lfloor$IfT$\rfloor$, $\lfloor$IfF$\rfloor$, $\lfloor$Rec$\rfloor$, $\lfloor$Res$\rfloor$, and $\lfloor$Par$\rfloor$ are immediate by the IH. Finally, the case for Rule $\lfloor$Str$\rfloor$ is immediate by IH and Thm. 3.67.

□

Lemma 3.85 (Single-Step Semantic Correspondence). *Let P be a well-formed aπ program. Then, the following holds:*

1. *If $P \ggg\!\rightarrow Q$ then $P \longrightarrow^*_{\mathsf{A}} Q$.*
2. *If $P \longrightarrow_{\mathsf{A}} Q$ then there exists R such that $P \ggg\!\rightarrow R$ and $Q \longrightarrow^*_{\mathsf{A}} R$.*

Proof (see Page 104). We prove both numerals.

1. We derive the proof from the definitions. Assume that $1 \leq n \leq m$:

(1)	$P = C[P_1 \mid \ldots \mid P_n, Q_1 \mid \ldots \mid Q_m]$	(Thm. 3.69, Lem. 3.73)
(2)	$P \ggg\!\!\rightarrow Q = \mathsf{unm}(Q_0)$, for some Q_0	(Lem. 3.83, (1))
(3)	$P \rightarrowtail C[P_1 \mid \ldots \mid P_n, Q'_1 \mid \ldots \mid Q'_m]$	(Inversion on (2))
(4)	$Q_1 \mid \ldots \mid Q_m \rightharpoondown^* Q'_1 \mid \ldots \mid Q'_m$	(Inversion on (3))
(5)	$C[P_1 \mid \ldots \mid P_n, Q'_1 \mid \ldots \mid Q'_m] \rightharpoonup^* Q_0 \not\rightharpoonup$	(Inversion on (2))
(6)	$Q_1 \mid \ldots \mid Q_m \longrightarrow_{\mathsf{A}}^* Q'_1 \mid \ldots \mid Q'_m$ using Rule $\lfloor\textsc{Com}\rfloor$	(Lem. 3.84(1), (4))
(7)	$C[P_1 \mid \ldots \mid P_n, Q'_1 \mid \ldots \mid Q'_m]$ $\longrightarrow_{\mathsf{A}}^* \mathsf{unm}(Q_0) = Q$	(Lem. 3.84(2), (5))
(8)	$P \longrightarrow_{\mathsf{A}}^* C[P_1 \mid \ldots \mid P_n, Q'_1 \mid \ldots \mid Q'_m]$ $\longrightarrow_{\mathsf{A}}^* \mathsf{unm}(Q_0) = Q$	(Fig. 3.5, (6), (7))

2. By Thm. 3.69 and Lem. 3.73, $P = C[P_1 \mid \ldots \mid P_n, Q_1 \mid \ldots \mid Q_m]$ with $1 \leq n \leq m$. By assumption, $P \longrightarrow_{\mathsf{A}} Q$, given the shape of the process we deduce this reduction can only take place using Rule $\lfloor\textsc{Res}\rfloor$. By inversion, we have that $P_1 \mid \ldots \mid P_n \mid Q_1 \mid \ldots \mid Q_m \longrightarrow_{\mathsf{A}} Q'$, which is only possible by applying Rule $\lfloor\textsc{Par}\rfloor$. We then distinguish two cases, depending on which processes reduce:

   $\exists i \in \{1, \ldots, n\}, j \in \{n+1, \ldots, m\}$ such that $P_i \mid Q_j \longrightarrow_{\mathsf{A}} P'_i \mid Q'_j$: If this is the case, then the reduction must have occurred with either Rule $\lfloor\textsc{Send}\rfloor$, Rule $\lfloor\textsc{Sel}\rfloor$, Rule $\lfloor\textsc{IfT}\rfloor$, Rule $\lfloor\textsc{IfF}\rfloor$, Rule $\lfloor\textsc{Recv}\rfloor$, or Rule $\lfloor\textsc{Bra}\rfloor$. We only show the case for Rule $\lfloor\textsc{Send}\rfloor$, as the other cases are similar:

   Rule $\lfloor\textsc{Send}\rfloor$: In this case, $P_i = x\langle v\rangle.P'_i$ and $Q_j = x[i : \widetilde{h_1}, o : \widetilde{h_2}]$. We distinguish cases for (1) $\widetilde{h_2} = \epsilon$ or (2) $\widetilde{h_2} = v_1 \cdot \widetilde{h'_2}$. We show the latter, as it is the most interesting one. By assumption,

$$
\begin{aligned}
P &= C[P_1 \mid \ldots \mid x\langle v\rangle.P'_i \mid \ldots \mid P_n, Q_1 \mid \ldots \mid \\
&\qquad\qquad x[i : \widetilde{h_1}, o : v_1 \cdot \widetilde{h'_2}] \mid \ldots \mid Q_m] \\
&\longrightarrow_{\mathsf{A}} C[P_1 \mid \ldots \mid P'_i \mid \ldots \mid P_n, Q_1 \mid \ldots \mid \\
&\qquad\qquad x[i : \widetilde{h_1}, o : v_1 \cdot \widetilde{h'_2} \cdot v] \mid \ldots \mid Q_m] \\
&= Q
\end{aligned}
$$

   We now prove the existence of process R. Consider the big-step reduction $P \ggg\!\!\rightarrow R$. By Def. 3.81, process R is given by the big-step reduction sequence $P \rightarrowtail R_0 \rightharpoonup^* R'_0 \not\rightharpoonup$, where $R = \mathsf{unm}(R'_0)$. We analyze the first part of the sequence: $P \rightarrowtail R_0$. By Def. 3.78,

$$
\begin{aligned}
P \rightarrowtail C[P_1 \mid \ldots \mid x\langle v\rangle.P'_i \mid \ldots \mid P_n, Q'_1 \mid \ldots \mid \\
x[i : \widetilde{h'_1}, o : \widetilde{h'_2}] \mid \ldots \mid Q'_m] = R_0
\end{aligned}
$$

   as Rule $\lfloor\textsc{ComM}\rfloor$ is applied to allow synchronization between all the queues in P. In R_0, we distinguish two cases: (1) $\widetilde{h'_1} = \widetilde{h_1}$ or (2) $\widetilde{h'_1} = \widetilde{h} \cdot \widehat{v'}$. We only show the latter, as the other case is similar.

If $\widetilde{h'_1} = \widetilde{h} \cdot \widehat{v'}$ then, in the reduction sequence $R_0 \rightharpoonup^* R'_0 \not\rightharpoonup$, we have that:

$$R_0 \rightharpoonup^* C[P'_1 \mid \ldots \mid P''_i \mid \ldots \mid P'_n, Q''_1 \mid \ldots \mid x[i : \widetilde{h''_1}, o : \widetilde{h''_2}] \mid \ldots \mid Q'_m] = R'_0 \not\rightharpoonup$$

with $\widetilde{h''_1} = \widetilde{h_0} \cdot \widehat{v'}$, and $\widetilde{h''_2} = \widetilde{h'_2} \cdot v \cdot \widetilde{h'_0}$, for some $\widetilde{h_0}$ and some $\widetilde{h'_0}$.
We now have to prove that $Q \longrightarrow^*_{\mathsf{A}} R$. Consider the big-step reduction $Q \ggg\!\!\rightarrow S$. We will show that $S = R$ and use the result obtained in Numeral 1. of the proof to show the existence of the reduction sequence $Q \longrightarrow^*_{\mathsf{A}} R$. By Def. 3.81, we analyze the sequence $Q \rightarrowtail S_0 \rightharpoonup^* S'_0 \not\rightharpoonup$. As above:

$$Q \rightarrowtail C[P_1 \mid \ldots \mid P'_i \mid \ldots \mid P_n, Q_1 \mid \ldots \mid x[i : \widetilde{h'_1}, o : \widetilde{h''_2}] \mid \ldots \mid Q_m] = S_0$$

this follows from Thm. 3.70, which ensures the existence of a unique queue for endpoint $\overline{x}$.
A similar argument can be used to show that $S_0 \rightharpoonup^* S'_0 \not\rightharpoonup$. Namely, the big-step reduction $\rightharpoonup$ only allows k-processes to interact with their respective queue. Thus, since by Thm. 3.70 there is only one queue for endpoint x and only process P'_i implements the protocol for endpoint x, we can conclude that $S'_0 = R'_0$. Furthermore, by Lem. 3.85(1) (i.e, Numeral 1. in this statement), we have that $\mathtt{unm}(Q) \longrightarrow^*_{\mathsf{A}} \mathtt{unm}(R_0)$, and considering that $R = \mathtt{unm}(R_0)$ and that, by Def. 3.80, $\mathtt{unm}(Q) = Q$ (since Q contains no marks), we have shown that $Q \longrightarrow^*_{\mathsf{A}} R$, finishing the proof.

$\exists i, j \in \{n+1, \ldots, m\}$ **such that** $Q_i \mid Q_j \longrightarrow_{\mathsf{A}} Q'_i \mid Q'_j$**:** The reduction originated from the application of Rule $\lfloor\text{Com}\rfloor$. We prove the case by showing that if $P \longrightarrow_{\mathsf{A}} Q$, $P \ggg\!\!\rightarrow R$, and $Q \ggg\!\!\rightarrow S$ then $S = R$. This proceeds similarly as above.

□

A.5 Proofs for lcc$^{\mathsf{p}}$

Lemma 3.91 (Subject Congruence). *If $P \equiv Q$ and $\vdash_\diamond P$, then $\vdash_\diamond Q$.*

Proof (see Page 109). By a case analysis on $P \equiv Q$ (cf. Def. 2.28). Since congruences are symmetric, we need to prove for both $P \equiv Q$ and $Q \equiv P$. There are eight cases. The most interesting ones are for (SC$_1$:4) and (SC$_1$:7). We detail them below:

Case $!\,P \equiv P \parallel !\,P$**:** We distinguish two sub-cases corresponding to the direction of $\equiv$:

Sub-case ($\Rightarrow$):

Assumption $\quad !\,P \equiv P \parallel !\,P \quad (1)$

Assumption	$\vdash_\diamond \, !\,P$	(2)
(2), inversion on Rule (L:Repl)	$\vdash_\diamond P$	(3)
(2), (3), formation on Rule (L:Par)	$\vdash_\diamond P \parallel !\,P$	(4)

Sub-case ($\Leftarrow$):

Assumption	$!\,P \parallel P \equiv !\,P$	(1)
Assumption	$\vdash_\diamond \, !\,P \parallel P$	(2)
(2), inversion on Rule (L:Par)	$\vdash_\diamond \, !\,P$	(3)

Case $P \parallel \exists z.\, Q \equiv \exists z.\, (P \parallel Q)$: We distinguish two sub-cases corresponding to the direction of $\equiv$:

Sub-case ($\Rightarrow$):

Assumption	$P \parallel \exists z.\, Q \equiv \exists z.\, (P \parallel Q)$	(1)
Assumption	$\vdash_\diamond P \parallel \exists z.\, Q$	(2)
(1), inversion on Rule (ScL:7)	$z \notin \mathsf{fv}(P)$	(3)
(2), inversion on Rule (L:Par)	$\vdash_\diamond P$	(3)
(2), inversion on Rule (L:Par)	$\vdash_\diamond \exists z.\, Q$	(5)
(5), inversion on Rule (L:Local)	$\vdash_\diamond Q$	(6)
(4),(6), formation on Rule (L:Par)	$\vdash_\diamond P \parallel Q$	(7)
(7), formation on Rule (L:Local) using z	$\vdash_\diamond \exists z.\, (P \parallel Q)$	(8)

Sub-case ($\Leftarrow$):

Assumption	$\exists z.\, (P \parallel Q) \equiv P \parallel \exists z.\, Q$	(1)
Assumption	$\vdash_\diamond \exists z.\, (P \parallel Q)$	(2)
(1), inversion on Rule (ScL:7)	$z \notin \mathsf{fv}(P)$	(3)
(2), inversion on Rule (L:Local)	$\vdash_\diamond P \parallel Q$	(4)
(4), inversion on Rule (L:Par)	$\vdash_\diamond P$	(5)
(4), inversion on Rule (L:Par)	$\vdash_\diamond Q$	(6)
(6), formation on Rule (L:Local) using z	$\vdash_\diamond \exists z.\, Q$	(7)
(5),(7), formation on Rule (L:Par)	$\vdash_\diamond P \parallel \exists z.\, Q$	(8)

□

Theorem 3.93 (Subject Reduction). *If $P \xrightarrow{\alpha}_1 Q$ and $\vdash_\diamond P$ then $\vdash_\diamond Q$.*

Proof (see Page 110). By a case analysis on the transition rule applied (cf. Fig. 2.5). There are eight cases, where the most interesting one is the case for Rule $\lfloor$C:SyncLoc$\rfloor$, which is detailed below:

Case Rule $\lfloor$C:SyncLoc$\rfloor$:

Assumption	$\overline{c} \parallel \forall \widetilde{x}(d\,;\, e \rightarrow P) \xrightarrow{\tau}_1 \exists \widetilde{y}.\, (P\{\widetilde{t}/\widetilde{x}\} \parallel \overline{f})$	(1)

1, inversion on Rule ⌊C:SyncLoc⌋	$c \otimes d \vdash \exists \widetilde{y}.(e\{\widetilde{t}/\widetilde{x}\} \otimes f)$	(2)
(1), inversion on Rule ⌊C:SyncLoc⌋	$\widetilde{y} \cap \mathsf{fv}(c, d, e, P) = \emptyset$	(3)
(1), inversion on Rule ⌊C:SyncLoc⌋	$\mathbf{mgc}\big(c \otimes d, \exists \widetilde{y}.(e\{\widetilde{t}/\widetilde{x}\} \otimes f)\big)$	(4)
(1), inversion on Rule ⌊C:SyncLoc⌋	$c \otimes d \vdash \texttt{ff} \implies c \vdash \texttt{ff}$	(5)
Assumption	$\vdash_\diamond \overline{c} \parallel \forall \widetilde{x}(d\,;\, e \rightarrow P)$	(6)
(6), inversion on Rule (L:Par)	$\vdash_\diamond \overline{c}$	(7)
(6), inversion on Rule (L:Par)	$\vdash_\diamond \forall \widetilde{x}(d\,;\, e \rightarrow P)$	(8)
(8), inversion on Rule (L:Guard)	$\vdash_{\mathbf{A}} \forall \widetilde{x}(d\,;\, e \rightarrow P)$	(9)
(9), inversion on Rule (L:Abs)	$\vdash_\diamond P$	(10)
(9), inversion on Rule (L:Abs)	$\Delta; \Theta \vdash_\bullet e$	(11)
(9), inversion on Rule (L:Abs)	$\widetilde{x} \subseteq \Theta \setminus \mathsf{fv}(d)$	(12)
(2), deductive closure	$\exists \widetilde{y}.(e\{\widetilde{t}/\widetilde{x}\} \otimes f) \in \mathcal{C}$	(13)
(13), transitivity	$f \in \mathcal{C}$	(14)
(14), formation on Rule (L:Tell)	$\vdash_\diamond \overline{f}$	(15)
(4), Def. 2.29	$\widetilde{t}$ is a vector of terms	(16)
(10), (16), Prop. 3.92	$\vdash_\diamond P\{\widetilde{t}/\widetilde{x}\}$	(17)
(15), (17), formation on Rule (L:Par)	$\vdash_\diamond P\{\widetilde{t}/\widetilde{x}\} \parallel \overline{f}$	(18)
(18), formation on Rule (L:Local) with $\widetilde{y}$	$\vdash_\diamond \exists \widetilde{y}.\,(P\{\widetilde{t}/\widetilde{x}\} \parallel \overline{f})$	(19)

□

B
Chapter 4

B.1 Junk Processes

Lemma 4.15. *Let J be junk. Then: (1) $J \not\xrightarrow{\tau}_1$ (and) (2) there is no $c \in \mathcal{C}$ (cf. Def. 4.1) such that $J \parallel \overline{c} \xrightarrow{\tau}_1$.*

Proof (see Page 130). We prove each item individually.

(1) By induction on the structure of J. We show two cases:

Case $J = \forall \epsilon((b = \neg b) \to P)$**:** This case is immediate by inspecting the rules in Fig. 2.5; in particular, since Rule (C:Sync) cannot be applied $J \not\xrightarrow{\tau}_1$.

Case $J = J_1 \parallel J_2$**:** By IH, $J_1 \not\xrightarrow{\tau}_1$ and $J_2 \not\xrightarrow{\tau}_1$. We prove that a reduction $J_1 \parallel J_2 \xrightarrow{\tau}_1 J_1' \parallel J_2'$ cannot occur. By Def. 4.14, junk processes are either ask-guarded processes or $\overline{\texttt{tt}}$. To reduce, one of J_1 and J_2 must add a constraint to the store; two ask processes in parallel cannot reduce (cf. Fig. 2.5). Now, since $\overline{\texttt{tt}}$ does not add any information to the store, we have that $J_1 \parallel J_2 \not\xrightarrow{\tau}_1$.

(2) By induction on the structure of J. We detail three cases:

Case $J = \overline{\texttt{tt}}$**:** This case is immediate, as $\overline{\texttt{tt}} \parallel \overline{c} \not\xrightarrow{\tau}_1$, for any constraint $c \in \mathcal{C}$.

Case $J = \forall \epsilon((b = \neg b) \to P)$**:** Suppose, for the sake of contradiction, that there is a $c \in \mathcal{C}$ (cf. Def. 4.1) such that $J \parallel \overline{c} \xrightarrow{\tau}_1 S$, for some S. Then, by Def. 4.1, c must be composed of the predicates snd, rcv, sel, bra, $\{\cdot{:}\cdot\}$ (cf. Fig. 4.1), the multiplicative conjunction $\otimes$, replication and the existential quantifier. We now apply induction on the structure of c: there are five base cases (one for each predicate) and 3 inductive cases (one for each logical connective). We show only two representative cases:

Sub-case $c = \mathsf{snd}(x, v)$**:** This base case is immediate, as $\mathsf{snd}(x, v) \not\vdash (b = \neg b)$ using the Rules in Fig. 2.4. Thus, Rule (C:SYNC) is not applicable, therefore contradicting our assumption.

Sub-case $c = c_1 \otimes c_2$**:** By IH, $c_i \not\vdash (b = \neg b)$. Then, by the rules in Fig. 2.4, $c_1 \otimes c_2 \not\vdash (b = \neg b)$, leading to a contradiction as in the previous sub-case.

Case $J = J_1 \parallel J_2$**:** Then $J \parallel \overline{c} \not\xrightarrow{}_1$ follows immediately from the IH (which ensures $J_1 \parallel \overline{c} \not\xrightarrow{}_1$ and $J_2 \parallel \overline{c} \not\xrightarrow{}_1$) and Item (1).

$\square$

Lemma 4.16 (Junk Observables). *For every junk process J and every $\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}\mathcal{C}$-context $C[-]$, we have that:*

1. $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J) = \emptyset$ *(and)*
2. $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(C[J]) = \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(C[\overline{\mathtt{tt}}])$.

Proof (see Page 130). We prove each item separately:

1. By induction on the structure of J. We show the most representative 3 cases:

   Case $J = \forall\epsilon((b = \neg b) \to P)$**:** By Lem. 4.15(1), $J \not\xrightarrow{}_1$. Therefore, by Def. 2.30 $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J) = \emptyset$.

   Case $J = \overline{\mathtt{tt}}$**:** By Lem. 4.15(1), $J \not\xrightarrow{}_1$. Moreover, since $\mathtt{tt} \notin \mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}$, by Def. 2.30, $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J) = \emptyset$.

   Case $J = J_1 \parallel J_2$**:** By IH, $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J_i) = \emptyset, i \in \{1, 2\}$. By Lem. 4.15(1), $J \not\xrightarrow{}_1$ and therefore, $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J) = \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J_1) \cup \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J_2) = \emptyset$.

2. The proof is by induction on the structure of J, followed in each case by a case analysis on $C[-]$ (cf. Def. 2.32). All cases follow from the definitions; we detail two representative cases:

   Case $J = \forall\epsilon((b = \neg b) \to P_1)$**:** We apply a case analysis on context $C[-]$. We will show only two sub-cases, as the third one is symmetrical:

   Sub-case $C[-] = \exists\widetilde{x}. -$**:** This case follows from Lem. 4.15(1,2).

   Sub-case $C[-] = - \parallel P_2$**:** Then $C[J] = \forall\epsilon((b = \neg b) \to P_1) \parallel P_2$. By Lem. 4.15(1,2), $\forall\epsilon((b = \neg b) \to P_1) \not\xrightarrow{}_1$ and there is no $c \in \mathcal{C}$ (cf. Def. 4.1) such that $J \parallel \overline{c} \xrightarrow{\tau}_1$. As such, a reduction $C[J] \xRightarrow{\tau}_1 C'[J']$ is not possible (i.e., J cannot reduce in the context). Therefore, by Def. 2.30 and Item (1), we have that:

   $$\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(C[J]) = \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(J) \cup \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(P_2) = \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(P_2)$$

   Following a similar analysis, and using Lem. 4.15(1,2), Item(1) and Def. 2.30, $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(C[\overline{\mathtt{tt}}]) = \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(\overline{\mathtt{tt}}) \cup \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(P_2) = \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(P_2)$. We conclude $\mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(C[\overline{\mathtt{tt}}]) = \mathcal{O}^{\mathcal{D}_{\pi^{\lightning}_{\mathsf{OR}}}}(C[J])$, as wanted.

Case $J = J_1 \parallel J_2$**:** We apply a case analysis on context $C[-]$. We will show only two sub-cases, as the third one is symmetrical:

Sub-case $C[-] = \exists \widetilde{x}.(\,-\,)$**:** This case follows from Lem. 4.15(1,2).

Sub-case $C[-] = - \parallel P_2$**:** By IH, we have $\mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(C[J_1]) = \mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(C[\overline{\mathtt{tt}}])$ and $\mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(C[J_2]) = \mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(C[\overline{\mathtt{tt}}])$. Also, by Lem. 4.15(1), $J_1 \parallel J_2 \not\xrightarrow{\tau}_1$ and by Lem. 4.15(2), there is no $c \in \mathcal{C}$ (cf. Def. 4.1) such that $J \parallel \overline{c} \xrightarrow{\tau}_1$. Hence, by Def. 2.30 and Item (1), $\mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(C[J_1 \parallel J_2]) = \mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(\overline{\mathtt{tt}})$, as wanted.

□

Lemma 4.17 (Junk Behavior). *For every junk J, every $\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}\mathcal{C}$-context $C[-]$, and every process P, we have $C[P \parallel J] \approx_1^{\pi^{\natural}_{\mathsf{OR}}} C[P]$.*

Proof (see Page 131). By coinduction, i.e., by exhibiting a weak o-barbed bisimulation containing the pair $(C[P \parallel J], C[P])$. To achieve we recall Def. 2.33. Then, we define a symmetric relation $\mathcal{R}$ such that $(R, Q) \in \mathcal{R}$ implies:

1. $\mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(R) = \mathcal{O}^{\mathcal{D}_{\pi^{\natural}_{\mathsf{OR}}}}(Q)$ (and),
2. whenever $R \xrightarrow{\tau}_1 R'$ there exists Q' such that $Q \xRightarrow{\tau}_1 Q'$ and $R'\mathcal{R}Q'$.

We define

$$\begin{aligned}\mathcal{R} = &\{(R, Q) \mid C[P \parallel J] \xrightarrow{\tau}{}_1^{n} R \wedge C[P] \xrightarrow{\tau}{}_1^{n} Q, n \geq 0\} \\ \cup\ &\{(Q, R) \mid C[P \parallel J] \xrightarrow{\tau}{}_1^{n} R \wedge C[P] \xrightarrow{\tau}{}_1^{n} Q, n \geq 0\} \qquad (1)\end{aligned}$$

Notice that this relation is symmetric by definition. Moreover, Item (1) is immediately satisfied by $\mathcal{R}$, thanks to Lem. 4.16(2).

As for Item (2), first notice that $(R, Q) \in \mathcal{R}$: we have $R = C[P \parallel J]$ and $Q = C[P]$ (with $n = 0$). Now suppose that $R \xrightarrow{\tau}_1 R'$; we show a matching transition $Q \xRightarrow{\tau}_1 Q'$ such that $R'\mathcal{R}Q'$. To this end, we use a case analysis on the reduction(s) possible from $C[P \parallel J]$. There are six possibilities:

(a) $C[P \parallel J] \xrightarrow{\tau}_1 C[P \parallel J']$ (i.e., an autonomous reduction from J);

(b) $C[P \parallel J] \xrightarrow{\tau}_1 C'[P \parallel J']$ (i.e., a reduction from the interplay of C and J);

(c) $C[P \parallel J] \xrightarrow{\tau}_1 C[P' \parallel J']$ (i.e., a reduction from the interplay of P and J);

(d) $C[P \parallel J] \xrightarrow{\tau}_1 C'[P \parallel J]$ (i.e., an autonomous reduction from C);

(e) $C[P \parallel J] \xrightarrow{\tau}_1 C'[P' \parallel J]$ (i.e., an interaction between C and P);

(f) $C[P \parallel J] \xrightarrow{\tau}_1 C[P' \parallel J]$ (i.e., an autonomous reduction from P).

Notice that Lem. 4.15(1,2) and Lem. 4.16(1,2) exclude cases (a)–(c). Thus, reductions for $C[J]$ will only be of the form (d)–(f). Clearly, this transition from R can be matched by Q as follows:

- $Q = C[P] \xrightarrow{\tau}_1 C'[P] = Q'$, with $(C'[P \parallel J], C'[P]) \in \mathcal{R}$ (or)
- $Q = C[P] \xrightarrow{\tau}_1 C'[P'] = Q'$, with $(C'[P' \parallel J], C'[P']) \in \mathcal{R}$ (or)
- $Q = C[P] \xrightarrow{\tau}_1 C[P'] = Q'$, with $(C[P' \parallel J], C[P']) \in \mathcal{R}$.

With these reductions, we conclude the proof for this case. The case when $Q \xrightarrow{\tau}_1 Q'$ is similar. □

Lemma 4.19 (Occurrences of Junk). *Let R be a redex (Def. 2.19).*

1. *If $R = x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$, with $j \in I$ then: $[\![(\boldsymbol{\nu} xy)R]\!] \xrightarrow{\tau}{}_1^3 \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![Q_j]\!] \parallel J\big)$, where $J = \prod_{i\in I'} \forall\epsilon(l_j = l_i \to [\![Q_i]\!])$, with $I' = I \setminus \{j\}$, and $\exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![Q_j]\!] \parallel J\big) \cong_1^{\pi^{\not{\iota}}_{\mathsf{OR}}} \exists x, y.\big(\,!\,\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![Q_j]\!]\big)$.*
2. *If $R = b?\,(P_1) : (P_2)$, $b \in \{\mathtt{tt}, \mathtt{ff}\}$, then $[\![R]\!] \xrightarrow{\tau}_1 [\![P_i]\!] \parallel J$, $i \in \{1, 2\}$ with $J = \forall\epsilon(b = \neg b \to [\![P_j]\!])$, $j \neq i$, and $[\![P_i]\!] \parallel J \cong_1^{\pi^{\not{\iota}}_{\mathsf{OR}}} [\![P_i]\!]$.*
3. *If $R = x\langle v\rangle.P \mid y(z).Q$ then $[\![(\boldsymbol{\nu} xy)R]\!] \xrightarrow{\tau}{}_1^2 \cong_1^{\pi^{\not{\iota}}_{\mathsf{OR}}} \exists x, y.\big([\![P]\!] \parallel [\![Q\{v/z\}]\!] \parallel J\big)$ with $J = \overline{\mathtt{tt}}$.*
4. *If $R = x\langle v\rangle.P \mid * \, y(z).Q$ then $[\![(\boldsymbol{\nu} xy)R]\!] \xrightarrow{\tau}{}_1^2 \cong_1^{\pi^{\not{\iota}}_{\mathsf{OR}}} \exists x, y.\big([\![P]\!] \parallel [\![Q\{v/z\}]\!] \parallel [\![*\, y(z).P]\!] \parallel J\big)$ with $J = \overline{\mathtt{tt}}$.*

Proof (*see Page 131*). Each item follows from the translation definition (cf. Def. 4.4 and Fig. 4.2). Items (1) and (2) refer to reductions that induce junk (no junk is generated in Items (3) and (4)); those cases rely on the definition of weak o-barbed congruence (cf. Def. 4.3) and Cor. 4.18.

1. Given $R = x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$ (with $j \in I$), by Def. 4.4, Fig. 4.2, and the operational semantics of `lcc` in Def. 2.5:

$$\begin{aligned}
[\![(\boldsymbol{\nu} xy)R]\!] &= \exists x, y.\big(!\overline{\{x{:}y\}} \parallel \overline{\mathsf{sel}(x, l_j)} \parallel \forall z\big(\mathsf{bra}(z, l_j) \otimes \{x{:}z\} \to [\![P]\!]\big) \\
&\qquad \forall l, w\big(\mathsf{sel}(w, l) \otimes \{w{:}x\} \to \overline{\mathsf{bra}(x, l)} \parallel \prod_{i\in I} \forall\epsilon(l = l_i \to [\![Q_i]\!])\big)\big) \\
&\xrightarrow{\tau}_1 \exists x, y.\big(!\overline{\{x{:}y\}} \parallel \forall z\big(\mathsf{bra}(z, l_j) \otimes \{x{:}z\} \to [\![P]\!]\big) \\
&\qquad \overline{\mathsf{bra}(x, l_j)} \parallel \prod_{i\in I} \forall\epsilon(l_j = l_i \to [\![Q_i]\!])\big) \\
&\xrightarrow{\tau}_1 \exists x, y.\big(!\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel \prod_{i\in I} \forall\epsilon(l_j = l_i \to [\![Q_i]\!])\big) \\
&\xrightarrow{\tau}_1 \exists x, y.\big(!\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![P_j]\!] \parallel \underbrace{\prod_{i\in I\setminus\{j\}} \forall\epsilon(l_j = l_i \to [\![Q_i]\!])}_{J}\,\big) \\
&\cong_1^{\pi^{\not{\iota}}_{\mathsf{OR}}} \exists x, y.\big(!\overline{\{x{:}y\}} \parallel [\![P]\!] \parallel [\![P_j]\!]\big)
\end{aligned}$$

where the last step is justified by Cor. 4.18.

2. Given that $R = b?\,(P_1):(P_2)$, we distinguish two cases: $b = \mathtt{tt}$ and $b = \mathtt{ff}$. We only detail the analysis when $R = \mathtt{tt}?\,(P_1):(P_2)$, as the other case is analogous. By the translation definition (cf. Def. 4.4 and Fig. 4.2), $[\![R]\!] = \forall\epsilon(\mathtt{tt} = \mathtt{tt} \to [\![P_1]\!]) \parallel \forall\epsilon(\mathtt{tt} = \mathtt{ff} \to [\![P_2]\!])$. Then, by the rules in Fig. 2.5, $[\![R]\!] \xrightarrow{\tau}_1 [\![P_1]\!] \parallel J$, with $J = \forall\epsilon(\mathtt{tt} = \mathtt{ff} \to [\![P_2]\!])$. By Cor. 4.18, we may conclude as follows:

$$[\![R]\!] \xrightarrow{\tau}_1 [\![P]\!] \parallel \forall\epsilon(\mathtt{tt} = \mathtt{ff} \to [\![Q]\!]) \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P]\!]$$

3. Given $R = x\langle v\rangle.P \mid y(z).Q$, by the translation definition (cf. Fig. 4.2), and the semantics in Fig. 2.5:

$$\begin{aligned}
[\![(\boldsymbol{\nu} xy)R]\!] &\equiv \exists x, y.\big(\overline{!\{x{:}y\} \otimes \mathsf{snd}(x,v)} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{x{:}z\} \to [\![P]\!]) \parallel \\
&\qquad\qquad \forall z, w(\mathsf{snd}(w,z) \otimes \{w{:}y\} \to \overline{\mathsf{rcv}(y,z)} \parallel [\![Q]\!])\big) \\
&\xrightarrow{\tau}_1 \exists x, y.\big(\overline{!\{x{:}y\}} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{x{:}z\} \to [\![P]\!]) \parallel \\
&\qquad\qquad \overline{\mathsf{rcv}(y,v)} \parallel [\![Q\{v,x/z,w\}]\!]\big) \\
&\equiv \exists x, y.\big(\overline{!\{x{:}y\} \otimes \mathsf{rcv}(y,v)} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{x{:}z\} \to [\![P]\!]) \parallel \\
&\qquad\qquad [\![Q\{v,x/z,w\}]\!]\big) \\
&\xrightarrow{\tau}_1 \exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P\{y/z\}]\!] \parallel [\![Q\{v,x/z,w\}]\!]\big) \\
&\cong_1^{\pi_{\mathsf{OR}}^{\lightning}} \exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P\{y/z\}]\!] \parallel [\![Q\{v,x/z,w\}]\!] \parallel \overline{\mathtt{tt}}\big)
\end{aligned}$$

Let $J = \overline{\mathtt{tt}}$. Finally, we conclude by Cor. 4.18.

4. When $R = x\langle v\rangle.Q \mid *\,y(z).P$, the proof follows the same reasoning as above.

□

B.2 Operational Completeness

Theorem 4.20 (Completeness for $[\![\cdot]\!]$). *Let $[\![\cdot]\!]$ be the translation in Def. 4.4. Also, let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program. Then, if $P \longrightarrow^* Q$ then $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$.*

Proof (see Page 131). By induction on the length of the reduction $\longrightarrow^*$, with a case analysis on the last applied rule. The base case is whenever $P \longrightarrow^0 P$, and it is trivially true since $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 [\![P]\!]$. For the inductive step, assume by IH, that $P \longrightarrow^* P_0 \longrightarrow Q$ and that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![P_0]\!]$. We then have to prove that $[\![P_0]\!] \overset{\tau}{\Longrightarrow}_1 \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$. There are nine cases since cases for Rules (Res), (Par) and (Str) are immediate by IH.

Rule $\lfloor$IfT$\rfloor$:

(1) $P_0 = \mathtt{tt}?\,(P'):(P'')$.

(2) By (1), $P_0 \longrightarrow P' = Q$.

(3) By Def. 4.4, $[\![P_0]\!] = \forall\epsilon(\mathtt{tt} = \mathtt{tt} \to [\![P']\!]) \parallel \forall\epsilon(\mathtt{tt} = \mathtt{ff} \to [\![P'']\!])$.

(4) By Rule (C:Sync) (cf. Fig. 2.5), with $c = \mathtt{tt}$ we have the following (note that $\Vdash \mathtt{tt} = \mathtt{tt}$):

$$[\![P_0]\!] \xrightarrow{\tau}_1 [\![P']\!] \parallel \forall \epsilon(\mathtt{tt} = \mathtt{ff} \rightarrow [\![P'']\!]) = R$$

(5) By (4) note that the process $\forall \epsilon(\mathtt{tt} = \mathtt{ff} \rightarrow [\![P'']\!])$ is junk (cf. Def. 4.14). Then, by Cor. 4.18 $R \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$, which is what we wanted to prove.

Rule $\lfloor$IfF$\rfloor$**:** Analogous to the previous case.

Rule $\lfloor$Com$\rfloor$**:**

(1) $P_0 = (\boldsymbol{\nu} xy)(x\langle v\rangle.P' \mid y(z).P'' \mid S)$, with S collecting all the processes that may contain x and y. Notice that by typing, S can only contain (replicated) input processes on y.

(2) By (1) $P_0 \longrightarrow (\boldsymbol{\nu} xy)(P' \mid P''\{^{v}/_{z}\} \mid T) = Q$.

(3) By Fig. 4.2:

$$\begin{aligned}[\![P_0]\!] = \exists x, y.\big(\overline{!\{x{:}y\}} \parallel \overline{(\mathsf{snd}(x, v)} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\ \forall z_2, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y, z_2)} \parallel [\![P'']\!]) \parallel [\![S]\!]\big)\end{aligned}$$

(4) By Fig. 2.5, Def. 2.28:

$$\begin{aligned}[\![P_0]\!] &\equiv \exists x, y.\, ((\overline{!\{x{:}y\} \otimes \mathsf{snd}(x, v)} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\ &\qquad \forall z_2, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y, z_2)} \parallel [\![P'']\!]) \parallel [\![S]\!]) \\ &\longrightarrow_1 \exists x, y.\, ((\overline{!\{x{:}y\}} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \overline{\mathsf{rcv}(y, v)} \parallel \\ &\qquad [\![P''\{^{v, x}/_{z_2, w}\}]\!] \parallel [\![S]\!]) \\ &\equiv \exists x, y.\big(\overline{!\{x{:}y\} \otimes \mathsf{rcv}(y, v)} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\} \rightarrow [\![P']\!]) \parallel \\ &\qquad [\![P''\{^{v, x}/_{z_2, w}\}]\!] \parallel [\![T]\!]\big) \\ &\longrightarrow_1 \exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P'\{^{y}/_{z_1}\}]\!] \parallel [\![P''\{^{v, x}/_{z_2, w}\}]\!] \parallel [\![S]\!]\big)\end{aligned}$$

(5) By Fig. 4.2 we have that $w \notin \mathsf{fv}(P'')$ and $z_1 \notin \mathsf{fv}(P')$. Therefore: $\exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P'\{^{y}/_{z}\}]\!] \parallel [\![P''\{^{v, x}/_{z, w}\}]\!] \parallel [\![S]\!]\big) = \exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P''\{^{v}/_{z}\}]\!] \parallel [\![S]\!]\big) = [\![Q]\!]$

(6) Finally, by reflexivity of $\cong_1^{\pi_{\mathsf{OR}}^{\lightning}}$ (Def. 4.3), $[\![Q]\!] \cong_1^{\pi_{\mathsf{OR}}^{\lightning}} [\![Q]\!]$.

Rule $\lfloor$Repl$\rfloor$**:**

(1) Assume $P_0 = (\boldsymbol{\nu} xy)(x\langle v\rangle.P' \mid *\, y(z).P'' \mid S)$, with S collecting all the processes that may contain x and y. Notice that by typing, S can only contain (replicated) input processes on y.

(2) By (1) $P_0 \longrightarrow (\boldsymbol{\nu} xy)(P' \mid P''\{^{v}/_{z}\} \mid *\, y(z).P'' \mid S) = Q$ using Rule $\lfloor$Rep$\rfloor$.

(3) By definition of $[\![\cdot]\!]$:

$$\begin{aligned}
[\![P_0]\!] &= \exists x, y.\big(\overline{!\{x{:}y\}} \parallel \overline{(\mathsf{snd}(x, v)} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad !\forall z_2, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y, z_2)} \parallel [\![P'']\!]) \parallel [\![S]\!]\big) \\
&\equiv \exists x, y.\big(\overline{!\{x{:}y\}} \parallel \overline{(\mathsf{snd}(x, v)} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad \forall z_2, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y, z_2)} \parallel [\![P'']\!])) \parallel \\
&\qquad !\forall z_2, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y, z_2)} \parallel [\![P'']\!]) \parallel [\![S]\!]\big)
\end{aligned}$$

(4) Let $R = !\forall z_2, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y, z_2)} \parallel [\![P'']\!]))$. By using the rules of structural congruence and reduction of `lcc` the following transitions can be shown:

$$\begin{aligned}
[\![P_0]\!] &\equiv \exists x, y.\big(\overline{!\{x{:}y\} \otimes \mathsf{snd}(x, v)} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad \forall z_2, w(\mathsf{snd}(w, z_2) \otimes \{w{:}y\}) \rightarrow (\overline{\mathsf{rcv}(y, z_2)} \parallel [\![P'']\!])) \parallel \\
&\qquad R \parallel [\![S]\!]\big) \\
&\xrightarrow{\tau}_1 \exists x, y.\big(\overline{!\{x{:}y\}} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \overline{\mathsf{rcv}(y, v)} \parallel \\
&\qquad [\![P''\{v, x/z_2, w\}]\!]) \parallel R \parallel [\![S]\!]\big) \\
&\equiv \exists x, y.\big(\overline{!\{x{:}y\} \otimes \mathsf{rcv}(y, v)} \parallel \forall z_1((\mathsf{rcv}(z_1, v) \otimes \{x{:}z_1\}) \rightarrow [\![P']\!]) \parallel \\
&\qquad [\![P''\{v, x/z_2, w\}]\!] \parallel R \parallel [\![S]\!]\big) \\
&\xrightarrow{\tau}_1 \exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P'\{y/z_1\}]\!] \parallel [\![P''\{v, x/z_2, w\}]\!] \parallel R \parallel [\![S]\!]\big)
\end{aligned}$$

(5) As in Case $\lfloor$Com$\rfloor$, we have that

$$\begin{aligned}
&\exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P'\{y/z_1\}]\!] \parallel [\![P''\{v, x/z_2, w\}]\!] \parallel R \parallel [\![T]\!]\big) \\
&= \exists x, y.\big(\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P''\{v/z_2\}]\!] \parallel R \parallel [\![T]\!]\big)
\end{aligned}$$

since $w \notin \mathsf{fv}(P'')$ and $z_1 \notin \mathsf{fv}(P'')$, by Fig. 4.2.

(6) Finally, observe that:

$$\begin{aligned}
[\![Q]\!] &= [\![(\boldsymbol{\nu} xy)(P' \mid P''\{v/z\} \mid *y(z).P'' \mid T)]\!] \\
&= \exists x, y.\big((\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P''\{v/z_2\}]\!] \parallel R \parallel [\![T]\!])\big)
\end{aligned}$$

Rule $\lfloor$Sel$\rfloor$:

(1) Assume $P_0 = (\boldsymbol{\nu} xy)(x \triangleleft l_j.P' \mid x \triangleright \{l_i : P_i\}_{i \in I} \mid T)$. Notice that since P_0 is a well-formed program, typing implies that process $T \equiv \mathbf{0}$, since x, y cannot be shared. Thus, we do not consider T below.

(2) By (1) $P_0 \longrightarrow (\boldsymbol{\nu} xy)(P' \mid P_j) = Q$ using Rule $\lfloor$Sel$\rfloor$.

(3) By definition of $[\![\cdot]\!]$ (cf. Fig. 4.2):

$$\begin{aligned}
[\![P_0]\!] &= \exists x, y.\big(\overline{!\{x{:}y\}} \parallel \overline{\mathsf{sel}(x, l_j)} \parallel \forall z\big(\mathsf{bra}(z, l_j) \otimes \{x{:}z\} \rightarrow [\![P']\!]\big) \\
&\qquad \forall l, w\big(\mathsf{sel}(w, l) \otimes \{w{:}x\} \rightarrow \overline{\mathsf{bra}(x, l)} \parallel \prod_{i \in I} \forall \epsilon(l = l_i \rightarrow [\![P_i]\!])\big)\big)
\end{aligned}$$

(4) By using the semantics of lcc (cf. Def. 2.5) and Cor. 4.18, we obtain the following derivation

$$\begin{aligned}
[\![P_0]\!] &\xrightarrow{\tau}_1 \exists x,y.\big(\overline{!\{x{:}y\}} \parallel \forall z\big(\mathsf{bra}(z,l_j)\otimes\{x{:}z\}\rightarrow[\![P']\!]\big) \\
&\qquad \overline{\mathsf{bra}(x,l_j)} \parallel \prod_{i\in I}\forall\epsilon(l_j=l_i\rightarrow[\![P_i]\!])\big) \\
&\xrightarrow{\tau}_1 \exists x,y.\big(\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel \prod_{i\in I}\forall\epsilon(l_j=l_i\rightarrow[\![P_i]\!])\big) \\
&\xrightarrow{\tau}_1 \exists x,y.\big(\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P_j]\!] \parallel \underbrace{\prod_{i\in I\setminus\{j\}}\forall\epsilon(l_j=l_i\rightarrow[\![P_i]\!])}_{J}\big) \\
&\cong_1^{\pi^{\not{z}}_{\mathsf{OR}}} \exists x,y.\big(\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P_j]\!]\big)
\end{aligned}$$

(5) By definition of $[\![\cdot]\!]$ (cf. Fig. 4.2), $[\![Q]\!] = [\![(\boldsymbol{\nu}xy)(P' \mid P_j)]\!] = \exists x,y.\big(\overline{!\{x{:}y\}} \parallel [\![P']\!] \parallel [\![P_j]\!]\big)$.

(6) By (iv) and (v) we conclude the proof.

□

B.3 Invariants for Pre-Redexes and Redexes

Lemma 4.24 (Invariants of $[\![\cdot]\!]$ for Input-Like Pre-Redexes). *Let P be a pre-redex such that $\mathcal{I}^{\mathcal{D}^\star_{\pi^{\not{z}}_{\mathsf{OR}}}}([\![P]\!]) = \emptyset$. Then one of the following holds:*

1. *If $[\![P]\!] \parallel \overline{\mathsf{sel}(x,l_j)\otimes\{y{:}x\}} \xrightarrow{\tau}_1 S$ then $\mathsf{bra}(y,l_j) \in \mathcal{I}^{\mathcal{D}^\star_{\pi^{\not{z}}_{\mathsf{OR}}}}(S)$ and $P = y \triangleright \{l_i : P_i\}_{i\in I}$, with $j \in I$.*

2. *If $[\![P]\!] \parallel \overline{\mathsf{snd}(x,v)\otimes\{y{:}x\}} \xrightarrow{\tau}_1 S$ then $\mathsf{rcv}(y,v) \in \mathcal{I}^{\mathcal{D}^\star_{\pi^{\not{z}}_{\mathsf{OR}}}}(S)$ and $P = \diamond\, y(z).P_1$.*

Proof (see Page 136). By assumption, P is a pre-redex and $\mathcal{I}^{\mathcal{D}^\star_{\pi^{\not{z}}_{\mathsf{OR}}}}(P) = \emptyset$. Then, by Lem. 4.23(4), we have that $[\![P]\!] \not\longrightarrow_1$ and that $P = y(z).P_1$, $P = *\,y(z).P_1$ or $P = y \triangleright \{l_1 : P_i\}_{i\in I}$. We now apply a case analysis on each numeral in the statement:

Case $[\![P]\!] \parallel \overline{\mathsf{sel}(w,l)\otimes\{y{:}x\}} \xrightarrow{\tau}_1 S$: We observe the behavior of each possibilities for P in the presence of constraint $\mathsf{sel}(w,l_j)\otimes\{y{:}x\}$ for some l_j, following Fig. 4.2. First we observe:

$$\begin{aligned}
[\![y(z).P_1]\!] \parallel \overline{\mathsf{sel}(w,l)\otimes\{y{:}x\}} &= \forall z,w\big(\mathsf{snd}(w,z)\otimes\{w{:}y\}\rightarrow\overline{\mathsf{rcv}(y,z)} \parallel [\![P_1]\!]\big) \parallel \\
&\qquad \overline{\mathsf{sel}(x,l_j)\otimes\{y{:}x\}} \not\longrightarrow_1 \\
[\![*\,y(z).P_1]\!] \parallel \overline{\mathsf{sel}(w,l)\otimes\{y{:}x\}} &= !(\forall z,w\big(\mathsf{snd}(w,z)\otimes\{w{:}y\}\rightarrow\overline{\mathsf{rcv}(y,z)} \parallel [\![P_1]\!]\big)) \parallel \\
&\qquad \overline{\mathsf{sel}(x,l_j)\otimes\{y{:}x\}} \not\longrightarrow_1
\end{aligned}$$

In contrast, process $[\![y \triangleright \{l_1 : P_i\}_{i\in I}]\!] \parallel \overline{\mathsf{sel}(w,l) \otimes \{y{:}x\}}$ can reduce: from the semantics of `lcc` (cf. Fig. 2.5) and under the assumption that $j \in I$ for l_j, we have:

$$\forall l, w\big(\mathsf{sel}(w,l) \otimes \{w{:}y\} \to \overline{\mathsf{bra}(y,l)} \parallel \prod_{1\leq i\leq n} \forall\epsilon(l = l_i \to [\![P_i]\!])\big) \parallel \overline{\mathsf{sel}(x,l_j) \otimes \{y{:}x\}}$$

$$\xrightarrow{\tau}_1 \overline{\mathsf{bra}(y,l_j)} \parallel \prod_{1\leq i\leq n} \forall\epsilon(l = l_i \to [\![P_i]\!]) = S$$

Finally, by Def. 4.22: $\mathcal{I}^{\mathcal{D}^\star_{\pi^\lightning_{\mathsf{OR}}}}(S) = \{\mathsf{bra}(y,l_j)\} \cup \mathcal{I}^{\mathcal{D}^\star_{\pi^\lightning_{\mathsf{OR}}}}(\prod_{1\leq i\leq n} \forall\epsilon(l_j = l_i \to [\![P_i]\!]))$, thus concluding the proof.

Case $[\![P]\!] \parallel \overline{\mathsf{snd}(x,v) \otimes \{y{:}x\}} \xrightarrow{\tau}_1 S$**:** This case proceeds as above by noticing that a reduction into S is enabled only when $P = y(z).P_1$ or $P = *\,y(z).P_1$.

□

Lemma 4.27 (Invariants for Redexes and Intermediate Redexes). *Let R be a redex enabled by $\widetilde{x}, \widetilde{y}$, such that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'$. Then one of the following holds:*

1. *If $R \equiv_{\mathsf{S}} v?\,(P_1) : (P_2)$ and $v \in \{\mathtt{tt}, \mathtt{ff}\}$, then $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R]\!] \xrightarrow{\tau}_1 \cong_1^{\pi^\lightning_{\mathsf{OR}}} (\boldsymbol{\nu}\widetilde{x}\widetilde{y})[\![P_i]\!]$, with $i \in \{1,2\}$.*

2. *If $R \equiv_{\mathsf{S}} x\langle v\rangle.P \mid \diamond\, y(w).Q$, then $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R]\!] \longrightarrow_1 \equiv C_{\widetilde{x}\widetilde{y}}[(\!|R|\!)^1_{\widetilde{x}\widetilde{y}}] \longrightarrow_1 \cong_1^{\pi^\lightning_{\mathsf{OR}}} [\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R']\!]$.*

3. *If $R \equiv_{\mathsf{S}} x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$, with $j \in I$, then we have the reductions in Fig. 4.6.*

Proof (see Page 136). This proof proceeds by using the translation (cf. Fig. 4.2) the `lcc` semantics (cf. Fig. 2.5). All items are shown in the same way; we detail only Item 3, which is arguably the most interesting case:

3. By assumption, $R \equiv_{\mathsf{S}} x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$, with $j \in I$ and $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'$. By Fig. 2.1, $(\boldsymbol{\nu}xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i{:}Q_i\}_{i\in I}) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q_j)$, with $j \in I$. Finally, by Fig. 4.2, Fig. 2.5 and expanding Not. 4.12:

$$\begin{aligned}
[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R]\!] = \exists\widetilde{x},\widetilde{y}.(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel \overline{\mathsf{sel}(x,l_j)} \parallel \forall z\big(\mathsf{bra}(z,l_j) \otimes \{x{:}z\} \to [\![P]\!]\big) \parallel \\
\forall l, w\big(\mathsf{sel}(w,l) \otimes \{w{:}y\} \to \overline{\mathsf{bra}(y,l)} \parallel \prod_{1\leq i\leq n} \forall\epsilon(l = l_i \to [\![P_i]\!])\big))\\
\xrightarrow{\tau}_1 \exists\widetilde{x},\widetilde{y}.(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel \forall z\big(\mathsf{bra}(z,l_j) \otimes \{x{:}z\} \to [\![P]\!]\big) \parallel \\
\overline{\mathsf{bra}(y,l_j)} \parallel \prod_{1\leq i\leq n} \forall\epsilon(l_j = l_i \to [\![P_i]\!]))\\
\equiv \exists\widetilde{x},\widetilde{y}.(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel \forall z\big(\mathsf{bra}(z,l_j) \otimes \{x{:}z\} \to [\![P]\!]\big) \parallel
\end{aligned}$$

$$\overline{\mathsf{bra}(y,l_j)} \parallel \forall\epsilon(l_j = l_j \rightarrow [\![P_j]\!]) \parallel \prod_{i\in I\setminus\{j\}} \forall\epsilon(l_j = l_i \rightarrow [\![P_i]\!])\big)$$

$$\equiv \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel (\!|x \triangleleft l_j.P \mid y \triangleright \{l_i{:}Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}}\big) = T$$

up to this point, we have shown that $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R]\!] \xrightarrow{\tau}_1 \equiv \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel (\!|R|\!)^1_{\widetilde{x}\widetilde{y}}\big)$. We now distinguish cases for the next reduction, as there are two possibilities:

(a) From Fig. 2.5 and Cor. 4.18:

$$T \xrightarrow{\tau}_1 \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel [\![P]\!] \parallel \forall\epsilon(l_j = l_j \rightarrow [\![P_j]\!]) \parallel$$
$$\prod_{i\in I\setminus\{j\}} \forall\epsilon(l_j = l_i \rightarrow [\![P_i]\!])\big)$$
$$\equiv \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel (\!|x \triangleleft l_j.P \mid y \triangleright \{l_i{:}Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}}\big)$$
$$\xrightarrow{\tau}_1 \cong^{\pi^{\lightning}_{\mathsf{OR}}}_1 \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel [\![P]\!] \parallel [\![P_j]\!]\big)$$

(b) From Fig. 2.5 and Cor. 4.18:

$$T \xrightarrow{\tau}_1 \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel \forall z\big(\mathsf{bra}(z,l_j) \otimes \{x{:}z\} \rightarrow [\![P]\!]\big) \parallel$$
$$\overline{\mathsf{bra}(y,l_j)} \parallel [\![P_j]\!] \parallel \prod_{i\in I\setminus\{j\}} \forall\epsilon(l_j = l_i \rightarrow [\![P_i]\!])\big)$$
$$\equiv \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel (\!|x \triangleleft l_j.P \mid y \triangleright \{l_i{:}Q_i\}_{i\in I}|\!)^3_{\widetilde{x}\widetilde{y}}\big)$$
$$\xrightarrow{\tau}_1 \cong^{\pi^{\lightning}_{\mathsf{OR}}}_1 \exists\widetilde{x},\widetilde{y}.\big(\overline{!\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}} \parallel [\![P]\!] \parallel [\![P_j]\!]\big)$$

□

B.4 Invariants for Well-Typed Translated Programs

Lemma 4.30. *Let P be a well-typed program. If $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S$ then*

$$S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n \parallel J]$$

where $n \geq 1$, J is some junk, and for all $i \in \{1, \dots, n\}$ we have $U_i = \overline{\mathtt{tt}}$ or one the following:

1. *$U_i = [\![R_k]\!]$, where R_k is a conditional redex (cf. Def. 2.19) reachable from P;*
2. *$U_i = [\![R_k]\!]$, where R_k is a pre-redex reachable from P;*
3. *$U_i \in \{\![R_k \mid R_j]\!\}$ (cf. Def. 4.25), where redex $R_k \mid R_j$ is reachable from P.*

Proof (*see Page 138*). By induction on the length k of the reduction $\overset{\tau}{\Longrightarrow}_1$. The base case ($k = 0$) is immediate: since $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 [\![P]\!]$, by Lem. 4.13 we have $S = [\![P]\!] = C_{\widetilde{x}\widetilde{y}}[[\![R_1]\!] \parallel \cdots \parallel [\![R_n]\!]]$, and the property holds because every $[\![R_i]\!]$ is captured by Cases (1) and (2).

The inductive step ($k > 0$) proceeds by a case analysis of the transition $S_0 \xrightarrow{\tau}_1 S$. We state the IH:

IH1: If $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_0 \xrightarrow{\tau}_1 S$, then $S_0 = C_{\widetilde{x}\widetilde{y}}[W_1 \parallel \cdots \parallel W_m \parallel J_0]$ where $m \geq 1$, for some junk J_0, and every W_i is either $\overline{\mathtt{tt}}$ or satisfies one of the three cases.

The transition $S_0 \xrightarrow{\tau}_1 S$ can only originate in some $W_i \neq \overline{\mathtt{tt}}$. There are then three cases to consider: W_i is a conditional redex, a pre-redex, or an intermediate process. We have:

Case $W_i = [\![b?\,(P_1){:}(P_2)]\!]$ **with** $b \in \{\mathtt{tt}, \mathtt{ff}\}$**:** There are two sub-cases, depending on whether $b = \mathtt{tt}$ or $b = \mathtt{ff}$. We only detail the case $b = \mathtt{tt}$, as the case $b = \mathtt{ff}$ proceeds similarly. We have:

(1) $W_i = \forall\epsilon(\mathtt{tt} = \mathtt{tt} \to [\![P_1]\!]) \parallel \forall\epsilon(\mathtt{tt} = \mathtt{ff} \to [\![P_2]\!])$ (Fig. 4.2).

(2) $\exists P'.(P \longrightarrow^* P' = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(\mathtt{tt}?\,(P_1){:}(P_2) \mid Q))$ (IH1).

(3) $P' \longrightarrow P'' = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid Q)$ (Fig. 2.1, (2)).

(4) $S_0 \xrightarrow{\tau}_1 S = C_{\widetilde{x}\widetilde{y}}[W_1 \parallel \cdots \parallel [\![P_1]\!] \cdots \parallel W_m \parallel J]$, with $J = \forall\epsilon(\mathtt{tt} = \mathtt{ff} \to [\![P_2]\!]) \parallel J_0$ (Fig. 2.5, (1)).

To conclude this case, we proceed by induction on the structure of P_1:

Case $P_1 = \mathbf{0}$**:** By (4) and Fig. 4.2, $S = C_{\widetilde{x}\widetilde{y}}[W_1 \parallel \cdots \parallel \overline{\mathtt{tt}} \parallel \cdots \parallel W_m \parallel J]$, and so the thesis follows.

Case $P_1 = b?\,(Q_1){:}(Q_2)$**:** By (4) and Fig. 4.2, $S = C_{\widetilde{x}\widetilde{y}}[W_1 \parallel \cdots \parallel [\![P_1]\!] \parallel \cdots \parallel W_m \parallel J]$. Hence, the thesis follows under Case (1).

Cases: $P_1 = x\langle v\rangle.P$, $P_1 = x(y).Q$, $P_1 = x \triangleleft l_j.Q$, $P_1 = *\,x(y).Q$, and $P_1 = x \triangleright \{l_i : Q_i\}_{i\in I}$. From the rules in Fig. 4.2 and (4), $S = C_{\widetilde{x}\widetilde{y}}[W_1 \parallel \cdots \parallel [\![P_1]\!] \parallel \cdots \parallel W_m \parallel J]$. Hence, the thesis follows under Case (2).

Case $P_1 = Q_1 \mid Q_2$**:** By IH, the thesis holds for $[\![Q_1]\!]$ and $[\![Q_2]\!]$, and the reduction from S_0 to S generates one additional parallel process inside $C_{\widetilde{x}\widetilde{y}}[\cdot]$.

Case $P_1 = (\boldsymbol{\nu}xy)Q$**:** By IH, the thesis holds for $[\![Q]\!]$. By noticing that:

$$\begin{aligned}&C_{\widetilde{x}\widetilde{y}}[W_1 \parallel \cdots \parallel [\![(\boldsymbol{\nu}xy)Q]\!] \parallel \cdots \parallel W_m \parallel J]\\ &= C_{\widetilde{x}x\widetilde{y}y}[W_1 \parallel \cdots \parallel [\![Q]\!] \parallel \cdots \parallel W_m \parallel J]\end{aligned}$$

the thesis follows.

Case $W_i = [\![R_k]\!]$, for some pre-redex R_j: Then the transition from S_0 to S can only occur if there exists a $W_j = [\![R_j]\!]$, such that $R_k \mid R_j$ is a redex reachable from P. There are multiple sub-cases, depending on the shape of R_k and R_j. We only detail a representative sub-case; the rest are similar:

Sub-case $R_k = x\langle v\rangle.P$: We then have that $R_k = y(z).Q$ and so

$$\begin{aligned}S_0 &= C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel W_i \parallel \cdots \parallel W_j \parallel \cdots \parallel W_m]\\ &= C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel [\![x\langle v\rangle.P]\!] \parallel \cdots \parallel [\![y(z).Q]\!] \parallel \cdots \parallel W_m]\\ &\xrightarrow{\tau}_1 C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel (\!|x\langle v\rangle.P \mid y(z).Q|\!)^1_{\widetilde{xy}} \parallel \cdots \parallel W_m] = S\end{aligned}$$

where the transition to S follows Lem. 4.27(2). The thesis then follows Case (3).

Case $W_i \in \{\![R_k \mid R_j]\!\}$, for some redex $R_k \mid R_j$: Then, depending on the shape of R_k and R_j (and relying on Not. 4.26), the transition from S_0 to S corresponds to one of the following five sub-cases:

(a) $W_i = (\!|x\langle v\rangle.P \mid y(z).Q|\!)^1_{\widetilde{xy}}$

(b) $W_i = (\!|x\langle v\rangle.P \mid *\, y(z).Q|\!)^1_{\widetilde{xy}}$

(c) $W_i = (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{xy}}$

(d) $W_i = (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{xy}}$

(e) $W_i = (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{xy}}$

We only detail Sub-cases (a), (c) and (e); the rest are similar:

Sub-case $W_i = (\!|x\langle v\rangle.P \mid y(z).Q|\!)^1_{\widetilde{xy}}$: Then we have:

$$\begin{aligned}S_0 = \ & C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel (\!|x\langle v\rangle.P \mid y(z).Q|\!)^1_{\widetilde{xy}} \parallel \cdots \parallel W_m]\\ &\xrightarrow{\tau}_1 C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel [\![P]\!] \parallel [\![Q]\!]\{{}^{v}\!/_{z}\} \parallel \cdots \parallel W_m] = S\end{aligned}$$

and the proof proceeds by a simultaneous induction on the structure of both P and Q, as shown for the case of the conditional redex above.

Sub-case $W_i = (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{xy}}$: Then we have:

$$\begin{aligned}S_0 = \ & C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{xy}} \parallel \cdots \parallel W_m]\\ &\xrightarrow{\tau}_1 C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^k_{\widetilde{xy}} \parallel \cdots \parallel W_m] = S\end{aligned}$$

Sub-case $W_i = (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{xy}}$: Assuming $l = l_j$ for some $j \in I$, then we have:

$$\begin{aligned}S_0 = \ & C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel (\!|x \triangleleft l.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{xy}} \parallel \cdots \parallel W_m]\\ &\xrightarrow{\tau}_1 C_{\widetilde{xy}}[W_1 \parallel \cdots \parallel [\![P]\!] \parallel [\![Q_j]\!] \parallel \cdots \parallel W_m] = S\end{aligned}$$

and the proof proceeds by a simultaneous induction on the structure of both P and Q, as shown for the case of the conditional redex above.

□

Lemma 4.31. *Let P be a well-typed $\pi^{\sharp}_{\mathsf{OR}}$ program. Then, for every S, S' such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S \xrightarrow{\tau}_1 S'$ one of the following holds:*

(*a*) $\mathcal{I}_S \subseteq \mathcal{I}_{S'}$ (*cf. Not. 4.29*) *and one of the following holds:*

1. $S \equiv C_{\widetilde{x}\widetilde{y}}[[\![b?\,(P_1):(P_2)]\!] \parallel U]$ *and* $S' = C_{\widetilde{x}\widetilde{y}}[[\![P_i]\!] \parallel U]$, *with* $i \in \{1, 2\}$;
2. $S \equiv C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$ *and* $S' = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{x}\widetilde{y}} \parallel U]$;
3. $S \equiv C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel U]$ *and* $S' = C_{\widetilde{x}\widetilde{y}}[[\![P']\!] \parallel [\![Q_j]\!] \parallel U]$.

(*b*) $\mathcal{I}_S \not\subseteq \mathcal{I}_{S'}$ *and* $|\mathcal{I}_S \setminus \mathcal{I}_{S'}| = 1$.

Proof (*see Page 138*). We first use Lem. 4.30 to characterize every parallel sub-process U_i of S; then, by a case analysis on the shape of the U_i that originated the transition $S \xrightarrow{\tau}_1 S'$ it is shown how each case will fall under either (a) or (b). More in details, by Lem. 4.30 we have:

$$S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n]$$

where for every U_i either $U_i = \overline{\mathtt{tt}}$ or

(i) $U_i = [\![R_k]\!]$, where R_k is a conditional redex reachable from P;

(ii) $U_i = [\![R_k]\!]$, where R_k is a pre-redex reachable from P;

(iii) $U_i \in \{\![R_k \mid R_j]\!\}$, where redex $R_k \mid R_j$ is reachable from P.

Hence, transition $S \xrightarrow{\tau}_1 S'$ must originate from some U_i. There are 12 different possibilities for this transition:

A. $U_i = [\![\mathtt{tt}?\,(Q_1):(Q_2)]\!]$;

B. $U_i = [\![\mathtt{ff}?\,(Q_1):(Q_2)]\!]$;

C. $U_i = [\![x\langle v\rangle.Q_1]\!]$;

D. $U_i = [\![x(z).Q_1]\!]$;

E. $U_i = [\![x \triangleleft l.Q_1]\!]$;

F. $U_i = [\![x \triangleright \{l_i : Q_i\}_{i\in I}]\!]$;

G. $U_i = [\![*\,x(z).Q_1]\!]$.

H. $U_i = (\!|x\langle v\rangle.Q_1 \mid y(z).Q_2|\!)^1_{\widetilde{x}\widetilde{y}}$;

I. $U_i = (\!|x\langle v\rangle.Q_1 \mid *\,y(z).Q_2|\!)^1_{\widetilde{x}\widetilde{y}}$;

J. $U_i = (\!|x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}}$;

K. $U_i = (\!|x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}}$;

L. $U_i = (\!|x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{x}\widetilde{y}}$;

Notice that in Sub-cases A-B and H-L, the U_i can transition by itself; in Sub-cases C-G, the U_i needs to interact with some other U_j (with $i \neq j$) to produce the transition. Also, notice that in Sub-case J, two more sub-cases are generated, which depend on the transition induced by $U_i = (\!|x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}}$:

J(1). $S \xrightarrow{\tau}_1 S' = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel \cdots \parallel U_n]$

J(2). $S \xrightarrow{\tau}_1 S' = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{x}\widetilde{y}} \parallel \cdots \parallel U_n]$

These two additional sub-cases are distinguished according to Lem. 4.27(3). All sub-cases are proven in the same way: first, identify the exact shape of S involved, and use the appropriate rule(s) in Fig. 2.5 to obtain S'. Next, compare the stores $\mathcal{I}_S$ and $\mathcal{I}_{S'}$. If $\mathcal{I}_S \subseteq \mathcal{I}_{S'}$, then the sub-case falls under (a). Otherwise, the sub-case falls under (b). We detail the proof for two representative sub-cases:

Sub-Case A: Since the U_i that originates the transition is a conditional redex then $S = C_{\widetilde{x}y}[U_1 \parallel \cdots \parallel [\![\mathtt{tt}?\,(Q_1)\!:\!(Q_2)]\!] \parallel \cdots \parallel U_n]$. By Fig. 2.5, and eliminating the junk with Cor. 4.18, we have:

$$S \xrightarrow{\tau}_1 \cong_1^{\pi^{\lightning}_{\mathsf{OR}}} S' = C_{\widetilde{x}y}[U_1 \parallel \cdots \parallel [\![Q_1]\!] \parallel \cdots \parallel U_n]$$

Then, we are left to prove that $\mathcal{I}_S \subseteq \mathcal{I}_{S'}$. This follows straightforwardly by considering that:

$$\forall e \in \mathcal{I}_S.(e \in \mathcal{I}_{S'})$$

because the transition of a conditional redex does not consume any constraint and that:

$$\forall e \in \mathcal{I}_{[\![Q_1]\!]}.(e \in \mathcal{I}_{S'})$$

because new constraints are added by $[\![Q_1]\!]$. Hence, this sub-case falls under (a).

Sub-Case H: Notice that well-typedness, via Lem. 3.16, ensures that there will never be two processes in parallel prefixed with the same variable, unless they are input processes. Furthermore, it is not possible for more than a single input process to interact with its corresponding partner, ensuring the uniqueness of the constraint. Using this, we can detail the case:

1. $U_i = (\!|x\langle.\rangle R' \mid y(z).R''|\!)^1_{\widetilde{x}\widetilde{y}} = \overline{\mathsf{rcv}(y,v)} \parallel \forall z(\mathsf{rcv}(z,v) \otimes \{z{:}x\} \to [\![R']\!]) \parallel [\![R''\{v/x\}]\!]$ (Not. 4.26).
2. $\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_i \parallel \cdots \parallel U_n]) = \{\exists \widetilde{x}, \widetilde{y}.\mathsf{rcv}(y,v)^m\} \cup \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n])$ (Def. 4.22,(1)).
3. $S \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![R']\!] \parallel [\![R''\{v/x\}]\!] \parallel \cdots \parallel U_n]$ (Fig. 2.5 - Rule (C:Sync), (1)).
4. $\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![R']\!] \parallel [\![R''\{v/x\}]\!] \parallel \cdots \parallel U_n]) = \{\exists \widetilde{x}\widetilde{y}.c \mid c \in \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}([\![R']\!] \parallel [\![R''\{v/x\}]\!])\} \cup \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n])$ (Def. 4.22, (3)).
5. $\mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_i \parallel \cdots \parallel U_n]) \setminus \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![R']\!] \parallel [\![R''\{v/x\}]\!] \parallel \cdots \parallel U_n]) = \{\exists \widetilde{x}, \widetilde{y}.\mathsf{rcv}(y,v)\}$ (Set difference, (2),(4)).

We can then conclude by observing that:
$\mathcal{I}_S = \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_i \parallel \cdots \parallel U_n])$;
$\mathcal{I}_{S'} = \mathcal{I}^{\mathcal{D}^{\star}_{\pi^{\lightning}_{\mathsf{OR}}}}(C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![R']\!] \parallel [\![R''\{v/x\}]\!] \parallel \cdots \parallel U_n])$
and considering that:
$|\mathcal{I}_S \setminus \mathcal{I}_{S'}| = |\{\exists \widetilde{x}, \widetilde{y}.\mathsf{rcv}(y,v)\}| = 1$. Hence, this sub-case falls under (b).

Sub-Case	S'	(a)	(b)
A	$C_{\widetilde{x}y}[U_1 \parallel \cdots \parallel \llbracket P_1 \rrbracket \parallel \cdots \parallel U_n]$	✓	
B	$C_{\widetilde{x}y}[U_1 \parallel \cdots \parallel \llbracket P_1 \rrbracket \parallel \cdots \parallel U_n]$	✓	
C	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llparenthesis x\langle v\rangle.Q_1 \mid y(z).Q_2 \rrparenthesis^1_{\widetilde{xy}} \parallel \cdots \parallel U_n]$		✓
D	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llparenthesis x(z).Q_1 \mid y\langle v\rangle.Q_2 \rrparenthesis^1_{\widetilde{xy}} \parallel \cdots \parallel U_n]$		✓
E	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llparenthesis x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^1_{\widetilde{xy}} \parallel \cdots \parallel U_n]$		✓
F	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llparenthesis x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^1_{\widetilde{xy}} \parallel \cdots \parallel U_n]$		✓
G	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llparenthesis y\langle v\rangle.Q_1 \mid *\, x(z).Q_2 \rrparenthesis^1_{\widetilde{xy}} \parallel \cdots \parallel U_n]$		✓
H	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llbracket Q_1 \rrbracket \parallel \llbracket Q_2 \rrbracket\{^v\!/z\} \parallel \cdots \parallel U_n]$		✓
I	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llbracket Q_1 \rrbracket \parallel \llbracket Q_2 \rrbracket\{^v\!/z\} \parallel \llbracket *\, y(z).Q_2 \rrbracket \parallel \cdots \parallel U_n]$		✓
J(1)	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llparenthesis x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^2_{\widetilde{xy}} \parallel \cdots \parallel U_n]$		✓
J(2)	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llparenthesis x \triangleleft l_j.Q \mid y \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^3_{\widetilde{xy}} \parallel \cdots \parallel U_n]$	✓	
K	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llbracket Q \rrbracket \parallel \llbracket Q_j \rrbracket \parallel \cdots \parallel U_n]$		✓
L	$C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel \llbracket Q \rrbracket \parallel \llbracket Q_j \rrbracket \parallel \cdots \parallel U_n]$	✓	

Table B.1: Proof of Lem. 4.31: summary of the case analysis. Recall that $S = C_{\widetilde{x}y}[U_1 \parallel \cdots \parallel U_i \parallel \cdots \parallel U_n]$.

Table B.1 summarizes the results for all sub-cases.

$\square$

Lemma 4.33 (Invariants of Target Terms (I): Adding Information). *Let P be a well-typed $\pi^{\sharp}_{\mathsf{OR}}$ program. For any S, S' such that $\llbracket P \rrbracket \overset{\tau}{\Longrightarrow}_1 S \xrightarrow{\tau}_1 S'$ and $\mathcal{I}_S \subseteq \mathcal{I}_{S'}$ (cf. Not. 4.29) one of the following holds, for some U:*

1. *$S \equiv C_{\widetilde{z}}[\llbracket b?\,(P_1) : (P_2) \rrbracket \parallel U \parallel J_1]$ and $S' = C_{\widetilde{z}}[\llbracket P_i \rrbracket \parallel \forall\epsilon(b = \neg b \to P_j) \parallel U \parallel J_1]$ with $i, j \in \{1, 2\}, i \neq j$;*
2. *$\llbracket P \rrbracket \overset{\tau}{\Longrightarrow}_1 S_0 \equiv C_{\widetilde{xy}}[\overline{\{x{:}y\}} \parallel \llbracket x \triangleleft l_j.P' \parallel y \triangleright \{l_i\ Q_i\}_{i\in I} \rrbracket \parallel U \parallel J_1]$ and either:*
   - *(a) All of the following hold:*
     - *(i) $S_0 \xrightarrow{\tau}_1 C_{\widetilde{xy}}[\llparenthesis y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^1_{\widetilde{xy}} \parallel U \parallel J_1] \xrightarrow{\tau}_1 S$,*
     - *(ii) $S = C_{\widetilde{xy}}[\llparenthesis y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^2_{\widetilde{xy}} \parallel U \parallel J_1]$ (and)*
     - *(iii) $S' = C_{\widetilde{xy}}[\llbracket P' \rrbracket \parallel \llbracket Q_j \rrbracket \parallel U \parallel J_1 \parallel J_2]$.*
   - *(b) All of the following hold:*
     - *(i) $S_0 \xrightarrow{\tau}_1 S = C_{\widetilde{xy}}[\llparenthesis y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^1_{\widetilde{xy}} \parallel U \parallel J_1]$,*
     - *(ii) $S' = C_{\widetilde{xy}}[\llparenthesis y \triangleleft l_j.P' \mid x \triangleright \{l_i : Q_i\}_{i\in I} \rrparenthesis^3_{\widetilde{xy}} \parallel U \parallel J_1]$ (and)*
     - *(iii) $S' \xrightarrow{\tau}_1 C_{\widetilde{xy}}[\llbracket P' \rrbracket \parallel \llbracket Q_j \rrbracket \parallel U \parallel J_1 \parallel J_2]$.*

   where $J_2 = \prod_{k \in I\setminus\{j\}} \forall\epsilon(l_j = l_k \to \llbracket P_k \rrbracket)$.

Proof (*see Page 140*). By induction on the length of the transition $\overset{\tau}{\Longrightarrow}_1 \xrightarrow{\tau}_1$. First, by applying Lem. 4.13:

$$\llbracket P \rrbracket \equiv C_{\widetilde{xy}}[\llbracket R_1 \rrbracket \parallel \cdots \parallel \llbracket R_n \rrbracket] \tag{1}$$

where every R_i is either a pre-redex or a conditional process. We apply induction on the length of transition $\overset{\tau}{\Longrightarrow}_1$:

Base Case: We analyze whenever $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 [\![P]\!] \xrightarrow{\tau}_1 S'$. Thus, let $S = [\![P]\!]$. Since $\mathcal{I}_S \subseteq \mathcal{I}_{S'}$, then by Lem. 4.31(a), we have:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[[\![R_j]\!] \parallel \prod_{i\in\{1\ldots n\}\setminus j} [\![R_i]\!]]$$
$$S' \equiv C_{\widetilde{x}\widetilde{y}}[S_j \parallel \prod_{i\in\{1\ldots n\}\setminus\{j\}} [\![R_i]\!]]$$

where $[\![R_j]\!] = [\![b?\,(Q_1) : (Q_2)]\!]$. Notice that we only analyze Item (1) of the statement, as Item (2) would require S to contain intermediate redexes, which is not possible since S is the translation of a process without any preceding transition. By assumption, P is a well-typed program, therefore, by Def. 3.14, $b \in \{\texttt{tt}, \texttt{ff}\}$. We distinguish cases for each $b = \texttt{tt}$ and $b = \texttt{ff}$. We only show the case $b = \texttt{tt}$, as the other is similar.

Case $b = \texttt{tt}$**:** By Fig. 4.2:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[\forall\epsilon(\texttt{tt} = \texttt{tt} \rightarrow [\![Q_1]\!]) \parallel \forall\epsilon(\texttt{tt} = \texttt{ff} \rightarrow [\![Q_2]\!]) \parallel \prod_{i\in\{1\ldots n\}\setminus\{j\}} [\![R_i]\!]]$$

By applying the rules in Fig. 2.5:

$$S \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[[\![Q_1]\!] \parallel \forall\epsilon(\texttt{tt} = \texttt{ff} \rightarrow [\![Q_2]\!]) \parallel \prod_{i\in\{1\ldots n\}\setminus j} [\![R_i]\!]] \equiv S'$$

By Def. 4.14, let $J = \overline{\texttt{tt}}$ and $J' = J \parallel \forall\epsilon(\texttt{tt} = \texttt{ff} \rightarrow [\![Q_2]\!])$. Therefore, by Def. 2.28:

$$U \equiv C_{\widetilde{x}\widetilde{y}}[\prod_{i\in\{1\ldots n\}\setminus j} [\![R_i]\!] \parallel J]$$
$$U' \equiv C_{\widetilde{x}\widetilde{y}}[\parallel \prod_{i\in\{1\ldots n\}\setminus\{j\}} [\![R_i]\!] \parallel J']$$

Finally, let $[\![R_i]\!] = U_i$ for every $i \in i \in \{1\ldots n\} \setminus \{j\}$, finishing the proof.

Inductive Step: By IH, $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_0 \xrightarrow{\tau}_1 S$ satisfies the property for m steps (i.e., $[\![P]\!] \xrightarrow{\tau}{}_1^{m-1} S_0 \xrightarrow{\tau}_1 S$). We must prove for $k = m + 1$:

$$[\![P]\!] \xrightarrow{\tau}{}_1^{m} S \xrightarrow{\tau}_1 S'$$

by Lem. 4.30:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n \parallel J]$$

for some junk J and for all $i \in \{1, \ldots, n\}$ either:

1. $U_i = [\![R_k]\!]$, where R_k is a conditional redex reachable from P;
2. $U_i = [\![R_k]\!]$, where R_k is a pre-redex reachable from P;
3. $U_i \in \{\![R_k \mid R_j]\!\}$, where redex $R_k \mid R_j$ is reachable from P.

Then, by Lem. 4.31(a), there exists $\llbracket R_j \rrbracket$ such that:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[U_j \parallel \prod_{i\in\{1\ldots n\}\setminus j} U_i \parallel J]$$
$$S' \equiv C_{\widetilde{x}\widetilde{y}}[U'_j \parallel \prod_{i\in\{1\ldots n\}\setminus j} U_i \parallel J']$$

and only cases (1), (3) will be considered:

Case $U_j = \llbracket R_j \rrbracket$ **with** R_j **a conditional redex:** Since $\mathcal{I}_S \subseteq \mathcal{I}_{S'}$, by inspection on Fig. 4.2, $R_j = b?\,(P_1):(P_2)$, with $b \in \{\mathtt{tt}, \mathtt{ff}\}$ and the case proceeds as the base case.

Case $U_j \in \{\![R_j \mid R_k]\!\}$**:** By inspection on Def. 4.25and Cor. 4.18, we have that $U_j \in \{\![y \triangleleft l_j.Q \mid x \triangleright \{l_i : Q_i\}_{i\in I}]\!\}$, for some Q, Q_i, l_j, and either,

(i) $U_j \cong^{\pi_{\mathsf{OR}}^{\lightning}}_1 \overline{\mathsf{bra}(y, l_j)} \parallel \forall z(\mathsf{bra}(z, l_j) \otimes \{z{:}x\} \rightarrow \llbracket Q \rrbracket) \parallel \forall \epsilon(l_j = l_j \rightarrow \llbracket Q_j \rrbracket)$, or

(ii) $U_j \cong^{\pi_{\mathsf{OR}}^{\lightning}}_1 \llbracket Q \rrbracket \parallel \forall \epsilon(l_j = l_j \rightarrow \llbracket Q_j \rrbracket)$.

We analyze each case:

Case (i): If $U_j = \overline{\mathsf{bra}(y, l_j)} \parallel \forall z(\mathsf{bra}(z, l_j) \otimes \{z{:}x\} \rightarrow \llbracket Q \rrbracket) \parallel \forall \epsilon(l_j = l_j \rightarrow \llbracket Q_j \rrbracket)$, then, by Cor. 4.28, there exists S_0, such that:

$$S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel \llbracket x \triangleleft l_j.Q \mid y \triangleright \{l_i\, Q_i\}_{i\in I} \rrbracket \parallel U_1 \parallel \cdots \parallel U_n \parallel J]$$

by the semantics in Fig. 2.5 and Cor. 4.18:

$$\begin{aligned} S_0 &\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel U_j \parallel U_1 \parallel \cdots \parallel U_n \parallel J] = S \\ &\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel \overline{\mathsf{bra}(y, l_j)} \parallel \forall z(\mathsf{bra}(z, l_j) \otimes \{z{:}x\} \rightarrow \llbracket Q \rrbracket) \parallel \llbracket Q_j \rrbracket \\ &\qquad\qquad \parallel U_1 \parallel \cdots \parallel U_n \parallel J \parallel] = S' \\ &\xrightarrow{\tau}_1 \cong^{\pi_{\mathsf{OR}}^{\lightning}}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel \llbracket Q \rrbracket \parallel \llbracket Q_j \rrbracket \parallel U_1 \parallel \cdots \parallel U_n \parallel J \parallel \\ &\qquad\qquad \prod_{h\in I} \forall \epsilon(l_h = l_j \rightarrow \llbracket Q_i \rrbracket)] \end{aligned}$$

The proof finalizes by letting $J' = J \parallel \prod_{h\in I} \forall \epsilon(l_h = l_j \rightarrow \llbracket Q_i \rrbracket)$.

Case (ii): If $U_j = \llbracket Q \rrbracket \parallel \forall \epsilon(l_j = l_j \rightarrow \llbracket Q_j \rrbracket)$, then by Def. 4.25 and Not. 4.26, there exists S_0, such that:

$$S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel \llparenthesis x \triangleleft l_j.Q \mid y \triangleright \{l_i\, Q_i\}_{i\in I} \rrparenthesis^{1}_{\widetilde{x}\widetilde{y}} \parallel U_1 \parallel \cdots \parallel U_n \parallel J]$$

by the semantics in Fig. 2.5:

$$\begin{aligned} S_0 &\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel \overline{\mathsf{bra}(y, l_j)} \parallel \forall z(\mathsf{bra}(z, l_j) \otimes \{z{:}x\} \rightarrow \llbracket Q \rrbracket) \parallel \llbracket Q_j \rrbracket \\ &\qquad\qquad \parallel U_1 \parallel \cdots \parallel U_n \parallel J] = S \\ &\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel \llbracket Q \rrbracket \parallel \llbracket Q_j \rrbracket \parallel U_1 \parallel \cdots \parallel U_n \parallel J \parallel \\ &\qquad\qquad \prod_{h\in I} \forall \epsilon(l_h = l_j \rightarrow \llbracket Q_i \rrbracket)] = S' \end{aligned}$$

The proof finishes by letting $J' = J \parallel \prod_{h\in I} \forall \epsilon(l_h = l_j \rightarrow \llbracket Q_i \rrbracket)$.

□

Lemma 4.34 (Invariants of Target Terms (II): Consuming Information). *Let P be a well-typed $\pi_{\mathsf{OR}}^{\lightning}$ program. For any S, S' such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S \xrightarrow{\tau}_1 S'$ and $\mathcal{I}_S \not\subseteq \mathcal{I}_{S'}$ the following holds, for some U:*

1. *If $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{snd}(x_1, v)^k_{\widetilde{x}\widetilde{y}}\}$ then all the following hold:*
   - (a) $S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x_1{:}y_1\}} \parallel [\![x_1\langle v\rangle.P_1 \mid \diamond\, y_1(z).P_2]\!] \parallel U]$;
   - (b) $S' = C_{\widetilde{x}\widetilde{y}}[(\!|x_1\langle v\rangle.P_1 \mid \diamond\, y_1(z).P_2|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$;
   - (c) $S' \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[[\![P_1 \mid P_2\{v/z\}]\!] \parallel S'' \parallel U]$, *where* $S'' = *\,[\![y(z).P_2]\!]$ *or* $S'' = \overline{\mathtt{tt}}$.
2. *If $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{rcv}(x_1, v)^k_{\widetilde{x}\widetilde{y}}\}$ then there exists S_0 such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_0 \xrightarrow{\tau}_1 S$ and all of the following hold:*
   - (a) $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x_1{:}y_1\}^m} \parallel [\![y_1\langle v\rangle.P_1 \mid \diamond\; x_1(z).P_2]\!] \parallel U]$;
   - (b) $S = C_{\widetilde{x}\widetilde{y}}[(\!|y_1\langle v\rangle.P_1 \mid \diamond\, x_1(z).P_2|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$;
   - (c) $S' = C_{\widetilde{x}\widetilde{y}}[[\![P_1 \mid P_2\{v/z\}]\!] \parallel S'_1 \parallel U]$, *where* $S'_1 = *\,[\![y(z).P_2]\!]$ *or* $S'_1 = \overline{\mathtt{tt}}$.
3. *If $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{sel}(x_1, l_j)^k_{\widetilde{x}\widetilde{y}}\}$ then all of the following hold:*
   - (a) $S \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x_1{:}y_1\}} \parallel [\![x_1 \triangleleft l.P_1 \mid y_1 \triangleright \{l_i : P_i\}_{i\in I}]\!]U]$;
   - (b) $S' = C_{\widetilde{x}\widetilde{y}}[(\!|x_1 \triangleleft l.P_1 \mid y_1 \triangleright \{l_i : P_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U]$;
   - (c) $S_1 \xrightarrow{\tau}{}^{2}_{1} \cong^{\pi_{\mathsf{OR}}^{\lightning}}_{1} C_{\widetilde{x}\widetilde{y}}[[\![P_1 \mid P_j]\!] \parallel U']$, *with* $U' \equiv U \parallel \prod_{h\in I} \forall\epsilon(l_h = l_j \rightarrow [\![Q_h]\!])$.
4. *If $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{bra}(x, l_j)^k_{\widetilde{x}\widetilde{y}}\}$, then there exists $S_0 \equiv C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![x \triangleleft l_j.Q \mid y \triangleright \{l_i\; Q_i\}_{i\in I}]\!] \parallel U]$ such that $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S_0$ and either:*
   - (a) *All of the following hold:*
     - (i) $S_0 \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.Q \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U] \xrightarrow{\tau}_1 S$,
     - (ii) $S = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.Q \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^3_{\widetilde{x}\widetilde{y}} \parallel U]$ *(and)*
     - (iii) $S' = C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![Q \mid Q_j]\!] \parallel U']$.
   - (b) *All of the following hold:*
     - (i) $S_0 \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel U] \equiv S$,
     - (ii) $S' = C_{\widetilde{x}\widetilde{y}}[(\!|y \triangleleft l_j.P \mid x \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel U]$ *(and)*
     - (iii) $S' \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[\overline{\{x{:}y\}} \parallel [\![P \mid Q_j]\!] \parallel U']$.

   with $U' \equiv U \parallel \prod_{h\in I} \forall\epsilon(l_h = l_j \rightarrow [\![Q_h]\!])$.

Proof (see Page 140). By induction on the transition $\overset{\tau}{\Longrightarrow}_1\xrightarrow{\tau}_1$. By Lem. 4.13:

$$[\![P]\!] \equiv \exists\widetilde{x},\widetilde{y}.([\![R_1]\!] \parallel \cdots \parallel [\![R_n]\!] \parallel! \overline{\bigotimes_{\substack{x_i\in\widetilde{x},\\ y_i\in\widetilde{y}}} \{x_i{:}y_i\}}) \tag{1}$$

where each R_i is a pre-redex or a conditional process. Also, by Lem. 4.31(b), the difference of observables between a process and the process obtained in a single τ-transition is a singleton. Therefore, we apply a case analysis on each one of those singletons.

Base Case: Then $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 [\![P]\!] \xrightarrow{\tau}_1 S'$. Let $[\![P]\!] = S$. By assumption, $S \xrightarrow{\tau}_1 S'$ and $\mathcal{I}_S \not\subseteq \mathcal{I}_{S'}$. Thus, there is a constraint $c \in \mathcal{I}_S$ such that $c \notin \mathcal{I}_{S'}$. Considering Eq. (1) and by inspection on Fig. 4.2 we only analyze from Case (2), as cases (1) and (3) do not apply: Case (1) does not apply as it does not entail constraint consumption (cf. Lem. 4.31(a)) and Case (3) does not apply as there are no intermediate redexes in S.

Case $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{snd}(x_1, v)_{\widetilde{x}\widetilde{y}}\}$: By Lem. 4.23, there exists j such that:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[[\![R_j]\!] \parallel \prod_{i\in\{1\dots n\}\setminus j} [\![R_i]\!]]$$

where $[\![R_j]\!] \equiv [\![x_j\langle v\rangle.Q]\!]$. Since $\mathcal{I}_S \not\subseteq \mathcal{I}_{S'}$, and every $c \in \mathcal{I}_S$ is unique, by inspection on Fig. 4.2 and Lem. 4.24, there must exist an R_k such that:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[[\![x_j\langle v\rangle.Q]\!] \parallel [\![R_k]\!] \parallel \prod_{i\in\{1\dots n\}\setminus\{j,k\}} [\![R_i]\!]]$$

where $[\![R_k]\!] \equiv [\![y_j(z).Q']\!]$ or $[\![R_k]\!] \equiv [\![* \, y_j(z).Q']\!]$. Without loss of generality, we only show the case for $[\![R_k]\!] \equiv [\![y_j(z).Q']\!]$:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[[\![x_j\langle v\rangle.Q]\!] \parallel [\![y_j(z).Q']\!] \parallel \prod_{i\in\{1\dots n\}\setminus\{j,k\}} [\![R_i]\!]]$$

By the semantics of `lcc` (cf. Fig. 2.5) and Lem. 4.27:

$$\begin{aligned} S &\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x_j\langle v\rangle.Q \mid y_j(z).Q'|\!)^1_{\widetilde{x}\widetilde{y}} \parallel \prod_{i\in\{1\dots n\}\setminus\{j,k\}} [\![R_i]\!]] \\ &\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[[\![Q]\!] \parallel [\![Q'\{v/z\}]\!] \parallel \prod_{i\in\{1\dots n\}\setminus\{j,k\}} [\![R_i]\!]] = S' \end{aligned}$$

Case $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{sel}(x_1, l_j)_{\widetilde{x}\widetilde{y}}\}$: By Lem. 4.23, there exists j such that:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[[\![R_j]\!] \parallel \prod_{i\in\{1\dots n\}\setminus j} [\![R_i]\!]]$$

where $[\![R_j]\!] \equiv [\![x_j \triangleleft l_j.Q]\!]$. Furthermore, by following the same analysis as in the previous case, there must exists R_j, such that:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[[\![x_j \triangleleft l_j.Q]\!] \parallel [\![R_k]\!] \parallel \prod_{i\in\{1\dots n\}\setminus\{j,k\}} [\![R_i]\!]]$$

where $[\![R_k]\!] \equiv [\![y_j \triangleright \{l_i : Q_i\}_{i\in I}]\!]$. Then, by the semantics of `lcc` (cf. Fig. 2.5):

$$S \xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|y_j \triangleright \{l_i : Q_i\}_{i\in I}|\!)^1_{\widetilde{x}\widetilde{y}} \parallel \prod_{i\in\{1\dots n\}\setminus\{j,k\}} [\![R_i]\!]] = S'$$

$$\begin{aligned}
&\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[(\!|y_j \triangleright \{l_i : Q_i\}_{i\in I}|\!)^2_{\widetilde{x}\widetilde{y}} \parallel \prod_{i\in\{1\ldots n\}\setminus\{j,k\}} [\![R_i]\!]] \\
&\xrightarrow{\tau}_1 C_{\widetilde{x}\widetilde{y}}[[\![Q]\!] \parallel [\![Q_j]\!] \parallel \prod_{i\in\{1\ldots n\}\setminus\{j,k\}} [\![R_i]\!] \parallel \prod_{h\in I} \forall\epsilon(l_h = l_j \to [\![Q_i]\!])] \\
&\cong^{\pi^{\lightning}_{\mathsf{OR}}}_1 \exists\widetilde{x},\widetilde{y}.([\![Q]\!] \parallel [\![Q_j]\!] \parallel \prod_{i\in\{1\ldots n\}\setminus\{j,k\}} [\![R_i]\!] \parallel \prod_{h\in I} \forall\epsilon(l_h = l_j \to [\![Q_i]\!]))
\end{aligned}$$

Inductive Case: By IH, $[\![P]\!] \overset{\tau}{\Longrightarrow}_1 S \xrightarrow{\tau}_1 S'$ satisfies the property for m steps (i.e., $[\![P]\!] \xrightarrow{\tau}{}^{m-1}_1 S \xrightarrow{\tau}_1 S'$). Therefore:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel U_n \parallel J]$$

for some junk J and for all $i \in \{1, \ldots, n\}$ either:

1. $U_i = [\![R_k]\!]$, where R_k is a conditional redex reachable from P;
2. $U_i = [\![R_k]\!]$, where R_k is a pre-redex reachable from P;
3. $U_i \in \{[R_k \mid R_j]\}$, where redex $R_k \mid R_j$ is reachable from P.

We now have to prove for $k = m + 1$:

$$[\![P]\!] \xrightarrow{\tau}{}^{k}_1 S \xrightarrow{\tau}_1 S'$$

Since $S \xrightarrow{\tau}_1 S'$, there exists $[\![R_j]\!]$ such that:

$$\begin{aligned}
S &\equiv C_{\widetilde{x}\widetilde{y}}[U_j \parallel \prod_{i\in\{1\ldots n\}\setminus j} U_i \parallel J] \\
S' &\equiv C_{\widetilde{x}\widetilde{y}}[U'_j \parallel \prod_{i\in\{1\ldots n\}\setminus j} U_i \parallel J']
\end{aligned}$$

As above, we distinguish only cases for 2 and 3. Notice that using Lem. 4.31(a) and Lem. 4.33 we can discard case 1:

Case 2: Proceeds as the base case, by distinguishing cases between the consumed constraints. The cases correspond to constraints snd and sel (cf. Fig. 4.1).

Case 3: By inspection on Def. 4.25 and Not. 4.26, we distinguish two cases corresponding to predicates rcv, bra (cf. Fig. 4.1):

Case $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{rcv}(x_1, v)_{\widetilde{x}\widetilde{y}}\}$: By Lem. 4.23 and Lem. 4.24, there exists j such that:

$$S \equiv C_{\widetilde{x}\widetilde{y}}[U_j \parallel U_1 \parallel \cdots \parallel U_n \parallel J]$$

where $U_j = \overline{\mathsf{rcv}(x_j, v)} \parallel W$, for some W. By inspecting Def. 4.25 and Lem. 4.24, there exists U_k such that $U_j \parallel U_k \in \{[x_j(z).Q \mid y_j\langle v\rangle.Q']\}$ or $U_j \parallel U_k \in \{[* \, x_j(z).Q \mid y_j\langle v\rangle.Q']\}$, for some x_j, y_j, v. Without loss of generality, we will only analyze the case when

$$U_j \parallel U_k \in \{[x_j(z).Q \mid y_j\langle v\rangle.Q']\}$$

By Def. 4.25 and Not. 4.26, $U_j = (\!| x_j(y).Q \mid y_j\langle v\rangle.Q' |\!)^1_{\widetilde{xy}}$. By expanding the previous definitions:

$$S \equiv C_{\widetilde{xy}}[\overline{\mathsf{rcv}(x_j, v)} \parallel \forall w(\mathsf{rcv}(w, v) \otimes \{w{:}y_j\} \to [\![Q]\!]) \parallel [\![Q'\{v/z\}]\!] \parallel \\ U_1 \parallel \cdots \parallel U_n \parallel J]$$

and by the application of Rule (C:Sync) in Fig. 2.5 (i.e., the `lcc` semantics):

$$S \xrightarrow{\tau}_1 C_{\widetilde{xy}}[[\![Q]\!] \parallel [\![Q'\{v/z\}]\!] \parallel U_1 \parallel J]$$

Case $\mathcal{I}_S \setminus \mathcal{I}_{S'} = \{\mathsf{bra}(x, l_j)_{\widetilde{xy}}\}$: This case proceeds as above. The conclusion is reached using the same analysis as in the inductive case in the proof of Lem. 4.33.

□

B.5 A Diamond Property for Target Terms

Lemma 4.47. *Let S be a target term such that $S \xrightarrow{\omega}_1 S_1$ and $S \xRightarrow{\gamma(\widetilde{xy})}_1 S_2$, where $\gamma(\widetilde{xy})$ is a closing sequence (cf. Not. 4.42). Then, there exists S_3 such that $S_1 \xRightarrow{\gamma(\widetilde{xy})}_1 S_3$ and $S_2 \xrightarrow{\omega}_1 S_3$.*

Proof (see Page 148). By induction on the length n of $|\gamma(\widetilde{xy})|$.

Base Case: $n = 0$. Then:

(1) By Assumption, $S \xrightarrow{\omega}_1 S_1$.

(2) By Assumption, $S \xRightarrow{\gamma(\widetilde{xy})}_1 S$.

(3) By Fig. 2.5, $S_1 \xRightarrow{\gamma(\widetilde{xy})}_1 S_1$.

Conclude by letting $S_1 = S_1$, $S_2 = S$ and $S_3 = S_1$ and using (1) and (3).

Inductive Step: $n \geq 1$. We state the IH:

IH: If $S \xrightarrow{\omega}_1 S_1$ and $S \xRightarrow{\gamma_0(\widetilde{xy})}_1 S_0 \xrightarrow{\kappa}_1 S_2$, then there exists S_0' such that $S_1 \xRightarrow{\gamma_0(\widetilde{xy})}_1 S_0'$ and $S_0 \xrightarrow{\omega}_1 S_0'$.

We distinguish cases for $\kappa \in \{\mathtt{IO}_1, \mathsf{RP}_1, \mathsf{CD}, \mathsf{SL}_2, \mathsf{SL}_3\}$. There are five cases and each one has four sub-cases that correspond to the opening labels in the set $\{\mathtt{IO}, \mathsf{SL}, \mathsf{RP}, \mathsf{SL}_1\}$. We detail three cases: $\kappa = \mathtt{IO}_1$, $\kappa = \mathsf{RP}_1$ and $\kappa = \mathsf{CD}$, as the other are similar:

Case $\kappa = \mathtt{IO}_1$: As mentioned above, there are four sub-cases depending on ω. We enumerate them below and only detail $\omega = \mathtt{IO}$, $\omega = \mathsf{RP}$:

Sub-case $\omega = \texttt{IO}$**:** First assume that the actions take place on endpoints x, y and w, z. This is a general asumption. Furthermore, by typing and Cor. 3.16 and Def. 3.14, it cannot be the case that $x = w$ and $y = z$, because this would imply that there are more than one output prefixed on x. Then, we we only consider the case when $x \neq w$ and $y \neq z$:

Sub-case $x \neq w \wedge y \neq z$**:** We proceed as follows:

(1) By Assumption, Fig. 4.7, and Lem. 4.30:

$$S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(u_1).Q_2]\!] \parallel \cdots \parallel U_n \parallel J]$$

(2) By Fig. 2.5 and (1):

$$S_1 = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(u_1).Q_2|\!)^1_{xy} \parallel \cdots \parallel U_n \parallel J]$$

(3) By IH, (1), and Lem. 4.30:

$$S_0 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(u_1).Q_2]\!] \parallel \\ (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U_m' \parallel J']$$

with $m \geq 1$.

(4) By (3) and Fig. 2.5

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \\ [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel \cdots \parallel U_m' \parallel J']$$

(5) By IH $S_0 \xrightarrow{\texttt{IO}(x,y)}_1 S_0'$.

(6) By Fig. 2.5 and (5)

$$S_0' = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(u_1).Q_2|\!)^1_{xy} \parallel \\ (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U_m' \parallel J']$$

(7) By IH $S_1 \xRightarrow{\gamma_0(\widetilde{x}\widetilde{y})}_1 S_0'$.

We can then reduce the proof to show that there exists some S_3 such that $S_2 \xrightarrow{\texttt{IO}(x,y)}_1 S_3$ and $S_0' \xrightarrow{\texttt{IO}_1(w,z)}_1 S_3$:

(a) where:

$$S_0' = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(u_1).Q_2|\!)^1_{xy} \parallel \\ (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{xy} \parallel \cdots \parallel U_m' \parallel J']$$

.

(b) where:

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel \\ [\![Q_4]\!]\{v'/u_2\} \parallel \cdots \parallel U_m' \parallel J']$$

then, let

$$S_3 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid y(u_1).Q_2 |\!)^1_{xy} \parallel [\![Q_3]\!] \parallel \\ [\![Q_4]\!]\{^{v'}\!/u_2\} \parallel \cdots \parallel U'_m \parallel J']$$

and we can show by Fig. 2.5 that:

$$S'_0 \xrightarrow{\mathtt{IO}_1(w,z)}_1 S_3 \qquad \text{and} \qquad S_2 \xrightarrow{\mathtt{IO}(x,y)}_1 S_3$$

which concludes the proof.

Sub-case $\omega = \mathsf{RP}$: As above, assume the actions take place in variables x, y and w, z. It is not possible for them to happen in the same variable, by Cor. 3.16.

(1) By Assumption, Fig. 4.7, Lem. 4.30

$$S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \cdots \parallel U_n \parallel J]$$

(2) By Fig. 2.5 and (1)

$$S_1 = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2 |\!)^1_{xy} \parallel \cdots \parallel U_n \parallel J]$$

(3) By IH, (1) and Lem. 4.30

$$S_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel \\ [\![* \, y(u_1).Q_2]\!] \parallel (\!| w\langle v'\rangle.Q_3 \mid z(u_2).Q_4 |\!)^1_{wz} \parallel \cdots \parallel U'_m \parallel J']$$

with $m \geq 1$.

(4) By (3) and Fig. 2.5

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \\ [\![Q_3]\!] \parallel [\![Q_4]\!]\{^{v'}\!/u_2\} \parallel \cdots \parallel U'_m \parallel J']$$

(5) By IH $S_0 \xrightarrow{\mathsf{RP}(x,y)}_1 S'_0$.

(6) By Fig. 2.5 and (5)

$$S'_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2 |\!)^1_{xy} \parallel \\ (\!| w\langle v'\rangle.Q_3 \mid z(u_2).Q_4 |\!)^1_{wz} \parallel \cdots \parallel U'_m \parallel J']$$

(7) $S_1 \xRightarrow{\gamma_0(\widetilde{x}\widetilde{y})}_1 S'_0$ (IH).

We reduce the proof to show the existence of some some S_3 such that $S_2 \xrightarrow{\mathsf{RP}(x,y)}_1 S_3$ and $S'_0 \xrightarrow{\mathtt{IO}_1(w,z)}_1 S_3$:

(a) Where:

$$S'_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2 |\!)^1_{xy} \parallel \\ (\!| w\langle v'\rangle.Q_3 \mid z(u_2).Q_4 |\!)^1_{xy} \parallel \cdots \parallel U'_m \parallel J']$$

(b) Where:

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![*\,y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel \cdots \parallel U'_m \parallel J']$$

then, let

$$S_3 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid *\,y(u_1).Q_2|\!)^1_{xy} \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel \cdots \parallel U'_m \parallel J']$$

and we can show by Fig. 2.5 that:

$$S'_0 \xrightarrow{\texttt{IO}_1(w,z)}_1 S_3 \qquad \text{and} \qquad S_2 \xrightarrow{\texttt{RP}(x,y)}_1 S_3$$

which concludes the proof.

Sub-case $\omega = \texttt{SL}()$**:** Similarly as above.

Sub-case $\omega = \texttt{SL}_1()$**:** Similarly as above.

Case $\kappa = \texttt{RP}_1$**:** We proceed similarly as above. The most interesting case is $\omega = \texttt{RP}$:

Sub-case $\omega = \texttt{RP}$**:** As above, assume the actions take place in variables x, y and w, z. It is not possible for them to happen in the same variable, by Cor. 3.16.

(1) By Assumption, Fig. 4.7, and Lem. 4.30

$$S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![*\,y(u_1).Q_2]\!] \parallel \cdots \parallel U_n \parallel J]$$

(2) By Fig. 2.5 and (1)

$$S_1 = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid *\,y(u_1).Q_2|\!)^1_{xy} \parallel \cdots \parallel U_n \parallel J]$$

(3) By IH, (1), and Lem. 4.30

$$S_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![*\,y(u_1).Q_2]\!] \parallel (\!|w\langle v'\rangle.Q_3 \mid *\,z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U'_m \parallel J']$$

with $m \geq 1$.

(4) $S_2 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \ldots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![*\,y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel [\![*\,z(u_2).Q_4]\!] \parallel \ldots \parallel U'_m \parallel J']$ ((3), Fig. 2.5).

(5) By IH $S_0 \xrightarrow{\texttt{RP}(x,y)}_1 S'_0$.

(6) By Fig. 2.5 and (5)

$$S'_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid *\,y(u_1).Q_2|\!)^1_{xy} \parallel (\!|w\langle v'\rangle.Q_3 \mid *\,z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U'_m \parallel J']$$

(7) By IH $S_1, \xRightarrow{\gamma_0(\widetilde{xy})}_1 S'_0$.

We reduce the proof to show the existence of S_3 such that $S_2 \xrightarrow{\mathsf{RP}(x,y)}_1 S_3$ and $S'_0 \xrightarrow{\mathsf{RP}_1(w,z)}_1 S_3$:

(a) where

$$S'_0 = C_{\widetilde{xy}}[U'_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid * y(u_1).Q_2 |\!)^1_{xy} \parallel (\!| w\langle v'\rangle.Q_3 \mid * z(u_2).Q_4 |\!)^1_{xy} \parallel \cdots \parallel U'_m \parallel J']$$

(b)

$$S_2 = C_{\widetilde{xy}}[U'_1 \parallel \ldots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{^{v'}/u_2\} \parallel [\![* z(y_2).Q_4]\!] \parallel \ldots \parallel U'_m \parallel J']$$

then, let

$$S_3 = C_{\widetilde{xy}}[U'_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid * y(u_1).Q_2 |\!)^1_{xy} \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{^{v'}/u_2\} \parallel [\![* z(y_2).Q_4]\!] \cdots \parallel U'_m \parallel J']$$

and we can show by Fig. 2.5 that:

$$S'_0 \xrightarrow{\mathsf{RP}_1(w,z)}_1 S_3 \qquad \text{and} \qquad S_2 \xrightarrow{\mathsf{RP}(x,y)}_1 S_3$$

which concludes the proof.

Sub-case $\omega = \mathtt{IO}$**:** Similarly as above.

Sub-case $\omega = \mathtt{SL}$**:** Similarly as above.

Sub-case $\omega = \mathtt{SL}_1$**:** Similarly as above.

Case $\kappa = \mathtt{CD}$**:** As above we distinguish four cases. We only develop Sub-case $\omega = \mathtt{IO}$, as the other cases are similar:

Sub-case $\omega = \mathtt{IO}$**:** Assume that the $\mathtt{IO}$ transition happens on endpoints x and y. Since $\mathtt{CD}$ does not occur on any channels, we do not need to assume more endpoints:

(1) $S = C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(z).Q_2]\!] \parallel \cdots \parallel U_n]$, $n \geq 1$ (Assumption, Fig. 4.7, Lem. 4.30).

(2) $S_1 = C_{\widetilde{xy}}[U_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid y(z).Q_2 |\!)^1_{xy} \parallel \cdots \parallel U_n]$, $n \geq 1$ ((1), Fig. 4.7).

(3) $S_0 = C_{\widetilde{xy}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(z).Q_2]\!] \parallel [\![b?\,(Q_3):(Q_4)]\!] \parallel \cdots \parallel U'_m]$, $m \geq 1$, with $b \in \{\mathtt{tt}, \mathtt{ff}\}$ (IH, Fig. 4.7).

(4) $S_2 = C_{\widetilde{xy}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(z).Q_2]\!] \parallel [\![Q_i]\!] \parallel \cdots \parallel U'_m]$, $i \in \{3,4\}$ ((3), Fig. 4.7).

(5) $S'_0 = C_{\widetilde{xy}}[U'_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid y(z).Q_2 |\!)^1_{xy} \parallel [\![b?\,(Q_3):(Q_4)]\!] \parallel \cdots \parallel U'_m]$, $m \geq 1$, with $b \in \{\mathtt{tt}, \mathtt{ff}\}$ (IH, Fig. 4.7).

Now, let $S_3 = C_{\widetilde{xy}}[U'_1 \parallel \cdots \parallel (\!| x\langle v\rangle.Q_1 \mid y(z).Q_2 |\!)^1_{xy} \parallel [\![Q_i]\!] \parallel \cdots \parallel U'_m]$, $m \geq 1$, with $b \in \{\mathtt{tt}, \mathtt{ff}\}$, $i \in \{3,4\}$. It can be shown, by Fig. 4.7 that:

$$S_2 \xrightarrow{\mathtt{IO}(x,y)}_1 S_3 \qquad \text{and} \qquad S'_0 \xrightarrow{\mathtt{CD}(-)}_1 S_3$$

which, by IH implies that $S_1 \xRightarrow{\gamma_0(\widetilde{xy})\mathtt{CD}(-)}_1 S_3$, concluding the proof.

Sub-case $\omega = \mathtt{RP}$**:** Similarly as above.

Sub-case $\omega = \mathtt{SL}$**:** Similarly as above.

Sub-case $\omega = \mathtt{SL}_1$**:** Similarly as above.

Case $\kappa = \mathtt{SL}_2$**:** Similarly as Case $\omega = \mathtt{IO}_1$.

Case $\kappa = \mathtt{SL}_3$**:** Similarly as Case $\omega = \mathtt{IO}_1$.

□

Lemma 4.48. *For every well-typed $\pi^{\lightning}_{\mathsf{OR}}$ program P and for every sequence of labels $\gamma(\widetilde{xy})$ such that $[\![P]\!] \xRightarrow{\gamma(\widetilde{xy})}_1 S$, there exist Q, S', and $\gamma'(\widetilde{xy})$ such that $P \longrightarrow^* Q$ and $S \xRightarrow{\gamma'(\widetilde{xy})}_1 S'$, with $\gamma'(\widetilde{xy}) = \gamma(\widetilde{xy})\!\downarrow$ (cf. Def. 4.46). Moreover, $[\![Q]\!] \cong_1^{\pi^{\lightning}_{\mathsf{OR}}} S'$.*

Proof (*see Page 148*). By induction on $|\gamma(\widetilde{xy})|$ and a case analysis on the last label of the sequence. The base case is immediate since $[\![P]\!] \xRightarrow{\gamma(\widetilde{xy})}_1 [\![P]\!]$ and $P \longrightarrow^* P$. For the inductive step, assume $|\gamma(\widetilde{xy})| = n \geq 0$. We state the IH:

IH: if $[\![P]\!] \xRightarrow{\gamma(\widetilde{xy})_0}_1 S_0 \xrightarrow{\alpha_{n+1}}_1 S$ then there exists Q_0, S_0' and $\gamma_0'(\widetilde{xy})$ such that $P \longrightarrow^* Q$, $S_0 \xRightarrow{\gamma'(\widetilde{xy})}_1 S_0'$, $\gamma_0'(\widetilde{xy}) = \gamma(\widetilde{xy})\!\downarrow$ and $S_0' \cong_1^{\pi^{\lightning}_{\mathsf{OR}}} [\![Q]\!]$.

Using the IH, the proof can be summarized by the diagram in Fig. 4.8, where we must show the existence of the dotted arrows. Details follow:

Base Case: $n = 0$. Then:

(1) $S \xrightarrow{\omega}_1 S_1$ (Assumption).

(2) $S \xRightarrow{\gamma(\widetilde{xy})}_1 S$ (Assumption)

(3) $S_1 \xRightarrow{\gamma(\widetilde{xy})}_1 S_1$ (Fig. 2.5)

Conclude by letting $S_1 = S_1$, $S_2 = S$ and $S_3 = S_1$ and using (1) and (3).

Inductive Step: $n \geq 1$. We state the IH:

IH: If $S \xrightarrow{\omega}_1 S_1$ and $S \xRightarrow{\gamma_0(\widetilde{xy})}_1 S_0 \xrightarrow{\kappa}_1 S_2$, then there exists S_0' such that $S_1 \xRightarrow{\gamma_0(\widetilde{xy})}_1 S_0'$ and $S_0 \xrightarrow{\omega}_1 S_0'$.

We distinguish cases for $\kappa \in \{\mathtt{IO}_1, \mathtt{RP}_1, \mathtt{CD}, \mathtt{SL}_2, \mathtt{SL}_3\}$. There are five cases and each one has four sub-cases, that correspond to the opening labels in the set $\{\mathtt{IO}, \mathtt{SL}, \mathtt{RP}, \mathtt{SL}_1\}$. We detail three cases: $\kappa = \mathtt{IO}_1$, $\kappa = \mathtt{RP}_1$ and $\kappa = \mathtt{CD}$, as the other are similar:

Case $\kappa = \mathtt{IO}_1$**:** As mentioned above, there are four sub-cases depending on ω. We enumerate them below and only detail $\omega = \mathtt{IO}$, $\omega = \mathtt{RP}$:

Sub-case $\omega = \mathtt{IO}$**:** First assume that the actions take place on endpoints x, y and w, z. This is a general assumption. Furthermore, by typing and Cor. 3.16 and Def. 3.14, it cannot be the case that $x = w$ and $y = z$, because this would imply that there are more than one output prefixed on x. Then, we we only consider the case when $x \neq w$ and $y \neq z$:

Sub-case $x \neq w \wedge y \neq z$: We proceed as follows:

(1) $S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(u_1).Q_2]\!] \parallel \cdots \parallel U_n \parallel J]$ (Assumption, Fig. 4.7, Lem. 4.30).

(2) By Fig. 2.5 and (1)

$$S_1 = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(u_1).Q_2|\!)^1_{xy} \parallel \cdots \parallel U_n \parallel J]$$

(3) By IH, (1), and Lem. 4.30

$$S_0 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(u_1).Q_2]\!] \parallel (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U_m' \parallel J']$$

with $m \geq 1$.

(4) By (3) and Fig. 2.5

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel \cdots \parallel U_m' \parallel J']$$

(5) $S_0 \xrightarrow{\mathtt{IO}(x,y)}_1 S_0'$ (IH)

(6) By Fig. 2.5 and (5)

$$S_0' = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(u_1).Q_2|\!)^1_{xy} \parallel (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U_m' \parallel J']$$

(7) $S_1 \xRightarrow{\gamma_0(\widetilde{x}\widetilde{y})}_1 S_0'$ (IH).

We can then reduce the proof to show that there exists some S_3 such that $S_2 \xrightarrow{\mathtt{IO}(x,y)}_1 S_3$ and $S_0' \xrightarrow{\mathtt{IO}_1(w,z)}_1 S_3$:

(a) Where:

$$S_0' = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(u_1).Q_2|\!)^1_{xy} \parallel (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{xy} \parallel \cdots \parallel U_m' \parallel J']$$

(b) Where:

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel \cdots \parallel U_m' \parallel J']$$

then, let

$$S_3 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(u_1).Q_2|\!)^1_{xy} \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel \cdots \parallel U_m' \parallel J']$$

and we can show by Fig. 2.5 that:

$$S_0' \xrightarrow{\mathtt{IO}_1(w,z)}_1 S_3 \qquad \text{and} \qquad S_2 \xrightarrow{\mathtt{IO}(x,y)}_1 S_3$$

which concludes the proof.

Sub-case $\omega = \mathsf{RP}$**:** As above, assume the actions take place in variables x, y and w, z. It is not possible for them to happen in the same variable, by Cor. 3.16.

(1) $S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \cdots \parallel U_n \parallel J]$ (Assumption, Fig. 4.7, Lem. 4.30).

(2) $S_1 = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel \cdots \parallel U_n \parallel J]$ (Fig. 2.5,(1)).

(3) By IH, (1), and Lem. 4.30

$$S_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U'_m \parallel J']$$

with $m \geq 1$.

(4) By (3) and Fig. 2.5

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{^{v'}/_{u_2}\} \parallel \cdots \parallel U'_m \parallel J']$$

(5) $S_0 \xrightarrow{\mathsf{RP}(x,y)}_1 S'_0$ (IH)

(6) By Fig. 2.5 and (5)

$$S'_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U'_m \parallel J']$$

(7) $S_1 \xRightarrow{\gamma_0(\widetilde{x}\widetilde{y})}_1 S'_0$ (IH).

We reduce the proof to show existence of S_3 such that $S_2 \xrightarrow{\mathsf{RP}(x,y)}_1 S_3$ and $S'_0 \xrightarrow{\mathtt{IO}_1(w,z)}_1 S_3$:

(a) Where:

$$S'_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel (\!|w\langle v'\rangle.Q_3 \mid z(u_2).Q_4|\!)^1_{xy} \parallel \cdots \parallel U'_m \parallel J']$$

(b) Where:

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{^{v'}/_{u_2}\} \parallel \cdots \parallel U'_m \parallel J']$$

then, let

$$S_3 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{^{v'}/_{u_2}\} \parallel \cdots \parallel U'_m \parallel J']$$

and we can show by Fig. 2.5 that:

$$S'_0 \xrightarrow{\mathtt{IO}_1(w,z)}_1 S_3 \qquad \text{and} \qquad S_2 \xrightarrow{\mathsf{RP}(x,y)}_1 S_3$$

which concludes the proof.

Sub-case $\omega = \mathsf{SL}$**:** Similarly as above.

Sub-case $\omega = \mathsf{SL}_1$**:** Similarly as above.

Case $\kappa = \mathsf{RP}_1$**:** We proceed similarly as above. The most interesting case is $\omega = \mathsf{RP}$:

Sub-case $\omega = \mathsf{RP}$**:** As above, assume the actions take place in variables x, y and w, z. It is not possible for them to happen in the same variable, by Cor. 3.16.

(1) $S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \cdots \parallel U_n \parallel J]$ (Assumption, Fig. 4.7, Lem. 4.30).

(2) $S_1 = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel \cdots \parallel U_n \parallel J]$ (Fig. 2.5,(1)).

(3) By IH, (1), and Lem. 4.30

$$S_0 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \\ (\!|w\langle v'\rangle.Q_3 \mid * \, z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U_m' \parallel J']$$

with $m \geq 1$.

(4) By (3) and Fig. 2.5

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \ldots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \\ [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel [\![* \, z(u_2).Q_4]\!] \parallel \ldots \parallel U_m' \parallel J']$$

(5) $S_0 \xrightarrow{\mathsf{RP}(x,y)}_1 S_0'$ (IH)

(6) By Fig. 2.5 and (5)

$$S_0' = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel \\ (\!|w\langle v'\rangle.Q_3 \mid * \, z(u_2).Q_4|\!)^1_{wz} \parallel \cdots \parallel U_m' \parallel J']$$

(7) $S_1 \xRightarrow{\gamma_0(\widetilde{x}\widetilde{y})}_1 S_0'$ (IH).

We reduce the proof to show the existence of S_3 such that $S_2 \xrightarrow{\mathsf{RP}(x,y)}_1 S_3$ and $S_0' \xrightarrow{\mathsf{RP}_1(w,z)}_1 S_3$:

(a) Where:

$$S_0' = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel \\ (\!|w\langle v'\rangle.Q_3 \mid * \, z(u_2).Q_4|\!)^1_{xy} \parallel \cdots \parallel U_m' \parallel J']$$

(b) Where:

$$S_2 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \ldots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![* \, y(u_1).Q_2]\!] \parallel \\ [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel [\![* \, z(y_2).Q_4]\!] \parallel \ldots \parallel U_m' \parallel J']$$

then, let

$$S_3 = C_{\widetilde{x}\widetilde{y}}[U_1' \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid * \, y(u_1).Q_2|\!)^1_{xy} \parallel [\![Q_3]\!] \parallel [\![Q_4]\!]\{v'/u_2\} \parallel \\ [\![* \, z(y_2).Q_4]\!] \cdots \parallel U_m' \parallel J']$$

and we can show by Fig. 2.5 that:

$$S'_0 \xrightarrow{\mathsf{RP}_1(w,z)}_1 S_3 \qquad \text{and} \qquad S_2 \xrightarrow{\mathsf{RP}(x,y)}_1 S_3$$

which concludes the proof.

Sub-case $\omega = \mathtt{IO}$**:** Similarly as above.

Sub-case $\omega = \mathsf{SL}$**:** Similarly as above.

Sub-case $\omega = \mathsf{SL}_1$**:** Similarly as above.

Case $\kappa = \mathsf{CD}$**:** As above we distinguish four cases. We only show Sub-case $\omega = \mathtt{IO}$, as the other cases are similar:

Sub-case $\omega = \mathtt{IO}$**:** Assume that the $\mathtt{IO}$ transition happens on endpoints x and y. Since CD does not occur on any channels, we do not need to assume more endpoints:

(1) $S = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(z).Q_2]\!] \parallel \cdots \parallel U_n]$, $n \geq 1$ (Assumption, Fig. 4.7, Lem. 4.30).

(2) $S_1 = C_{\widetilde{x}\widetilde{y}}[U_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(z).Q_2|\!)^1_{xy} \parallel \cdots \parallel U_n]$, $n \geq 1$ ((1), Fig. 4.7).

(3) $S_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!][\![y(z).Q_2]\!] \parallel [\![b?\,(Q_3):(Q_4)]\!] \parallel \cdots \parallel U'_m]$, $m \geq 1$, with $b \in \{\mathtt{tt}, \mathtt{ff}\}$ (IH, Fig. 4.7).

(4) $S_2 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel [\![x\langle v\rangle.Q_1]\!] \parallel [\![y(z).Q_2]\!] \parallel [\![Q_i]\!] \parallel \cdots \parallel U'_m]$, $i \in \{3,4\}$ ((3), Fig. 4.7).

(5) $S'_0 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(z).Q_2|\!)^1_{xy} \parallel [\![b?\,(Q_3):(Q_4)]\!] \parallel \cdots \parallel U'_m]$, $m \geq 1$, with $b \in \{\mathtt{tt}, \mathtt{ff}\}$ (IH, Fig. 4.7).

Now, let $S_3 = C_{\widetilde{x}\widetilde{y}}[U'_1 \parallel \cdots \parallel (\!|x\langle v\rangle.Q_1 \mid y(z).Q_2|\!)^1_{xy} \parallel [\![Q_i]\!] \parallel \cdots \parallel U'_m]$, $m \geq 1$, with $b \in \{\mathtt{tt}, \mathtt{ff}\}$, $i \in \{3,4\}$. It can be shown, by Fig. 4.7 that:

$$S_2 \xrightarrow{\mathtt{IO}(x,y)}_1 S_3 \qquad \text{and} \qquad S'_0 \xrightarrow{\mathsf{CD}(-)}_1 S_3$$

which, by IH implies that $S_1 \xRightarrow{\gamma_0(\widetilde{x}\widetilde{y})\mathsf{CD}(-)}_1 S_3$, concluding the proof.

Sub-case $\omega = \mathsf{RP}$**:** Similarly as above.

Sub-case $\omega = \mathsf{SL}$**:** Similarly as above.

Sub-case $\omega = \mathsf{SL}_1$**:** Similarly as above.

Case $\kappa = \mathsf{SL}_2$**:** Similarly as Case $\omega = \mathtt{IO}_1$.

Case $\kappa = \mathsf{SL}_3$**:** Similarly as Case $\omega = \mathtt{IO}_1$.

□

C
Chapter 5

C.1 Transforming Translated Terms Into `lcc` via Erasure

Lemma 5.31. *For every well-typed initialized closed network $N = H[P, M]$ such that*

$$P = (\boldsymbol{\nu} x_1, y_1)P_1 \mid \ldots \mid (\boldsymbol{\nu} x_n y_n)P_n$$

the following holds ($k \geq 0$)*:*

1. *If* $[\![P]\!] \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}_1^k S_1$ *then* $[\![P]\!]_f \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}_1^k S_2$ *and* $\delta(S_2) = S_1$.

2. *If* $[\![P]\!]_f \overset{k}{\xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}}{}_1 S$ *then* $[\![P]\!] \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}_1^k \delta(S)$.

Proof (*see Page 179*). We prove each statement individually:

1. By induction on the transition $[\![P]\!] \xrightarrow{\gamma(\widetilde{x}\widetilde{y},-)}{}_1^k S$.

   Base Case: Whenever $[\![P]\!] \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1^0 [\![P]\!]$. We need then to show that $\delta([\![P]\!]_f) = [\![P]\!]$, we will prove this by induction on the structure of P. We only show two cases, as the other follow in the same way:

   Case $P = \mathbf{0}$: Immediate by the definition of $\delta(\cdot)$ (cf. Fig. 5.7), as $\delta([\![\mathbf{0}]\!]_f) = [\![\mathbf{0}]\!] = \overline{\mathtt{tt}}$.

   Case $P = x\langle v\rangle.Q$: Immediate by the definition of $\delta(\cdot)$ (cf. Fig. 5.7). Observe that $\delta([\![x\langle v\rangle.Q]\!]_f) = [\![x\langle v\rangle.Q]\!]$, up to α-conversion:

$$\begin{aligned}
\delta([\![x\langle v\rangle.Q]\!]_f) &= \delta(\overline{\mathsf{snd}(x; v)} \parallel \\
&\qquad \forall\epsilon(ch(x;\epsilon)\,\mathbf{;}\,\{x{:}f_x\} \otimes ch(f_x;\epsilon) \otimes \mathsf{rcv}(f_x, v; \epsilon) \rightarrow [\![Q]\!]_f)) \\
&= \overline{\mathsf{snd}(x, v)} \parallel \forall w_1(\{x{:}f_x\} \otimes \mathsf{rcv}(w_1, v) \rightarrow \delta([\![P]\!]\{^{w_1}/f_x\})) \\
&= [\![x\langle v\rangle.Q]\!]
\end{aligned}$$

Inductive Step: Assume that $[\![P]\!] \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1^{k-1} S_0 \xrightarrow{\alpha(x,y)}{}_1 S_1$ in $k-1 \geq 0$ steps (for $[\![P]\!] \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1^{k-1} S_0$). By IH, $[\![P]\!]_f \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1^{k-1} S_0'$ and $\delta(S_0') = S_0$. We then have to prove that if $S_0 \xrightarrow{\alpha(x,y)}{}_1 S_1$ then $S_0' \xrightarrow{\alpha(x,y)}{}_1 S_2$ and that $\delta(S_2) = S_1$. This is done by a case analysis on $\alpha(x,y)$. There are nine cases corresponding to $\alpha \in \{\mathtt{IO}, \mathtt{SL}, \mathtt{RP}, \mathtt{CD}, \mathtt{IO}_1, \mathtt{RP}_1, \mathtt{SL}_1, \mathtt{SL}_2, \mathtt{SL}_3\}$. We will only show case $\alpha(x,y) = \mathtt{IO}(x,y)$ as it showcases the necessary machinery for solving the other cases.

Case $\alpha(x,y) = \mathtt{IO}(x,y)$**:**

(1) $S_0 = C_{\widetilde{x}\widetilde{y}}[[\![x\langle v\rangle.P_1]\!][\![y(z).P_2]\!] \cdots \parallel U_n]$ (Def. 4.37).

(2) $S_1 = C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^1_{xy} \parallel \cdots \parallel U_n]$ (Def. 4.37).

(3) $S_0' = [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![x\langle v\rangle.P_1]\!]_f \parallel [\![y(z).P_2]\!]_f \parallel \cdots \parallel U_n', \overline{\mathtt{tt}}]$ (Def. 5.25).

(4) By Def. 5.25:

$$S_2 = [\![P]\!]_f = [\![H_{\widetilde{x}\widetilde{y}}]\!][(\!|x\langle v\rangle.P_1 \mid y(z).P_2, f|\!)^1_{xy} \parallel \cdots \parallel U_n', \overline{\mathtt{tt}}]$$

(5) $\delta(S) = C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^1_{xy} \parallel \cdots \parallel U_n]$ (Def. 5.29 to (3), IH).

2. By induction on the transition $[\![P]\!]_f \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1 S$ and case analysis on the label of the last transition.

Base Case: Assume $[\![P]\!]_f \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1^0 [\![P]\!]_f$. The case is immediate as in the base case for Numeral (1).

Inductive Step: By IH, $[\![P]\!]_f \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1^{k-1} S_0$ in $k-1 \geq 0$ steps and $[\![P]\!] \xrightarrow{\gamma(\widetilde{x},\widetilde{y})}{}_1^{k-1} \delta(S_0)$. Then we are left to prove that if $S_0 \xrightarrow{\alpha(x,y)}{}_1 S$ then $\delta(S_0) \xrightarrow{\alpha(x,y)}{}_1 \delta(S)$. We proceed by a case analysis on $\alpha(x,y)$:

Case $\alpha(x,y) = \mathtt{IO}(x,y)$**:**

(1) $S_0 = [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![x\langle v\rangle.P_1]\!]_f \parallel [\![y(z).P_2]\!]_f \parallel \cdots \parallel W_n, \overline{\mathtt{tt}}]$ (Def. 5.25).

(2) $S = [\![H_{\widetilde{x}\widetilde{y}}]\!][(\!|x\langle v\rangle.P_1 \mid y(z).P_2, f|\!)^1_{xy} \parallel \cdots \parallel W_n, \overline{\mathtt{tt}}]$ (Def. 5.25).

(3) $\delta(S_0) = C_{\widetilde{x_1}\widetilde{x_2}}[[\![x\langle v\rangle.P_1]\!] \parallel [\![y(z).P_2]\!] \parallel \cdots \parallel \delta(W_n)]$ (Def. 5.29, IH).

(4) $\delta(S_0) \xrightarrow{\mathtt{IO}(x,y)}{}_1 C_{\widetilde{x}\widetilde{y}}[(\!|x\langle v\rangle.P_1 \mid y(z).P_2|\!)^1_{xy} \parallel \cdots \parallel \delta(W_n)] = S'$ (Def. 4.37, (3)).

(5) $\delta(S) = S'$ (Def. 5.29 to (2)).

□

C.2 Auxiliary Results for Operational Soundness

Lemma 5.36. *Let N be a well-typed closed network. If $[\![N]\!]_f \xRightarrow{\gamma(-,-,\widetilde{a})}{}_1 S$ and $S \not\xrightarrow{\alpha(a)}{}_1$ for any service name a, then there exists N' such that $N \xrightarrow{est}{}^* N' \not\xrightarrow{est}$ and $[\![N']\!]_g = S$ with $f \subseteq g$.*

Proof (*see Page 183*). By Def. 5.12, $N = H[P, M]$, for some P and M. By Lem. 5.11, M is a starting network. We proceed by induction on the maximum number of sessions k that can be established from N:

Base Case: $k = 0$. Therefore, $N \not\xrightarrow{\text{est}}$ and so N is an initialized network (cf. Def. 5.10). Then, by Lem. 5.28, $[\![N]\!]_f \not\xrightarrow{\alpha(a)}_1$, for any a. Thus, $N' = N$.

Inductive Step: $k \geq 1$. Assume that $[\![N]\!]_f \xRightarrow{\gamma(\widetilde{a})}_1 S_0$ and $S_0 \not\xrightarrow{\alpha(a)}_1$ for any service name a where N can establish a maximum number of k sessions. By IH, there exists N_0 such that $N \xrightarrow{\text{est}}{}^r\ N_0 \not\xrightarrow{\text{est}}$ and $S_0 = [\![N_0]\!]_{g_0}$ with $f \subseteq g_0$. We now prove the property for N being able to establish the maximum number of $k{+}1$ sessions (cf. Def. 5.15). Notice that $k + 1$ being the maximum number of sessions that can be established in N, implies that there are $k{+}1$ potentially satisfied requests (cf. Def. 5.15):

(1) $N = H[P, M]$, where N contains $k{+}1$ potentially satisfied requests (Assumption).

(2) $N \xrightarrow{\text{est}} N_1$ and N_1 contains k potentially satisfied requests (Fig. 3.3, Not. 5.14, (1)).

(3) $N = H[P, [\overline{a}^n\langle x\rangle.Q_1]^m \mid [*\, a^\rho(x).Q_2]^n \mid M']$, for some Q_1, Q_2 (By (2)).

(4) $N_1 = H[P \mid Q_1 \mid Q_2, [*\, a^\rho(x).Q_2]^n \mid M']$ (By (2)).

(5) $[\![N]\!]_f = [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_f, [\![[\overline{a}^n\langle x\rangle.Q_1]^m \mid [*\, a^\rho(x).Q_2]^n \mid M']\!]_f]$ (Fig. 5.3, (3)).

(6) $[\![N]\!]_f \xrightarrow{\mathsf{SE}(a)}_1 [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_f, (\![[\overline{a}^n\langle x\rangle.Q_1]^m \mid [*\, a^\rho(y).Q_2]^n, f]\!)^1_{xy} \parallel [\![M']\!]_f] = W$ (Def. 5.25, (5)).

(7) $W \xrightarrow{\mathsf{SE}_1(a)}_1 [\![H_{\widetilde{x}\widetilde{y}}]\!][[\![P]\!]_f, (\![[\overline{a}^n\langle x\rangle.Q_1]^m \mid [*\, a^\rho(y).Q_2]^n, f]\!)^2_{xy} \parallel [\![M']\!]_f] = W'$ (Def. 5.25, (6)).

(8) $W' \xrightarrow{\mathsf{SE}_2(a)}_1 [\![H_{\widetilde{x}x\widetilde{y}y}]\!][[\![P]\!]_{f\cup\{x:y\}} \,|\, [\![Q_1 \,|\, Q_2]\!]_{f\cup\{x:y\}}, [\![[*\, a^\rho(y).Q_2]^n]\!]_{f\cup\{x:y\}} \parallel [\![M']\!]_{f\cup\{x:y\}}] = W_1$ (Def. 5.25, (7)).

(9) $[\![N_1]\!]_{f\cup\{x:y\}} = W_1$, $f \subseteq f \cup \{x{:}y\}$ (By (4),(8)).

(10) There exists N', such that $N_1 \xrightarrow{\text{est}}{}^k\ N' \not\xrightarrow{\text{est}}$ (Lem. 5.16).

(11) There exists S such that $W_1 \xRightarrow{\gamma(\widetilde{a})}_1 S$, $S \not\xrightarrow{\mathsf{SE}(a)}_1$ nor $S \not\xrightarrow{\mathsf{SE}_1(a)}_1$ (By Lem. 5.36, (10)).

(12) $S = [\![N']\!]_g$, $f \cup \{x{:}y\} \subseteq g$ (IH (11), (10), since N_1 has k potentially satisfied requests).

□

Lemma 5.38. *Let S be a target term (Def. 5.22). If $S \xrightarrow{\alpha(a)}_1 S_1$ and $S \xRightarrow{\gamma(\widetilde{x}\widetilde{y},-)}_1 S_2$ then there exists an S_3 such that $S_1 \xRightarrow{\gamma(\widetilde{x}\widetilde{y},-)}_1 S_3$ and $S_2 \xrightarrow{\alpha(a)}_1 S_3$.*

Proof (*see Page 183*). Recalling Not. 5.26, there are three cases: (1) $\alpha(a) = \mathsf{SE}(a)$, (2) $\alpha(a) = \mathsf{SE}_1(a)$ and (3) $\alpha(a) = \mathsf{SE}_2(a)$. We only show Case (1), as Cases (2),

(3) proceed similarly. We proceed by induction on k, defined as the length of transition $S \xRightarrow{\gamma(\widetilde{xy},-)}_1 S_2$, together with a case analysis on the label $\alpha'(x,y)$ of the last action in the transition.

Base Case: $k = 1$. Then, $S \xrightarrow{\alpha(x,y)}_1 S_2$. We apply a case analysis on $\alpha(x,y)$ (cf. Def. 5.25). There are nine sub-cases. We only show one case as the other proceed similarly:

Sub-case $\alpha(x,y) = \mathtt{IO}(x,y)$**:**

(1) $S \xrightarrow{\mathsf{SE}(a)}_1 S_1$ (Assumption).

(2) $S \xrightarrow{\mathtt{IO}(x,y)}_1 S_2$ (Assumption).

(3) By (1), (2) and Def. 5.25:

$$S = [\![H_{\widetilde{xy}}]\!][[\![x\langle v\rangle.P_1]\!]_f \parallel [\![y(z).P_2]\!]_f \parallel S', \\ [\![[\overline{a}^n\langle w\rangle.Q_1]^m]\!]_f \parallel [\![[*\,a^\rho(z).Q_2]^n]\!]_f \parallel S'']$$

(4) By (3) and Def. 5.25:

$$S_1 = [\![H_{\widetilde{xy}}]\!][[\![x\langle v\rangle.P_1]\!]_f \parallel [\![y(z).P_2]\!]_f \parallel S', \\ (\![[\overline{a}^n\langle w\rangle.Q_1]^m \mid [*\,a^\rho(z).Q_2]^n, f]\!)^1_{xy} \parallel S'']$$

(5) By (3) and Def. 5.25:

$$S_2 = [\![H_{\widetilde{xy}}]\!][(\!|x\langle v\rangle.P_1 \mid y(z).P_2, f|\!)^1_{xy} \parallel S', \\ [\![[\overline{a}^n\langle w\rangle.Q_1]^m]\!]_f \parallel [\![[*\,a^\rho(z).Q_2]^n]\!]_f \parallel S'']$$

(6) By (4) and Def. 5.25:

$$S_1 \xrightarrow{\mathtt{IO}(x,y)}_1 [\![H_{\widetilde{xy}}]\!][(\!|x\langle v\rangle.P_1 \mid y(z).P_2, f|\!)^1_{xy} \parallel S', \\ (\!|[\overline{a}^n\langle w\rangle.Q_1]^m \mid [*\,a^\rho(z).Q_2]^n, f|\!)^1_{xy} \parallel S''] = S_3$$

(7) By (4) and Def. 5.25:

$$S_2 \xrightarrow{\mathsf{SE}(a)}_1 [\![H_{\widetilde{xy}}]\!][(\!|x\langle v\rangle.P_1 \mid y(z).P_2, f|\!)^1_{xy} \parallel S', \\ (\!|[\overline{a}^n\langle w\rangle.Q_1]^m \mid [*\,a^\rho(z).Q_2]^n, f|\!)^1_{xy} \parallel S''] = S_3$$

Inductive Step: Assume that $S \xRightarrow{\gamma(\widetilde{xy},-)}_1 S_0 \xrightarrow{\alpha(x,y)}_1 S_2$ in $k > 1$ steps (for the transition $S \xRightarrow{\gamma(\widetilde{xy},-)}_1 S_0$) and we need to prove that the property holds for the last step ($S_0 \xrightarrow{\alpha(x,y)}_1 S_2$). We proceed by a case analysis on $\alpha(x,y)$, which yields 3 cases. As above, for each case, there are nine sub-cases corresponding to all the labels in Def. 5.25 that do not correspond to session establishment (i.e., $\mathtt{IO}(x,y)$, $\mathtt{IO}_1(x,y)$, etc.). Each sub-case proceeds as the sub-case above and concludes by applying the IH.

□

Lemma 5.40. *Let S be a target term. If $S \xRightarrow{\gamma(\widetilde{xy},-)\alpha(a)\gamma_0(\widetilde{x_0 y_0},\widetilde{a_0})}_1 S_1$ then $S \xRightarrow{\gamma_1(\widetilde{xy},-)\alpha(a)\gamma_2(\widetilde{xy},-)\gamma_0(\widetilde{x_0 y_0},\widetilde{a_0})}_1 S_1$, where $\gamma_1(\widetilde{xy},-)\gamma_2(\widetilde{xy},-) = \gamma(\widetilde{xy},-)$.*

Proof (*see Page 184*). By induction on k, the length of sequence $\gamma(\widetilde{xy},-)$, coupled with a case analysis on the last label $\alpha'(x,y)$ in $\gamma(\widetilde{xy},-)$.

Base Case: $k = 1$. Then $\gamma(\widetilde{xy},-) = \alpha'(x,y)$. We apply a case analysis on $\alpha'(x,y)$. There are nine sub-cases, corresponding to the labels that do not involve session establishment in Def. 5.25. We will only show one case, as the others proceed similarly:

Sub-case $\alpha'(x,y) = \mathtt{IO}(x,y)$**:** We distinguish two further Sub-cases, depending on whether $\alpha(a) = \mathsf{SE}(a)$, $\alpha(a) = \mathsf{SE}_1(a)$ or $\alpha(a) = \mathsf{SE}_2(a)$. We only show the former, as the latter two are similar:

(1) $S \xrightarrow{\mathtt{IO}(x,y)}_1 S' \xrightarrow{\mathsf{SE}(a)}_1 S'' \xRightarrow{\gamma_0(\widetilde{x_0 y_0},\widetilde{a_0})}_1 S_1$ (Assumption).

(2) By (1) and Def. 5.25:

$$\begin{aligned} S = \llbracket H_{\widetilde{xy}} \rrbracket [\llbracket x\langle v\rangle.P_1 \rrbracket_f \parallel \llbracket y(z).P_2 \rrbracket_f \parallel S_0, \\ \llbracket [\overline{a}^n\langle w\rangle.Q_1]^m \rrbracket_f \parallel \llbracket [*\, a^\rho(z).Q_2]^n \rrbracket_f \parallel S_0'] \end{aligned}$$

(3) By (1), (2), and Def. 5.25:

$$\begin{aligned} S' = \llbracket H_{\widetilde{xy}} \rrbracket [(\!| x\langle v\rangle.P_1 \mid y(z).P_2, f |\!)^1_{xy} \parallel S_0, \\ \llbracket [\overline{a}^n\langle w\rangle.Q_1]^m \rrbracket_f \parallel \llbracket [*\, a^\rho(z).Q_2]^n \rrbracket_f \parallel S_0'] \end{aligned}$$

(4) By (1), (3), and Def. 5.25:

$$\begin{aligned} S'' = \llbracket H_{\widetilde{xy}} \rrbracket [(\!| x\langle v\rangle.P_1 \mid y(z).P_2, f |\!)^1_{xy} \parallel S_0, \\ (\!| [\overline{a}^n\langle w\rangle.Q_1]^m \mid [*\, a^\rho(z).Q_2]^n, f |\!)^1_{xy} \parallel S_0'] \end{aligned}$$

(5) $S \xrightarrow{\mathsf{SE}(a)}_1 S_1' \xrightarrow{\mathtt{IO}(x,y)}_1 S_1''$ ((2), Rules $\lfloor \mathrm{SE} \rfloor$, $\lfloor \mathrm{IO} \rfloor$ in Fig. 4.7).

(6) By (5) and Def. 5.25:

$$\begin{aligned} S_1' = \llbracket H_{\widetilde{xy}} \rrbracket [\llbracket x\langle v\rangle.P_1 \rrbracket_f \parallel \llbracket y(z).P_2 \rrbracket_f \parallel S_0, \\ (\!| [\overline{a}^n\langle w\rangle.Q_1]^m \mid [*\, a^\rho(z).Q_2]^n, f |\!)^1_{xy} \parallel S_0'] \end{aligned}$$

(7) By (6) and Def. 5.25:

$$\begin{aligned} S_1'' = \llbracket H_{\widetilde{xy}} \rrbracket [(\!| x\langle v\rangle.P_1 \mid y(z).P_2, f |\!)^1_{xy} \parallel S_0, \\ (\!| [\overline{a}^n\langle w\rangle.Q_1]^m \mid [*\, a^\rho(z).Q_2]^n, f |\!)^1_{xy} \parallel S_0'] \end{aligned}$$

(8) $S'' = S_1''$, let $\gamma_1(\widetilde{xy},-)$ be the empty sequence and $\gamma_2(\widetilde{xy},-) = \alpha'(x,y)$ (By (7),(4)).

Inductive Step: $k > 1$. Assume that $S \xRightarrow{\gamma_1'(\widetilde{xy},-)\alpha'(x,y)\alpha(a)\gamma_0(\widetilde{x_0 y_0},\widetilde{a_0})}_1 S_1$ with $\gamma(\widetilde{xy},-) = \gamma_1'(\widetilde{xy},-)\alpha'(x,y)$. By IH, if $S \xRightarrow{\gamma_1'(\widetilde{xy},-)\alpha(a)\alpha'(x,y)\gamma_0(\widetilde{x_0 y_0},\widetilde{a_0})}_1 S_1$

then $S \xRightarrow{\gamma'(\widetilde{x}\widetilde{y},-)\alpha(a)\gamma''(\widetilde{x}\widetilde{y},-)\alpha'(x,y)\gamma_0(\widetilde{x_0}\widetilde{y_0},\widetilde{a_0})}_1 S_1$ with the sequence $\gamma'_1(\widetilde{x}\widetilde{y},-) = \gamma'(\widetilde{x}\widetilde{y},-)\gamma''(\widetilde{x}\widetilde{y},-)$. Thus, we need to prove that $S \xRightarrow{\gamma'_1(\widetilde{x}\widetilde{y},-)\alpha(a)\alpha'(x,y)}_1 S_1$. The proof proceeds as in the base case, by a case analysis on $\alpha'(x,y)$. All the 9 sub-cases proceed in the same way, each one concludes by applying the IH.

□

C.3 Secure Types and The Translation

Theorem 5.51 (Typability of $\llbracket\cdot\rrbracket_f$). *For every well-typed closed network N, the derivation $\vdash_\diamond \llbracket N\rrbracket_f$ holds.*

Proof (see Page 189). By induction on the structure of N. The first cases are in Fig. C.1, which gives the derivation tree for the case $N = [* \, a^\rho_y(x).Q]^m$ and $N = [\overline{a}^m\langle x\rangle.Q]^n$. All the other cases follow.

$$out = \mathsf{o} \qquad \mathrm{loc}_\rho(n;\epsilon) = \mathrm{l}^n_\rho \qquad \mathsf{r}(l) = \mathsf{r}_l \qquad \mathsf{p}(l) = \mathsf{p}_l$$

Case $N = \mathbf{0}$**:** This case is immediate, by Rule $\lfloor$L:Tell$\rfloor$.

Case $N = N_1 \mid N_2$**:** By IH, assume $\vdash_\diamond N_1$ and $\vdash_\diamond N_2$ and thus we can apply Rule $\lfloor$L:Par$\rfloor$ to finish the proof.

Case $N = (\boldsymbol{\nu} xy)P$**:** By induction on the structure of P. There are nine Sub-cases. Sub-cases $P = Q_1 \mid Q_2$, $P = \mathbf{0}$ and $P = (\boldsymbol{\nu} wz)Q$ are immediate by IH. Thus, we only show the 6 remaining cases:

Sub-case $P = x\langle v.Q\rangle$**:** The derivation tree is given. In the derivation tree [1] stands for the application of Rules (L:Comb), (L:Comb), (L:Pred), (L:Pred), (L:Pred), in that order.

$$\dfrac{\dfrac{\text{(L:Tell)}}{\vdash_\diamond \overline{\mathsf{snd}(x;v)}} \qquad \dfrac{\dfrac{\dfrac{\text{IH}}{\vdash_\diamond \llbracket Q\rrbracket_f} \qquad \dfrac{[1]}{\{f_x,v\};\emptyset \vdash_\bullet \{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathrm{rcv}(f_x,v;\epsilon)}}{\vdash_{\mathbf{A}} \forall\epsilon(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathrm{rcv}(f_x,v;\epsilon)\to\llbracket Q\rrbracket_f)}\text{(L:Abs)}}{\vdash_\diamond \forall\epsilon(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathrm{rcv}(f_x,v;\epsilon)\to\llbracket Q\rrbracket_f)}\text{(L:Guard)}}{\vdash_\diamond \overline{\mathsf{snd}(x;v)} \parallel \forall\epsilon(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathrm{rcv}(f_x,v;\epsilon)\to\llbracket Q\rrbracket_f)}\text{(L:Par)}$$

Sub-case $P = x(y).Q$ The derivation tree is given. In the derivation tree [1] stands for the application of Rules (L:Comb), (L:Comb), (L:Pred), (L:Pred), (L:Pred), in that order. Also, [2] = (L:Abs).

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(L:Tell)}}{\vdash_\diamond \overline{\mathrm{rcv}(x,y;\epsilon)}} \qquad \dfrac{\text{IH}}{\vdash_\diamond \llbracket Q\rrbracket_f}}{\vdash_\diamond \overline{\mathrm{rcv}(x,y;\epsilon)} \parallel \llbracket Q\rrbracket_f}\text{(L:Par)} \qquad \dfrac{[1]}{\{f_x\};\{y\} \vdash_\bullet \{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{snd}(f_x;y)}}{\vdash_{\mathbf{A}} \forall y\big(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{snd}(f_x;y)\to\overline{\mathrm{rcv}(x,y;\epsilon)} \parallel \llbracket Q\rrbracket_f\big)}[2]}{\vdash_\diamond \forall y\big(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{snd}(f_x;y)\to\overline{\mathrm{rcv}(x,y;\epsilon)} \parallel \llbracket Q\rrbracket_f\big)}\text{(L:Guard)}$$

Sub-case $P = x \triangleleft l_i.P$**:** The derivation tree is given. In the derivation tree [1] stands for the application of Rules (L:Comb), (L:Comb), (L:Pred), (L:Pred), (L:Pred), in that order.

$$\dfrac{\dfrac{(\text{L:Tell})}{\vdash_\diamond \overline{\mathsf{sel}(x;l)}} \qquad \dfrac{\dfrac{\dfrac{\text{IH}}{\vdash_\diamond [\![Q]\!]_f} \quad \dfrac{[1]}{\{f_x,l\};\emptyset \vdash_\bullet \{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes \mathsf{bra}(f_x,l;\epsilon)}}{\vdash_{\mathbf{A}} \forall\epsilon(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes\mathsf{bra}(f_x,l;\epsilon)\rightarrow[\![Q]\!]_f)}\,(\text{L:Abs})}{\vdash_\diamond \forall\epsilon(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes\mathsf{bra}(f_x,l;\epsilon)\rightarrow[\![Q]\!]_f)}\,(\text{L:Guard})}{\vdash_\diamond \overline{\mathsf{sel}(x;l)} \parallel \forall\epsilon(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes\mathsf{bra}(f_x,l;\epsilon)\rightarrow[\![Q]\!]_f)}\,(\text{L:Par})$$

Sub-case $P = x \triangleright \{l_i : Q_i\}_{i\in I}$**:** The derivation tree is given. In the derivation tree [1] stands for the application of Rules (L:Comb), (L:Comb), (L:Pred), (L:Pred), (L:Pred), in that order. Notice that when using IH, we previously need to use n steps with (L:Par). Also [2] = (L:Guard).

$$\dfrac{\dfrac{D_1 \qquad \dfrac{[1]}{\{f_x\};\{l\}\vdash_\bullet \{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes\mathsf{sel}(f_x;l)}}{\vdash_{\mathbf{A}} \forall l\big(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes\mathsf{sel}(f_x;l)\rightarrow\overline{\mathsf{bra}(x,l;\epsilon)}\parallel \prod_{1\le i\le n} l=l_i\rightarrow[\![Q_i]\!]_f\big)}\,(\text{L:Abs})}{\vdash_\diamond \forall l\big(ch(x;\epsilon)\,;\,\{x{:}f_x\}\otimes ch(f_x;\epsilon)\otimes\mathsf{sel}(f_x;l)\rightarrow\overline{\mathsf{bra}(x,l;\epsilon)}\parallel \prod_{1\le i\le n} l=l_i\rightarrow[\![Q_i]\!]_f\big)}\,[2]$$

and sub-tree D_1 is:

$$\dfrac{\dfrac{(\text{L:Tell})}{\vdash_\diamond \overline{\mathsf{bra}(x,l;\epsilon)}} \qquad \dfrac{\text{IH}}{\vdash_\diamond \prod_{1\le i\le n} l=l_i\rightarrow[\![Q_i]\!]_f}}{\vdash_\diamond \overline{\mathsf{bra}(x,l;\epsilon)}\parallel\prod_{1\le i\le n} l=l_i\rightarrow[\![Q_i]\!]_f}\,(\text{L:Par})$$

Sub-case $P = v?\,(Q):(R)$**:** The proof is immediate as all the variables of an equality are unrestricted; we omit the derivation tree.

□

$$
\dfrac{
  \dfrac{}{\vdash_{\diamond} \overline{\mathrm{o}(\{\langle a,w,n\rangle\}_{\mathsf{p}_m})}}\,(\text{L:Tell})
  \quad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{\text{IH}}{\vdash_{\diamond} \llbracket Q\rrbracket_f}
        \quad
        \dfrac{
          \dfrac{}{\vdash_{\diamond} !\overline{\{x{:}y\}}}\,(\text{L:Tell})
          \quad
          \dfrac{}{\vdash_{\diamond} \overline{\mathrm{o}(\{\langle a,w,x,y\rangle\}_{\mathsf{p}_m})}}\,(\text{L:Tell})
        }{\vdash_{\diamond} \overline{\mathrm{o}(\{\langle a,w,x,y\rangle\}_{\mathsf{p}_m})} \parallel !\overline{\{x{:}y\}}}\,(\text{L:Par})
      }{\vdash_{\diamond} \overline{\mathrm{o}(\{\langle a,w,x,y\rangle\}_{\mathsf{p}_m})} \parallel !\overline{\{x{:}y\}} \parallel \llbracket Q\rrbracket_{f\cup\{x:y\}}}
      \quad
      \dfrac{}{\{n\};\{a,w,y,m\} \vdash_{\bullet} \mathrm{o}(\{\langle a,w,y,m\rangle\}_{\mathsf{p}_n})}\,(\text{L:Pred})
    }{\vdash_{\mathbf{A}} \forall y\big(\mathrm{o}(\mathsf{r}_n)\,;\,\mathrm{o}(\{\langle a,w,y,m\rangle\}_{\mathsf{p}_n}) \to \overline{\mathrm{o}(\{\langle a,w,x,y\rangle\}_{\mathsf{p}_m})} \parallel !\overline{\{x{:}y\}} \parallel \llbracket Q\rrbracket_{f\cup\{x:y\}}\big)}\,(\text{L:Abs})
  }{\vdash_{\diamond} \forall y\big(\mathrm{o}(\mathsf{r}_n)\,;\,\mathrm{o}(\{\langle a,w,y,m\rangle\}_{\mathsf{p}_n}) \to \overline{\mathrm{o}(\{\langle a,w,x,y\rangle\}_{\mathsf{p}_m})} \parallel !\overline{\{x{:}y\}} \parallel \llbracket Q\rrbracket_{f\cup\{x:y\}}\big)}\,(\text{L:Guard})
}{
  \dfrac{\vdash_{\diamond} \overline{\mathrm{o}(\{\langle a,w,n\rangle\}_{\mathsf{p}_m})} \parallel \forall y\big(\mathrm{o}(\mathsf{r}_n)\,;\,\mathrm{o}(\{\langle a,w,y,m\rangle\}_{\mathsf{p}_n}) \to \overline{\mathrm{o}(\{\langle a,w,x,y\rangle\}_{\mathsf{p}_m})} \parallel !\overline{\{x{:}y\}} \parallel \llbracket Q\rrbracket_{f\cup\{x:y\}}\big)}{\vdash_{\diamond} \exists w,x.\,\big(\overline{\mathrm{o}(\{\langle a,w,n\rangle\}_{\mathsf{p}_m})} \parallel \forall y\big(\mathrm{o}(\mathsf{r}_n)\,;\,\mathrm{o}(\{\langle a,w,y,m\rangle\}_{\mathsf{p}_n}) \to \overline{\mathrm{o}(\{\langle a,w,x,y\rangle\}_{\mathsf{p}_m})} \parallel !\overline{\{x{:}y\}} \parallel \llbracket Q\rrbracket_{f\cup\{x:y\}}\big)\big)}\,(\text{L:Local})
}\,(\text{L:Par})
$$

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{\vdash_{\diamond} \overline{\mathrm{o}(\{\langle a,z,x,m\rangle\}_{\mathsf{p}_n})}}\,(\text{L:Tell})
        \quad
        \dfrac{
          \dfrac{\text{IH}}{\vdash_{\diamond} \llbracket Q\rrbracket_f}
          \quad
          \dfrac{}{\{m\};\{a,z,x,y\} \vdash_{\bullet} \mathrm{o}(\{\langle a,z,x,y\rangle\}_{\mathsf{p}_m})}\,(\text{L:Pred})
        }{\vdash_{\diamond} \forall y(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,x,y\rangle\}_{\mathsf{p}_m}) \to \llbracket P\rrbracket_{f\cup\{x:y\}})}\,[1]
      }{\vdash_{\diamond} \overline{\mathrm{o}(\{\langle a,z,x,m\rangle\}_{\mathsf{p}_n})} \parallel \forall y(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,x,y\rangle\}_{\mathsf{p}_m}) \to \llbracket P\rrbracket_{f\cup\{x:y\}})}\,[2]
      \quad
      \dfrac{
        \dfrac{}{\{m\};\{a,n,z\} \vdash_{\bullet} \mathrm{o}(\{\langle a,z,n\rangle\}_{\mathsf{p}_m})}\,(\text{L:Pred})
        \quad
        \dfrac{}{\emptyset;\{n,\rho\} \vdash_{\bullet} 1^n_\rho}\,(\text{L:Pred})
      }{\{m\};\{a,z,n,\rho\} \vdash_{\bullet} \mathrm{o}(\{\langle a,z,n\rangle\}_{\mathsf{p}_m}) \otimes 1^n_\rho}\,[3]
    }{\vdash_{\mathbf{A}} \forall z,n\big(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,n\rangle\}_{\mathsf{p}_m}) \otimes 1^n_\rho \to \overline{\mathrm{o}(\{\langle a,z,x,m\rangle\}_{\mathsf{p}_n})} \parallel \forall y(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,x,y\rangle\}_{\mathsf{p}_m}) \to \llbracket P\rrbracket_{f\cup\{x:y\}}\,)\big)}\,[4]
  }{\vdash_{\diamond} \forall z,n\big(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,n\rangle\}_{\mathsf{p}_m}) \otimes 1^n_\rho \to \overline{\mathrm{o}(\{\langle a,z,x,m\rangle\}_{\mathsf{p}_n})} \parallel \forall y(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,x,y\rangle\}_{\mathsf{p}_m}) \to \llbracket P\rrbracket_{f\cup\{x:y\}}\,)\big)}\,[5]
}{\vdash_{\diamond} !\,\big(\exists x.\big(\forall z,n\big(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,n\rangle\}_{\mathsf{p}_m}) \otimes 1^n_\rho \to \overline{\mathrm{o}(\{\langle a,z,x,m\rangle\}_{\mathsf{p}_n})} \parallel \forall y(\mathrm{o}(\mathsf{r}_m)\,;\,\mathrm{o}(\{\langle a,z,x,y\rangle\}_{\mathsf{p}_m}) \to \llbracket P\rrbracket_{f\cup\{x:y\}}\,)\big)\big)\big)}\,[6]
$$

Figure C.1: Typing derivations for $\llbracket [a^\rho(x).P]^m \rrbracket$ and $[\overline{a}^m\langle x\rangle.Q]^n$, as used in the proof of Thm. 5.51. The missing rules are numbered: [1] = (L:Abs), [2] = (L:Par), [3] = (L:Comb), [4] = (L:Abs), [5] = (L:Guard), [6] = (L:Repl),(L:Local).

D
Chapter 7

D.1 Auxiliary Results for Operational Correspondence

Lemma 7.19. *For every $\pi_{\mathsf{R}}^{\sharp}$ pre-redex or inaction process P and every target term S, it holds that $S = init(\llbracket P \rrbracket_f^g)$ implies that either:*

1. *$S = \llbracket P \rrbracket_f^g$ (or)*
2. *there exists a declaration context $D_{\widetilde{x}}$ such that $D_{\widetilde{x}}[S] = \llbracket P \rrbracket_f^g$, for some g and f and $\widetilde{x} = (\mathsf{fv}(S) \cap \mathsf{fv}(\llbracket P \rrbracket_g^f)) \setminus (ran(f) \cup ran(g))$.*

Proof (*see Page 208*). By applying a case analysis on T. We only show the case for: $S = init(\llbracket x\langle v\rangle.Q \rrbracket_f^g)$ since the other cases are either immediate ($T = init(\llbracket x(y).Q \rrbracket_f^g)$, $S = init(\llbracket x \triangleright \{l_i : Q_i\}_{i \in I} \rrbracket_f^g)$ and $S = init(\llbracket \mathbf{0} \rrbracket_f^g)$) or similar ($S = init(\llbracket * \, x(y).Q \rrbracket_f^g)$ and $T = init(\llbracket x \triangleleft l.Q \rrbracket_f^g)$).

(1) By Assumption, $P = x\langle v\rangle.Q$.

(2) By Assumption, $S = init(P)$.

(3) By Fig. 7.2 and (1):

$$\begin{aligned}\llbracket P \rrbracket_g^f = {} & \texttt{signal}\ x', x''\ \texttt{in} \\ & \texttt{do loop}\ (\texttt{emit}\ f_x\ (f_v, g_v, x', x''); \texttt{pause}) \\ & \ \texttt{until}\ g_x \to \llbracket Q \rrbracket_{f,\{x \leftarrow x'\}}^{g,\{x' \leftarrow x''\}}\end{aligned}$$

(4) Def. 7.16 and (2):

$$S = \texttt{do loop}\ (\texttt{emit}\ f_x\ (f_v, x', x''); \texttt{pause})\ \texttt{until}\ g_x \to \llbracket Q \rrbracket_{f,\{x \leftarrow x'\}}^{g,\{x' \leftarrow x''\}}$$

(5) By the Def. of $\mathsf{fv}(\cdot)$ and (4), $\mathsf{fv}(S) = \{f_x, g_x, x', x'', f_v\} \cup \mathsf{fv}([\![Q]\!]^{g,\{x'\leftarrow x''\}}_{f,\{x\leftarrow x'\}})$.

(6) By the Def. of $\mathsf{fv}(\cdot)$ and (3), $\mathsf{fv}([\![P]\!]^{g}_{f}) = \{f_x, g_x, f_v\} \cup \mathsf{fv}([\![Q]\!]^{g,\{x'\leftarrow x''\}}_{f,\{x\leftarrow x'\}})$.

(7) By applying $\cap$ to (5) and (6), $\mathsf{fv}([\![P]\!]^{g}_{f}) \cap \mathsf{fv}(S) = \{x', x''\} = \widetilde{x}$.

(8) Let $D_{\widetilde{x}}[\cdot] = \texttt{signal } x', x'' \texttt{ in } [\cdot]$ by Def. 2.40 and (7).

(9) Finally, by applying $=$ to (8) and (4):

$$\begin{aligned} D_{\widetilde{x}}[S] = \; & \texttt{signal } x', x'' \texttt{ in} \\ & \texttt{do loop (emit } f_x\ (f_v, g_v, x', x'')\texttt{; pause) until } g_x \\ & \;\to [\![Q]\!]^{g,\{x'\leftarrow x''\}}_{f,\{x\leftarrow x'\}} \\ = \; & [\![P]\!]^{g}_{f} \end{aligned}$$

$\square$

Lemma 7.23. *Let R be a well-typed $\pi^{\sharp}_{\mathsf{R}}$ redex enabled by $\widetilde{x}, \widetilde{y}$. If $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R]\!]^{g}_{f} \longmapsto S$ then there exists R' such that $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})R'$ and the following holds:*

1. *$S \hookrightarrow_{\mathsf{R}} [\![R']\!]^{g'}_{f'}$ for some f', g' such that $\widetilde{x}\widetilde{y} \in dom(f')$ and $\widetilde{x}\widetilde{y} \in dom(g')$ and*

2. *there exists $D_{\widetilde{z}}$ such that $D_{\widetilde{z}}[S] \equiv_{\alpha} [\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R']\!]^{g'}_{f'}$ for some $\widetilde{z}$.*

Proof (see Page 210). By a case analysis on the possible redexes. For every case we first show Item 1 and then Item 2. There are three cases:

(a) $x\langle v\rangle.P \mid y(z).Q$,

(b) $x\langle v\rangle.P \mid *\, y(z).Q$, and

(c) $x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I}$, with $j \in I$.

We only show Case (a) as the others proceed similarly. Assume $R = x\langle v\rangle.P \mid y(z).Q$. Also, assume that f_x, g_x, g_y and g_y are in the signal environment, since they are emitted by the processes in the translation. First we prove Item (1): we show the derivation of the big-step reduction, obtained from the rules in Fig. 2.8.

$$\lfloor\textsc{Sig-Dec}\rfloor\;\dfrac{\lfloor\textsc{Let-Par}\rfloor\;\dfrac{\lfloor\textsc{Sig-Dec}\rfloor\;\dfrac{\lfloor\text{DU-P}\rfloor\;\dfrac{\begin{array}{c}\lfloor\textsc{Lp-Stu}\rfloor, \lfloor\textsc{L-Done}\rfloor\\ \lfloor\textsc{Emit}\rfloor, \lfloor\textsc{Pause}\rfloor\end{array}}{S \longmapsto [\![P]\!]^{g_2}_{f_2}}}{[\![x\langle v\rangle.P]\!]^{g_1}_{f_1} \longmapsto [\![P]\!]^{g_2}_{f_2}} \qquad D_1}{[\![(x\langle v\rangle.P \mid y(z).Q)]\!]^{f_1}_{g_1} \longmapsto [\![P]\!]^{g_2}_{f_2} \parallel [\![Q]\!]^{g_3}_{f_3}\{v/z\}}}{[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(x\langle v\rangle.P \mid y(z).Q)]\!]^{f}_{g} \longmapsto [\![P]\!]^{g_2}_{f_2} \parallel [\![Q]\!]^{g_3}_{f_3}\{v/z\}}$$

where D_1 is given below:

$$\lfloor\textsc{L-Done}\rfloor\;\dfrac{\begin{array}{c}\lfloor\text{DW-}\textsc{Int}\rfloor, \lfloor\textsc{Emit}\rfloor\\ \lfloor\text{DU-P}\rfloor, \lfloor\textsc{L-Stu}\rfloor, \lfloor\textsc{Pause}\rfloor\end{array}}{[\![y(z).Q]\!]^{g_1}_{f_1} \longmapsto [\![Q]\!]^{g_3}_{f_3}\{v/z\}}$$

and $S = \texttt{do}\ (\texttt{loop}\ (\texttt{emit}\ w\ (v, g_v, x', c'); \texttt{pause}\ ))\ \texttt{until}\ c \ \rightarrow\ ([\![P]\!]^{g_1,\{x' \leftarrow c'\}}_{f_1,\{x \leftarrow x'\}})$, $f_1 = f, \{\widetilde{x} \leftarrow \widetilde{w}, \widetilde{y} \leftarrow \widetilde{w}\}$, $g_1 = g, \{\widetilde{x} \leftarrow \widetilde{c}, \widetilde{y} \leftarrow \widetilde{c}\}$, $f_2 = f_1, \{x \leftarrow x'\}$, $g_2 = g_1, \{x \leftarrow c'\}$, $f_3 = f_2$, and $g_3 = g_2$ (since r, r' in $\texttt{await}\ f_y(z, r, r')\ \texttt{in}\ [\![Q]\!]^{g_3,\{y \leftarrow c, z \leftarrow c\}}_{f_3,\{y \leftarrow r, z \leftarrow z\}}$ have been replaced by the emitted values x' and c' resp.). Thus, $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(x\langle v\rangle.P \mid y(z).Q)]\!]^f_g \longmapsto [\![P]\!]^{g_2}_{f_2} \parallel [\![Q]\!]^{g_2}_{f_2}\{^v\!/z\}$. Furthermore, by Thm. 7.9: $[\![P]\!]^{g_2}_{f_2} \parallel [\![Q]\!]^{g_2}_{f_2}\{^v\!/z\} = [\![P \mid Q\{^v\!/z\}]\!]^{g_2 \odot \{^v\!/z\}}_{f_2 \odot \{^v\!/z\}} = S$. Then, by Rule $\lfloor\text{Com}\rfloor$, $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(x\langle v\rangle.P \mid y(z).Q) \longrightarrow (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P \mid Q\{^v\!/z\})$, with $R' = P \mid Q\{^v\!/z\}$. Thus, letting $f' = f_2 \odot \{^v\!/z\}, g' = g_2 \odot \{^v\!/z\}$, we conclude: $S \hookrightarrow_{\mathsf{R}} [\![R']\!]^{g'}_{f'}$.

To prove Item (2), we have to show the existence of $D_{\widetilde{z}}[\cdot]$. To do this, it is enough to prove that there exists a sequence $\widetilde{z}$ that satisfies the properties in the statement. Let $\widetilde{z} = f'_{\widetilde{x}} g'_{\widetilde{x}}$. By Def. 2.40, $D_{\widetilde{z}}[[\![R']\!]^{f'}_{g'}] = \texttt{signal}\ f'_{\widetilde{x}} g'_{\widetilde{x}}\ \texttt{in}\ [\![R']\!]^{g'}_{f'}$. Furthermore, by Lem. 7.20, $[\![(\boldsymbol{\nu}\widetilde{x}\widetilde{y})R']\!]^{g'}_{f'} = \texttt{signal}\ \widetilde{w}, \widetilde{c}\ \texttt{in}\ [\![R']\!]^{g',\{\widetilde{x} \leftarrow \widetilde{c}, \widetilde{y} \leftarrow \widetilde{c}\}}_{f',\{\widetilde{x} \leftarrow \widetilde{w}, \widetilde{y} \leftarrow \widetilde{w}\}}$. Finally, by α-conversion, $\texttt{signal}\ f'_{\widetilde{x}} g'_{\widetilde{x}}\ \texttt{in}\ [\![R']\!]^{f'}_{g'} \equiv_\alpha \texttt{signal}\ \widetilde{w}, \widetilde{c}\ \texttt{in}\ [\![R']\!]^{g',\{\widetilde{x} \leftarrow \widetilde{c}, \widetilde{y} \leftarrow \widetilde{c}\}}_{f',\{\widetilde{x} \leftarrow \widetilde{w}, \widetilde{y} \leftarrow \widetilde{w}\}}$.

□

Lemma 7.24. *Let $P = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P_1 \mid \ \dots\ \mid P_n)$, $n \geq 1$ be a well-typed $\pi^{\sharp}_{\mathsf{R}}$ program. Then, the following holds: for every S such that $[\![P]\!]^g_f \longmapsto^* S$, it holds that (1) $S = S_1 \parallel \cdots \parallel S_m$, $m \geq n$, (2) there exists $P' = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ \dots\ \mid P'_m)$ such that $P \longrightarrow^* P'$ and (3) for every $1 \leq i \leq m$ there exist f', g' such that either:*

1. *$S_i \hookrightarrow_{\mathsf{R}} init([\![P'_i]\!]^{g'}_{f'})$, for some pre-redex P'_i or*

2. *$S_i \hookrightarrow_{\mathsf{R}} [\![P'_i]\!]^{g'}_{f'}$, such that P'_i is a pre-redex, $P'_i = v?\,(Q_1) \!:\! (Q_2)$, or $P'_i = \mathbf{0}$.*

Proof (see Page 210). By Cor. 3.34, for every $1 \leq i \leq n$, P_i is a pre-redex or $P_i = v?\,(P'_i) \!:\! (P''_i)$. We apply induction on the length r of transition $[\![P]\!]^g_f \longmapsto^* S$. The base case is $[\![P]\!]^g_f \longmapsto^* [\![P]\!]^g_f$, which is immediate. We show the inductive step. First, the IH:

IH1: If $[\![P]\!]^g_f \longmapsto^* S'_0$ in $r - 1$ big-step reductions then $S'_0 = S'_1 \parallel \cdots \parallel S'_m$ and for every $1 \leq i \leq m$ either:

(a) $S'_i \hookrightarrow_{\mathsf{R}} init([\![P'_i]\!]^{g'}_{f'})$, for some pre-redex P'_i (or)

(b) $S'_i \hookrightarrow_{\mathsf{R}} [\![P'_i]\!]^{g'}_{f'}$, such that P'_i is a pre-redex or $P'_i = v?\,(Q_1) \!:\! (Q_2)$,

for some f', g', and there exists P'_0 such that $P \longrightarrow^* P'_0 = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ \dots\ \mid P'_m)$.

Then, we need to prove the property for $[\![P]\!]^g_f \longmapsto^* S'_0 \longmapsto S$, focusing on transition $S \longmapsto S'$. We prove this by induction on the number m of S'_i processes in S'_0:

Base Case: $m = 1$. Assuming that $S'_0 = S'_i$ then we apply a case analysis on S'_i. There are two main cases:

1. $S'_i \hookrightarrow_{\mathsf{R}} init([\![P'_i]\!]^{g'}_{f'})$, for some pre-redex P'_i: Immediate by Cor. 7.17.

2. $S'_i \hookrightarrow_{\mathsf{R}} [\![P'_i]\!]^{g'}_{f'}$, such that P'_i is a pre-redex or $P'_i = v?\,(Q_1) \!:\! (Q_2)$: There are six sub-cases. The first five that correspond to pre-redexes are immediate by Cor. 7.17 (since there is only one process in parallel). The last case corresponds to $P'_i = v?\,(Q_1) \!:\! (Q_2)$, which proceeds by Fig. 2.8.

Inductive Step: For the inductive step let $m \geq 2$ and we prove for m processes. The IH is as follows:

IH2: If $S'_1 \parallel \cdots \parallel S'_{m-1} \longmapsto S = S_1 \parallel \cdots \parallel S_s$, $s \geq m-1$ then for every $1 \leq j \leq s$ either:

(a) $S_j \hookrightarrow_{\mathsf{R}} init(\llbracket P''_j \rrbracket^{g''}_{f''})$, for some pre-redex P''_j (or)

(b) $S_j \hookrightarrow_{\mathsf{R}} \llbracket P''_j \rrbracket^{g''}_{f''}$, such that P''_j is a pre-redex or $P''_j = v?\,(Q_1)\!:\!(Q_2)$,

for some f'', g'', and there exists P' such that

$$P'_0 \longrightarrow^* P' = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P''_1 \mid \ldots \mid P''_s)$$

We then need to prove the statement for $S'_1 \parallel \cdots \parallel S'_{m-1} \parallel S'_m \longmapsto S_1 \parallel \cdots \parallel S_s$. We proceed by using a case analysis on S'_m. There are eleven cases:

- Three cases correspond to $S'_m = init(\llbracket P'_j \rrbracket^{g'}_{f'})$ for $P'_j = x\langle v\rangle.Q$, $P'_j = x \triangleleft v.Q$ and $*\,x(y).Q$.
- Six cases correspond to $S'_m = \llbracket P'_j \rrbracket^{g'}_{f'}$ for P'_j being a pre-redex.
- Two correspond to $S'_m = \llbracket \mathbf{0} \rrbracket^{g'}_{f'}$ and $S'_m = \llbracket v?\,(Q_1)\!:\!(Q_2) \rrbracket^{g'}_{f'}$.

We show one case: $S'_m = init(\llbracket x\langle v\rangle.Q \rrbracket^{g'}_{f'})$. The other cases proceed similarly:

Case $S'_m = init(\llbracket x\langle v\rangle.Q \rrbracket^{g'}_{f'})$: By assumption, $S'_1 \parallel \cdots \parallel S'_{m-1} \parallel S'_m = S'_1 \parallel \cdots \parallel S'_{m-1} \parallel init(\llbracket x\langle v\rangle.Q \rrbracket^{g'}_{f'})$, we identify two sub-cases: whenever there exists a S'_k, $1 \leq k \leq m-1$ such that

$$S'_k = \llbracket y(z).R \rrbracket^{g'}_{f'}$$

and $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(x\langle v\rangle.Q \mid y(z).R)$ is a redex enabled by $\widetilde{x}\widetilde{y}$, and whenever such S'_k does not exist. Notice that by typability and Thm. 3.28, we ensure that $y(z).R$ is unique in process $(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid P'_{m-1})$. Therefore, there will not be multiple translated processes that react to the signal emitted by S'_m. We show the first case, as the other follows by Cor. 7.17.

Sub-case $\exists S'_k = \llbracket y(z).R \rrbracket^{g'}_{f'}$: (1) $\vdash P$ (Assumption).

(2) $\llbracket P \rrbracket^{f}_{g} \longmapsto S'_0 = S'_1 \parallel \ldots \parallel \llbracket y(z).R \rrbracket^{g'}_{f'} \parallel \ldots \parallel S'_{m-1}$ (IH1, Assumption).

(3) $P \longrightarrow^* P' = (\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ldots \mid y(z).R \mid \ldots \mid P'_{m-1})$ ((1), (2), IH1).

(4) $S'_0 \longmapsto S = S_1 \parallel \ldots \parallel S_s$, $s \geq m-1$ (Assumption)

(5) $\forall 1 \leq j \leq s.(S'_j \hookrightarrow_{\mathsf{R}} init(\llbracket P'_j \rrbracket^{g'}_{f'}) \wedge S_j \hookrightarrow_{\mathsf{R}} \llbracket P''_j \rrbracket^{g''}_{f''})$ (IH2).

(6) $\vdash P'$ (Thm. 3.26, (1), (3)).

(7) $S = S_1 \parallel \cdots \parallel \llbracket y(z).R \rrbracket^{g'}_{f'} \parallel \cdots \parallel S_s$ ((6), Def. 3.27, (2)).

(8) $S'_0 \parallel S'_m = S'_1 \parallel \ldots \parallel \llbracket y(z).R \rrbracket^{g'}_{f'} \parallel \ldots \parallel S'_{m-1} \parallel S'_m$ (Assumption, (7)).

(9) $S'_0 \parallel S'_m \longmapsto S_1 \parallel \ldots \parallel [\![R]\!]^{g''}_{f''}\{v/z\} \parallel \ldots \parallel S_{m-1} \parallel [\![Q]\!]^{g''}_{f''}$ (Fig. 2.8, Lem. 7.23, (4), and (5)).

(10) Finally, by Fig. 2.1 and (3):

$$\begin{aligned} &(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P'_1 \mid \ \ldots \ \mid y(z).R \mid \ \ldots \ \mid P'_{m-1} \mid x\langle v\rangle.Q) \\ \longrightarrow\ &(\boldsymbol{\nu}\widetilde{x}\widetilde{y})(P''_1 \mid \ldots \mid R\{v/x\} \mid \ldots \mid P''_{m-1} \mid Q) = P' \end{aligned}$$

□

E
Chapter 8

E.1 Auxiliary Results for Operational Correspondence

Lemma 8.12. *Let $\mathcal{H}(\widetilde{k})$ be a process handler as in Def. 8.1. Then, if $\langle \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k}) \diamond \Sigma\rangle \dashmapsto^{*} \langle T \diamond \Sigma'\rangle$ for some Σ', it holds that:*

1. *$T = T_1 \parallel \cdots \parallel T_n$ with $n \geq 1$*
2. *There exists $D_{\widetilde{z}}, D_{\widetilde{z_1}}, \ldots, D_{\widetilde{z_n}}$ such that*

$$D_{\widetilde{z}}[D_{\widetilde{z_1}}[T_1] \parallel \cdots \parallel D_{\widetilde{z_n}}[T_n]] \equiv_{\mathsf{R}} \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k})$$

Proof (*see Page 231*). By induction on the length m of $\widetilde{k}$. The base case is $m = 2$ since by typing our translation will not allow handler processes with a single channel. By Def. 8.1:

$$K = \texttt{signal}\ k, \overline{k}, ack_i^k, ack_o^k, ack_i^{\overline{k}}, ack_o^{\overline{k}}\ \texttt{in}\ I(k) \parallel O(k) \parallel I(\overline{k}) \parallel O(\overline{k})$$

Then, by Lem. 8.11, there exists $k', k'', \overline{k'}, \overline{k''}, \Sigma'$ such that one of the following cases holds:

(1) $K \dashmapsto^{*} \langle I(k') \parallel O(k'') \parallel I(\overline{k''}) \parallel O(\overline{k'}) \diamond \Sigma'\rangle$

(2) $K \dashmapsto^{*} \langle I(k') \parallel init(O(k)) \parallel init(I(\overline{k})) \parallel O(\overline{k'}) \diamond \Sigma'\rangle$

(3) $K \dashmapsto^{*} \langle init(I(k)) \parallel O(k') \parallel I(\overline{k'}) \parallel init(O(\overline{k})) \diamond \Sigma'\rangle$

(4) $K \dashmapsto^{*} \langle init(I(k)) \parallel init(O(k)) \parallel init(I(\overline{k})) \parallel init(O(\overline{k})) \diamond \Sigma'\rangle$

We only show case (2), as cases (1), (3) and (4) are similar.

Assuming that $K \dashrightarrow^* \langle I(k') \parallel init(O(k)) \parallel init(I(\overline{k})) \parallel O(\overline{k'}) \diamond \Sigma' \rangle$. We consider the free signal names of each component:

$$\begin{aligned} \mathsf{fv}(I(k')) =& \{ack_o^{\overline{k}}, ack_i^{k}, k'\} \\ \mathsf{fv}(O(\overline{k'})) =& \{ack_o^{\overline{k}}, ack_i^{k}, k'\} \\ \mathsf{fv}(init(I(\overline{k}))) =& \{ack_o^{k}, ack_i^{\overline{k}}, \overline{k}\} \\ \mathsf{fv}(init(O(k))) =& \{ack_o^{k}, ack_i^{\overline{k}}, \overline{k}, \overline{k''}, k''\} \end{aligned}$$

We then prove the existence of $\widetilde{z}$, $\widetilde{z_1}$, $\widetilde{z_2}$, $\widetilde{z_3}$, and $\widetilde{z_4}$, which in turn shows the existence of the required signal declaration contexts. To calculate $\widetilde{z}$, we take the names in the union of the intersections of complementary handler components:

$$\widetilde{z} = (\mathsf{fv}(I(k')) \cap \mathsf{fv}(O(\overline{k'}))) \cup (\mathsf{fv}(init(I(\overline{k}))) \cap \mathsf{fv}(init(O(k))))$$

To calculate the other variable we just subtract $\widetilde{z}$ from their corresponding free signal names. Also notice that by Def. 8.10, $init(I(\overline{k})) = I(\overline{k})$, up-to its unfolding (as it is a recursive process). Thus, we obtain process:

$$\begin{aligned} & D_{\widetilde{z}}[D_{\widetilde{z_1}}[I(k')] \parallel D_{\widetilde{z_2}}[O(\overline{k'})] \parallel D_{\widetilde{z_3}}[init(I(\overline{k}))] \parallel D_{\widetilde{z_4}}[init(O(k))]] \\ & = \mathtt{signal}\ k', \overline{k}, ack_i^{k}, ack_o^{k}, ack_i^{\overline{k}}, ack_o^{\overline{k}}\ \mathtt{in}\ I(k') \parallel \\ & \quad O(\overline{k'}) \parallel I(\overline{k}) \parallel \mathtt{signal}\ k'', \overline{k''}\ \mathtt{in}\ init(O(k)) = R \end{aligned}$$

Observe that $\mathtt{signal}\ k'', \overline{k''}\ \mathtt{in}\ init(O(k)) = O(k)$, up-to its unfolding. Furthermore, since k' is now bound in R, we state that $R \equiv_{\mathsf{R}} \mathtt{signal}\ \widetilde{k}\ \mathtt{in}\ \mathcal{H}(\widetilde{k})$ up-to the renaming of bound variables. The inductive step proceeds by applying the IH and following an argument as the one presented above. $\square$

Lemma 8.13. *For every well-typed* aπ *program* $C[P,Q] = (\boldsymbol{\nu}\widetilde{k})(P \parallel Q)$ *the following holds: If* $(\!|C[P,Q]|\!) \dashrightarrow^* K$*, then* $K = \langle T_1 \parallel \cdots \parallel T_n \parallel T_{n+1} \parallel \cdots \parallel T_m \diamond \Sigma \rangle$, $1 \leq n \leq m$, *where:*

1. *There exists* R *such that* $P \ggg\!\!\rightarrow^* R = C[P',Q']$ *and* $\Sigma = \delta(Q')$.
2. *There exist contexts* $D_{\widetilde{z}}, D_{\widetilde{z_{n+1}}}, \ldots, D_{\widetilde{z_m}}$ *such that: (a)* $D_{\widetilde{z}}[D_{\widetilde{z_{n+1}}}[T_{n+1}] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]] \equiv_{\mathsf{R}} \mathtt{signal}\ \widetilde{k}\ \mathtt{in}\ \mathcal{H}(\widetilde{k})$*, and (b)* $T_1 \parallel \cdots \parallel T_n = [\![P']\!]$.
3. *There exist* $\widetilde{s}$ *and* $\widetilde{s'}$ *such that* $\widetilde{z} = \widetilde{s}\widetilde{s'}$ *and* $\langle D_{\widetilde{s}}[T_1 \parallel \cdots \parallel T_n \parallel D_{\widetilde{s'}}[D_{\widetilde{z_{n+1}}}[T_{n+1}] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]]] \diamond \Sigma \rangle \equiv_{\mathsf{R}} (\!|C[P',Q']|\!)$.

Proof (see Page 232). By induction on the length m of the reduction $(\!|C[P,Q]|\!) \dashrightarrow^* K$. The base case is immediate. We show the inductive step.

Let $(\!|C[P,Q]|\!) \dashrightarrow^* K_0 \dashrightarrow K$. By IH1, $K_0 = \langle S_1 \parallel \cdots \parallel S_n \parallel S_{n+1} \parallel \cdots \parallel S_m \diamond \Sigma_0 \rangle$ where the following holds:

1. There exists $C[P_0,Q_0]$ such that $C[P,Q] \ggg\!\!\rightarrow^* C[P_0,Q_0]$ and $\Sigma_0 = \delta(Q_0)$.
2. There exist $D_{\widetilde{z'}}, D_{\widetilde{z'_{n+1}}}, \ldots, D_{\widetilde{z'_m}}$ such that:

(a) $D_{\widetilde{z'}}[D_{\widetilde{z'_{n+1}}}[S_{n+1}] \parallel \cdots \parallel D_{\widetilde{z'_m}}[S_m]] = \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k})$ and

(b) $S_1 \parallel \cdots \parallel S_n = [\![P_0]\!]$.

3. There exist $\widetilde{s_0}$ and s'_0 such that $\widetilde{z'} = \widetilde{s_0}\widetilde{s'_0}$ and

$$\langle D_{\widetilde{s_0}}[S_1 \parallel \cdots \parallel S_n \parallel D_{\widetilde{s'_0}}[D_{\widetilde{z_{n+1}}}[S_{n+1}] \parallel \cdots \parallel D_{\widetilde{z_m}}[S_m]]] \diamond \Sigma_0\rangle \equiv_{\mathsf{R}} (\!|C[P_0, Q_0]|\!)$$

We now focus on the big-step reduction $K_0 \longmapsto K$. Since $K_0 = \langle S_1 \parallel \cdots \parallel S_n \parallel S_{n+1} \parallel \cdots \parallel S_m \diamond \Sigma_0\rangle$, we apply induction on the number n of S_r processes:

Base Case: $n = 1$. By assumption, $K_0 = \langle S_1 \parallel S_2 \parallel \cdots \parallel S_m \diamond \Sigma_0\rangle$. By IH1, there exists $C[P_0, Q_0]$ such that $C[P, Q] \ggg\!\!\rightarrow^* C[P_0, Q_0]$, $\Sigma_0 = \delta(Q_0)$, $S_1 = [\![P_0]\!]$, and there exist $D_{\widetilde{z'}}, D_{\widetilde{z'_2}}, \ldots, D_{\widetilde{z'_m}}$ with $D_{\widetilde{z'}}[D_{\widetilde{z'_2}}[S_2] \parallel \cdots \parallel D_{\widetilde{z'_m}}[S_m]] = \texttt{signal}\ \widetilde{k}\ \texttt{in}\ \mathcal{H}(\widetilde{k})$. We proceed then by induction on the structure of P_0. We show only cases for $P_0 = \mathbf{0}$ and $P_0 = x\langle v\rangle.P'_0$, as the other cases are similar:

$P_0 = \mathbf{0}$**:** By Thm. 3.70, $Q_0 = k_2[i : \epsilon, o : \epsilon] \mid \ldots \mid k_m[i : \epsilon, o : \epsilon]$. Thus, by IH $\Sigma_0 = \delta(Q_0) = \{k_o^2 : \epsilon, k_i^2 : \epsilon, \ldots, k_o^m : \epsilon, k_i^m : \epsilon\}$ and hence,

$$\langle [\![\mathbf{0}]\!] \parallel S_2 \parallel \cdots \parallel S_m \diamond \Sigma_0\rangle \longmapsto \langle [\![\mathbf{0}]\!] \parallel T_2 \parallel \cdots \parallel T_m \diamond \Sigma_0\rangle$$

By Lem. 8.12, there exist $D_{\widetilde{z}}$, $D_{\widetilde{z_2}}, \ldots D_{\widetilde{z_m}}$ such that $\parallel D_{\widetilde{z}}[D_{\widetilde{z_2}}[T_2] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]] = \texttt{signal}\ k\ \texttt{in}\ \mathcal{H}(k)$, up-to renaming of bound variables.

Similarly, we can show that $\widetilde{z} = \widetilde{k'} \cdot \widetilde{ack_i^{k_r}} \cdot \widetilde{ack_o^{k_r}} \cdot \widetilde{ack_i^{\overline{k_r}}} \cdot \widetilde{ack_o^{\overline{k_r}}}$ for $k_r \in \widetilde{k}$, where $\widetilde{s} = \widetilde{k'}$ and $\widetilde{s'} = \widetilde{ack_i^{k_r}} \cdot \widetilde{ack_o^{k_r}} \cdot \widetilde{ack_i^{\overline{k_r}}} \cdot \widetilde{ack_o^{\overline{k_r}}}$ and that:

$$D_{\widetilde{s}}[[\![\mathbf{0}]\!] \parallel D_{\widetilde{s'}}[D_{\widetilde{z_2}}[T_2] \parallel \cdots \parallel D_{\widetilde{z_m}}[T_m]]]$$

we then conclude the proof by letting $P' = \mathbf{0}$ and $Q' = Q_0$.

$P_0 = x\langle v\rangle.P'_0$**:** By Thm. 3.70, $Q_0 = x[i : \widetilde{h_1}, o : \widetilde{h_2}] \mid Q'_0$, with Q'_0 a parallel composition of queue processes. Thus, $\Sigma_0 = x_i : \widetilde{h_1}, x_o : \widetilde{h_2}, \Sigma'_0$. By Fig. 3.9, Fig. 3.10, and Fig. 3.11:

$$\begin{aligned}&\langle [\![x\langle v\rangle.P'_0]\!] \parallel S_2 \parallel \cdots \parallel S_m \diamond x_i : \widetilde{h_1}, x_o : \widetilde{h_2}, \Sigma'_0\rangle \\ &\longmapsto \langle T_1 \parallel T_2 \parallel \cdots \parallel T_m \diamond x_i : \widetilde{h'_1}, x_o : \widetilde{h'_2} \cdot v \cdot \widetilde{h_3}, \Sigma'\rangle\end{aligned}$$

provided that $[\![P'_0]\!] \longmapsto T_1$ and $S_r \longmapsto T_r$ for $2 \leq r \leq m$ (rules $\lfloor$L-PAR$\rfloor$ and $\lfloor$L-DONE$\rfloor$). By applying IH2 we can conclude that $T_1 = [\![P']\!]$ for some P' such that $C[x\langle v\rangle.P'_0, Q_0] \ggg\!\!\rightarrow^* C[P', Q']$.

We now prove that $\delta(Q') = x_i : \widetilde{h'_1}, x_o : \widetilde{h'_2} \cdot v \cdot \widetilde{h_3}, \Sigma'$. For this we distinguish two cases: (1) whenever there is a synchronization possible between queues and (2) whenever such synchronization does not occur. We show case (1): assume that $\widetilde{h_2} = v_1 \cdot \widetilde{h'_2}$. Then, by Thm. 3.70, $Q'_0 = \overline{x}[i : \widetilde{h_4}, o : v_2 \cdot \widetilde{h_5}] \mid Q''_0$, for some Q''_0 composed of only queues. Thus:

$$\begin{aligned}&C[x\langle v\rangle.P'_0, x[i : \widetilde{h_1}, o : \widetilde{h_2}] \mid Q'_0] \\ &= C[x\langle v\rangle.P'_0, x[i : \widetilde{h_1}, o : v_1 \cdot \widetilde{h'_2}] \mid \overline{x}[i : \widetilde{h_4}, o : v_2 \cdot \widetilde{h_5}] \mid Q''_0]\end{aligned}$$

We analyze the big-step reduction:

$$C[x\langle v\rangle.P_0', x[i:\widetilde{h_1}, o:\widetilde{h_2}] \mid \overline{x}[i:\widetilde{h_4}, o:v_2\cdot\widetilde{h_5}] \mid Q_0''] \ggg\!\!\!\rightarrow R$$

By Def. 3.81, it can be shown that:

$$R = C[P', x[i:\widetilde{h_1''}\cdot v_2, o:\widetilde{h_2'}\cdot v\cdot\widetilde{h_3}] \mid \overline{x}[i:\widetilde{h_4'}\cdot v_1, o:\widetilde{h_5'}] \mid Q_0''']$$

where $\widetilde{h_1'} = \widetilde{h_1''}\cdot v_2$. We can then show that $\delta(x[i:\widetilde{h_1''}\cdot v_2, o:\widetilde{h_2'}\cdot v\cdot\widetilde{h_3}] \mid \overline{x}[i:\widetilde{h_4'}\cdot v_1, o:\widetilde{h_5'}] \mid Q_0''') = \Sigma$ by applying Lem. 8.11(1) and using Def. 8.2. Furthermore, by using an argument similar to the previous case, we can show the existence of $D_{\widetilde{z}}, D_{\widetilde{z_2}}, \ldots, D_{\widetilde{z_m}}$, finishing the proof.

Inductive Step: The inductive step proceeds by assuming the property holds for that $P_0 = P_1 \mid \ldots \mid P_n$ and we analyze for $P_0 = P_1 \mid \ldots \mid P_n \mid P_{n+1}$. The proof proceeds similarly as the base case, by induction on the structure of P_{n+1}.

□

F
Chapter 10

F.1 Reactivity

Lemma 10.13 (Size Reduction During Instantaneous Execution). *Let C be a reachable configuration. Then:*

$$C \longrightarrow C' \;\;\Rightarrow\;\; \mathit{Size}(C) \;>\; \mathit{Size}(C')$$

Proof (*see Page 256*). We distinguish two cases, depending on whether C is an initial configuration or not.

(1) Let $C = \langle P, M, E\rangle$. Then $C \longrightarrow C' = (\nu s)\langle P', M', E'\rangle$ is deduced by Rule [Init]. Here $C = \langle a[1](\alpha_1).P_1 \mid ... \mid a[n](\alpha_n).P_n \mid \bar{a}[n], \emptyset, \emptyset\rangle$ and

$$C' = (\nu s)\langle P_1\{{}^{s[1]}/\alpha_1\} \mid \;\ldots\; \mid P_n\{{}^{s[n]}/\alpha_n\}, M^{\emptyset}_s, \emptyset\rangle$$

Then we may conclude, since

$$\begin{aligned} \mathit{size}_M(P) &= \sum_{i=1}^{n} \mathit{size}_M(a[i](\alpha_i).P_i) = n + \sum_{i=1}^{n} \mathit{size}_{M^{\emptyset}_s}(P_i\{s[i]/\alpha_i\}) \\ &= n + \mathit{size}_{M^{\emptyset}_s}(P') > \mathit{size}_{M^{\emptyset}_s}(P') \end{aligned}$$

(2) Let $C = (\nu s)\langle P, M, E\rangle$. In this case we have $(\nu s)\langle P, M, E\rangle \longrightarrow (\nu s)\langle P', M', E'\rangle$ if and only $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$, and thus it is enough to prove the following statement:

If $(\nu s)\langle P, M, E\rangle$ is reachable then $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$ implies $\mathit{size}_M(P) > \mathit{size}_{M'}(P')$

To prove this statement we proceed by induction on the inference of the transition $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$, and case analysis on the last rule used in the inference. We examine the interesting cases.

- *Basic cases*
  - Rule [Out]. In this case, $\langle P, M, E\rangle = \langle s[\mathsf{p}]!\langle e\rangle.Q, M'' \cup s[\mathsf{p}] : \epsilon, E\rangle$ and that $\langle P', M', E'\rangle = \langle Q, M\{s[\mathsf{p}] \mapsto (v, \emptyset)\}, E\rangle$ where $e \downarrow v$. Then, since $(s,p) \notin \mathit{Fired}(M)$, we have $\mathit{size}_M(P) = 1 + \mathit{size}_{M'}(Q) = 1 + \mathit{size}_{M'}(P') > \mathit{size}_{M'}(P')$.
  - Rule [In]. In this case
    $$\langle P, M, E\rangle = \langle s[\mathsf{q}]?(\mathsf{p}, x).P, M'' \cup s[\mathsf{p}] : (v, \Pi), E\rangle$$
    for some Π such that $q \notin \Pi$. Moreover
    $$\langle P', M', E'\rangle = \langle P\{^{v}\!/_{x}\}, M\{s[p] \mapsto (v, \Pi \cup \mathsf{q})\}, E\rangle$$
    Hence, $\mathit{size}_M(P) = 1 + \mathit{size}_{M'}(P\{v/x\}) = 1 + \mathit{size}_{M'}(P') > \mathit{size}_{M'}(P')$.
- *Inductive cases*
  - Rule [Rec]. Here we have $\langle P, M, E\rangle = \langle \mathsf{rec}\ X\,.\,Q, M, E\rangle$, and the reduction
    $$\langle \mathsf{rec}\ X\,.\,Q, M, E\rangle \longrightarrow \langle P', M', E'\rangle$$
    is deduced from the reduction
    $$\langle Q\{^{(\mathsf{pause}.\,\mathsf{rec}\ X\,.\,Q)}\!/_{X}\}, M, E\rangle \longrightarrow \langle P', M', E'\rangle$$
    by induction, $\mathit{size}_M(Q\{^{(\mathsf{pause}.\,\mathsf{rec}\ X\,.\,Q)}\!/_{X}\}) > \mathit{size}_{M'}(P')$. Whence also:
    $$\begin{aligned} \mathit{size}_M(P) &= \mathit{size}_M(\mathsf{rec}\ X\,.\,Q) \\ &= \mathit{size}_M(Q\{^{(\mathsf{pause}.\,\mathsf{rec}\ X\,.\,Q)}\!/_{X}\}) \\ &> \mathit{size}_{M'}(P') \end{aligned}$$
  - Rule [Cont]. Easy induction.

□

Lemma 10.15 (Immediate Convergence of 0-*size* Configurations). *Let C be a reachable configuration. Then*
$$(\mathit{Size}(C) = 0) \quad \Leftrightarrow \quad C \ddagger_{\curlyvee}$$

Proof (*see Page 257*). Note that C cannot be an initial configuration, since in this case we would have $\mathit{Size}(C) > 0$. Hence $C = (\nu s)\langle P, M, E\rangle$. Since
$$\mathit{Size}((\nu s)\langle P, M, E\rangle) = \mathit{Size}(\langle P, M, E\rangle) = \mathit{size}_M(P)$$
and $(\nu s)\langle P, M, E\rangle \ddagger_{\curlyvee} \quad \Leftrightarrow \quad \langle P, M, E\rangle \ddagger_{\curlyvee}$, it is enough to prove the statement for $C = \langle P, M, E\rangle$. We prove each side of the biconditional in turn.

($\Rightarrow$) We proceed by simultaneous induction on the structure of P and on the number of pause-unguarded recursive calls in P, considering only the cases for which $\mathit{size}_M(P) = 0$. Note that the reachability assumption rules out the cases $P = X$, $P = \bar{a}[n]$ and $P = a[\mathsf{p}](\alpha).Q$, while the assumption $\mathit{size}_M(P) = 0$ rules out the cases $P = \mathtt{emit}\,ev.\,Q$ and $P \equiv \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$.

Basic Cases

- $P = \mathbf{0}$. Then $P \equiv \mathbf{0}$ and thus $\langle P, M, E \rangle \underset{\curlyvee}{\ddagger}$ by definition.
- $P = \mathsf{pause}.\, Q$. Then $\langle P, M, E \rangle \ddagger$ by Rule $(pause)$ in Fig. 10.7 and thus $\langle P, M, E \rangle \underset{\curlyvee}{\ddagger}$ by definition.
- $P = s[\mathsf{p}]!\langle e \rangle.Q$. Since we assumed $\mathit{size}_M(P) = 0$, we have $(s, \mathsf{p}) \in \mathit{Fired}(M)$, i.e., there exist v, Π such that $s[\mathsf{p}] : (v, \Pi) \in M$. Then $\langle P, M, E \rangle \ddagger$ by Rule (out_s) in Fig. 10.7.
- $P \equiv s[\mathsf{q}]?(\mathsf{p}, x).Q$. Since $\mathit{size}_M(P) = 0$, then either $(s \in \mathit{sn}(M) \wedge (s, \mathsf{p}) \notin \mathit{Fired}(M))$ or $(s, \mathsf{p}, \mathsf{q}) \in \mathit{Comm}(M)$. In both cases we can deduce $\langle P, M, E \rangle \ddagger$, respectively by Rule (in_s) and by Rule (in_s^2) in Fig. 10.7.

Inductive Cases

- $P = P_1 \mid P_2$. Since $\mathit{size}_M(P) = 0$, we have $\mathit{size}_M(P_i) = 0$ for $i = 1, 2$. By induction, this implies $\langle P_i, M, E \rangle \ddagger$ for $i = 1, 2$. Whence, by Rule (par_s) in Fig. 10.7, we deduce $\langle P_1 \mid P_2, M, E \rangle \ddagger$.
- $P = \mathsf{rec}\, X \,.\, Q$. Since $\mathit{size}_M(P) = 0$, we have that

  $$\mathit{size}_M(Q\{{}^{(\mathsf{pause}.\,\mathsf{rec}\, X \,.\, Q)}/_{X}\}) = 0$$

  by Property 10.12. By induction on the number of pause-unguarded recursive calls, we have that $\langle Q\{{}^{(\mathsf{pause}.\,\mathsf{rec}\, X \,.\, Q)}/_{X}\}, M, E \rangle \ddagger$. Therefore, we may conclude that $\langle P, M, E \rangle \ddagger$, using Rule (rec_s) in Fig. 10.7.
- $\mathsf{watch}\, ev\ \mathsf{do}\ Q\{R\}$. Since $\mathit{size}_M(P) = 0$, we have $\mathit{size}_M(Q) = 0$. By induction $\langle Q, M, E \rangle \ddagger$. Then $\langle P, M, E \rangle \ddagger$ by Rule $(watch_s)$ in Fig. 10.7.

($\Leftarrow$) There are two possibilities for $\langle P, M, E \rangle \underset{\curlyvee}{\ddagger}$:

1) If $P \equiv \mathbf{0}$, we proceed by induction on the definition of $\equiv$. In essence, $P \equiv \mathbf{0}$ if and only if P is an n-ary parallel composition whose components are either $\mathbf{0}$ or of the form $\mathsf{watch}\, ev\ \mathsf{do}\, Q\{R\}$, where $Q \equiv \mathbf{0}$. In all cases, by Def. 10.9 we have $\mathit{size}_M(P) = 0$.

2) If $\langle P, M, E \rangle \ddagger$, we proceed by induction on the definition of the suspension predicate in Fig. 10.7. In each case, the reasoning is dual to that for the ($\Rightarrow$) direction above.

□

Lemma 10.16 (Deadlock freedom). *Let C be a reachable configuration. Then*

$$\textit{either } C \underset{\curlyvee}{\ddagger} \textit{ or } \exists\, C' \,.\, C \longrightarrow C'$$

Proof (*see Page 257*). We distinguish two cases, depending on whether C is an initial configuration or not.

(1) Let $C = \langle P, M, E \rangle$ be an initial configuration. Then there is a reduction $C \longrightarrow C' = (\nu s)\langle P', M', E' \rangle$ deduced by Rule [Init].

(2) Let $C = (\nu s)\langle P, M, E\rangle$. Then C reduces if and only if $\langle P, M, E\rangle$ reduces and $C \underset{\curlyvee}{\ddagger}$ if and only if $\langle P, M, E\rangle \underset{\curlyvee}{\ddagger}$. Hence it is enough to prove the property for $\langle P, M, E\rangle$. We proceed by induction on the structure of P. Note that the reachability assumption rules out the cases $P = X$, $P = \bar{a}[n]$ and $P = a[\mathsf{p}](\alpha).Q$.

Basic Cases

- $P = \mathbf{0}$. Then $\langle P, M, E\rangle \underset{\curlyvee}{\ddagger}$ by definition.
- $P = \mathtt{emit}\, ev.\, Q$. By Rule [Emit], for any M, E we have that
$$\langle \mathtt{emit}\, ev.\, Q, M, E\rangle \longrightarrow \langle Q, M, E \cup \{ev\}\rangle$$
- $P = \mathtt{pause}.\, Q$. By Rule $(pause)$, for any M, E we have $\langle P, M, E\rangle\ddagger$.
- $P = \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$. Since the evaluation of e terminates, for any M, E a reduction may be inferred by either Rule [If-T] or Rule [If-F].
- $P \equiv s[\mathsf{p}]!\langle e\rangle.Q$. By the reachability condition, $s[\mathsf{p}] \in dom(M)$. There are then two possibilities:
(i) $M = M' \cup \{s[\mathsf{p}] : \varepsilon\}$, in which case $\langle P, M, E\rangle$ can reduce by Rule [Out];
(ii) $M = M' \cup \{s[\mathsf{p}] : (v, \Pi)\}$, in which case $\langle P, M, E\rangle\ddagger$ by Rule (out_s).
- $P \equiv s[\mathsf{q}]?(\mathsf{p}, x).Q$. In this case, there are three possibilities:
(i) there is a message sent by p that participant q has not read yet, in which case the configuration can reduce by Rule [In];
(ii) there is no message sent by p, in which case the configuration is suspended by Rule (in_s);
(iii) there is a message sent by p that participant q has read already, in which case the configuration is suspended by Rule (in^2_s).

Inductive cases. Straightforward, by applying the inductive hypothesis.

□

Theorem 10.18 (Bounded Reactivity). *Let C be a reachable configuration. Then*
$$\exists\, n \leq \mathit{Size}(C)\,.\; C \Downarrow_n$$

Proof (*see Page 258*). We distinguish two cases, depending on whether C is an initial configuration or not. Since the latter case depends on the former, we start by considering non initial configurations.

(1) C is not initial. In this case, C has the form $C = (\nu s)\langle P, M, E\rangle$, where P is a parallel composition of sequential session-closed processes. Therefore we have:
$$\begin{array}{rcl} \mathit{Size}((\nu s)\langle P, M, E\rangle) & = & \mathit{Size}(\langle P, M, E\rangle) \;=\; \mathit{size}_M(P) \\ (\nu s)\langle P, M, E\rangle \underset{\curlyvee}{\ddagger} & \Leftrightarrow & \langle P, M, E\rangle \underset{\curlyvee}{\ddagger} \\ (\nu s)\langle P, M, E\rangle \longrightarrow (\nu s)\langle P', M', E'\rangle & \Leftrightarrow & \langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle \end{array}$$
Therefore it is enough to prove the statement:
$$\exists\, n \leq \mathit{size}_M(P)\,.\;\; \langle P, M, E\rangle \Downarrow_n$$
We proceed by simultaneous induction on the structure of P, on the size of P and on the number of pause-unguarded recursive calls in P.

- *Basic case: $\mathit{size}_M(P) = 0$ and P has no pause-unguarded recursive calls.*
  By Lem. 10.15, if $\mathit{size}_M(P) = 0$ then either $P \equiv \mathbf{0}$ or $\langle P, M, E\rangle\ddagger$.
  By definition $\langle P, M, E\rangle \overset{\ddagger}{\curlyvee} \;\Rightarrow\; (\langle P, M, E\rangle \Downarrow_0 \langle P, M, E\rangle)$. Then we may conclude, since $n = 0 = \mathit{size}_M(P)$.

- *Inductive cases: $\mathit{size}_M(P) \geq 1$ or P has pause-unguarded calls.*
  - $P = \mathtt{emit}\, ev.\, P'$. In this case, by Rule [Emit] we have the reduction:

    $$\langle \mathtt{emit}\, ev.\, P', M, E\rangle \longrightarrow \langle P', M, E \cup \{ev\}\rangle$$

    By induction (on the size or on the structure), there must exist $n \leq \mathit{size}_M(P')$ such that $\langle P', M, E \cup \{ev\}\rangle \Downarrow_n$. Then

    $$\langle \mathtt{emit}\, ev.\, P', M, E\rangle \Downarrow_{n+1}$$

    where $n + 1 \leq \mathit{size}_M(P') + 1 = \mathit{size}_M(\mathtt{emit}\, ev.\, P')$.

  - $P = s[\mathsf{q}]?(\mathsf{p}, x).Q$. By Lem. 10.16, there are two possibilities:
    - $\langle s[\mathsf{q}]?(\mathsf{p}, x).Q, M, E\rangle\ddagger$. This is inferred either using Rule (in_s), if $s \in sn(M) \wedge (s, p) \notin \mathit{Fired}(M)$, or using Rule (in_s^2), if $(s, p, q) \in \mathit{Comm}(M)$.
      Then $\langle s[\mathsf{q}]?(\mathsf{p}, x).Q, M, E\rangle \Downarrow_0 \langle Q, M, E\rangle$ and we may conclude, since $n = 0 \leq \mathit{size}_M(P)$.
    - $\langle P, M, E\rangle = \langle s[\mathsf{q}]?(\mathsf{p}, x).Q, M''{\cup}s[\mathsf{p}] : (v, \Pi), E\rangle$ for some M'', v and Π such that $q \notin \Pi$. Then Rule [In] can be applied, yielding

      $$\langle P', M', E'\rangle = \langle Q\{{}^{v}\!/\!{}_{x}\}, M\{s[p] \mapsto (v, \Pi \cup \mathsf{q})\}, E\rangle$$

      By induction there exists $n \leq \mathit{size}_M(P')$ such that $\langle P', M', E'\rangle \Downarrow_n$. Hence $\langle P, M, E\rangle \Downarrow_{n+1}$, where

      $$n + 1 \leq \mathit{size}_M(P') + 1 = \mathit{size}_M(s[\mathsf{q}]?(\mathsf{p}, x).Q)$$

  - $P = s[\mathsf{p}]!\langle e\rangle.Q$. This case is similar to the previous one, and slightly simpler.
  - $P = \mathtt{rec}\, X\, .\, Q$. By Lem. 10.16, there are two possibilities:
    - $\langle \mathtt{rec}\, X\, .\, Q, M, E\rangle\ddagger$. Then $\langle \mathtt{rec}\, X\, .\, Q, M, E\rangle \Downarrow_0 \langle \mathtt{rec}\, X\, .\, Q, M, E\rangle$ and we may conclude, since $n = 0 = \mathit{size}_M(P)$.
    - There exist P', M', E' such that $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$. Then the reduction is inferred by Rule [Rec], namely:

      $$\frac{\langle Q\{{}^{(\mathsf{pause.\,rec}\, X\, .\, Q)}\!/\!{}_{X}\}, M, E\rangle \longrightarrow \langle P', M', E'\rangle}{\langle \mathtt{rec}\, X\, .\, Q, M, E\rangle \longrightarrow \langle P', M', E'\rangle}\;[\mathrm{Rec}]$$

      Since the call $\mathtt{rec}\, X\, .\, Q$ is pause-guarded in P', the number of pause-unguarded calls in P' is strictly less than in P. Then by induction there exists $n \leq \mathit{size}_M(P')$ such that $\langle P', M', E'\rangle \Downarrow_n$.

Whence $\langle P, M, E\rangle \Downarrow_{n+1}$. By Lem. 10.13, we know that $\mathit{size}_M(P') < \mathit{size}_M(Q\{^{(\mathsf{pause}.\,\mathsf{rec}\, X\,.\,Q)}/_X\})$. By Property 10.12,

$$\mathit{size}_M(Q\{^{(\mathsf{pause}.\,\mathsf{rec}\, X\,.\,Q)}/_X\}) = \mathit{size}_M(P)$$

We may thus conclude that $n < \mathit{size}_M(P)$, that is to say, $n + 1 \leq \mathit{size}_M(P)$.

– Conditional, parallel and watch: these cases are straightforward by induction on the structure of the process.

(2) $C = \langle P, M, E\rangle$ is initial. Thus we have

$$C = \langle a[1](\alpha_1).P_1 \mid \ldots \mid a[k](\alpha_k).P_k \mid \bar{a}[k], \emptyset, \emptyset\rangle$$

Then by Rule [Init] we have a reduction $C \longrightarrow C' = (\nu s)\langle P', M_s^{\emptyset}, \emptyset\rangle$, where

$$P' = (\nu s)P_1\{^{s[1]}/_{\alpha_1}\} \mid \ldots \mid P_k\{^{s[k]}/_{\alpha_k}\}$$

Moreover, by Def. 10.9:

$$\begin{aligned} \mathit{Size}(C) &= \sum_{i=1}^{k} \mathit{size}_{\emptyset}(a[i](\alpha_i).P_i) = k + \sum_{i=1}^{k} \mathit{size}_{M_s^{\emptyset}}(P_i\{s[i]/\alpha_i\}) \\ &= k + \mathit{size}_{M_s^{\emptyset}}(P') \;>\; \mathit{size}_{M_s^{\emptyset}}(P') = \mathit{Size}(C') \end{aligned}$$

By Point 1. there exists $m \leq \mathit{Size}(C')$ such that $C' \Downarrow_m$. Letting $n = m + 1$, we conclude that $C \Downarrow_n$ and $n \leq \mathit{Size}(C)$.

□

F.2 Type System

Lemma 10.24 (Correctness of Saturation with Respect to $\mathsf{OG}(T)$). *Let G be a global type such that $|\mathsf{Part}(G)| \geq 2$, p be a participant, and $\mathcal{P}$ be a set of participants such that (1) $\mathcal{P} \subseteq \mathsf{Part}(G)$, and (2) $\mathsf{Part}(G) \setminus \mathcal{P} \neq \emptyset$. Then, $\mathsf{OG}(\mathsf{S}_{\mathsf{Part(G)}}(G, \mathcal{P}) \upharpoonright \mathsf{p})$ holds for every $\mathsf{p} \in \mathsf{Part}(G) \setminus \mathcal{P}$.*

Proof (see Page 265). By induction on the structure of G. The base cases are $G = \mathsf{end}$ and $G = \mathbf{t}$, which are vacuously true since $|\mathsf{Part}(\mathsf{end})| = |\mathsf{Part}(\mathbf{t})| = 0$. For the inductive step we assume that the property holds for a global type G' and prove the statement for every type that contains G' as a sub-expression. There are five inductive cases.

Case $G = \mathsf{pause}.G'$:

(1)	$\mathsf{S}_{\mathsf{Part(G)}}(G, \mathcal{P}) \upharpoonright \mathsf{p} = \mathsf{S}_{\mathsf{Part(G)}}(\mathsf{pause}.G', \mathcal{P}) \upharpoonright \mathsf{p}$ $= \mathsf{p}_{\{1,n\}} \uparrow \langle S, \emptyset\rangle.\mathsf{pause}.\mathsf{S}_{\mathsf{Part(G')}}(G', \emptyset) \upharpoonright \mathsf{p}$	(Assumption, Fig. 10.11)
(2)	$\mathsf{Part}(G) \setminus \mathcal{P} = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$	(Assumption, Fig. 10.11)
(3)	$\forall \mathsf{p}_i \in \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}.(\mathsf{p}_{\{1,n\}} \uparrow \langle S, \emptyset\rangle.\mathsf{pause}.\mathsf{S}_{\mathsf{Part(G')}}(G', \emptyset) \upharpoonright \mathsf{p}_i$ $= !S_i.\mathsf{pause}.(\mathsf{S}_{\mathsf{Part(G')}}(G', \emptyset) \upharpoonright \mathsf{p}))$	((1), (2), Fig. 10.12)
(4)	$\mathsf{OG}(!S_i.\mathsf{pause}.(\mathsf{S}_{\mathsf{Part(G')}}(G', \emptyset) \upharpoonright \mathsf{p}))$ is true	(Def. 10.23, (3))

Case $G = \texttt{tick}.G'$**:** Analogous to the case above.

Case $G = \mathsf{r}\!\uparrow\!\langle S, \Pi\rangle.G'$**:** There are two cases depending on whether $\mathsf{r} \in \mathcal{P}$ or not:

Case $\mathsf{r} \notin \mathcal{P}$**:**

(1) By Def. 10.11,

$$\mathsf{S}_{\mathsf{Part(G)}}(\mathsf{r}\!\uparrow\!\langle S, \Pi\rangle.G', \mathcal{P}) = \mathsf{r}\!\uparrow\!\langle S, \Pi\rangle.\mathsf{S}_{\mathsf{Part(G)}}(G', \mathcal{P} \cup \{\mathsf{r}\})$$

(2) By IH, $\forall \mathsf{p} \in \mathcal{P}.(\mathsf{OG}(\mathsf{S}_{\mathsf{Part(G)}}(G', \mathcal{P}) \upharpoonright \mathsf{p}))$.

(3) By Def. 10.12, (2), and (1) $\mathsf{OG}(\mathsf{r} \uparrow \langle S, \Pi\rangle.\mathsf{S}_{\mathsf{Part(G)}}(G', \mathcal{P} \cup \{\mathsf{r}\}) \upharpoonright \mathsf{r})$ is true.

Case $\mathsf{r} \in \mathcal{P}$**:** By applying Fig. 10.11 to G, which adds an output for every $\mathsf{r} \in \mathcal{P}$.

Case $G = \texttt{watch}\ ev\ \texttt{do}\ G'\ \texttt{else}\ G''$**:** Follows directly from the IH. By Def. 10.23, we have that $\mathsf{OG}(\langle T_1, T_2\rangle^{ev}) = \mathsf{OG}(T_1)$.

□

Lemma 10.33. *If* $(\mathsf{S}_{\mathcal{R}}(G, \mathcal{P}) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}}(G, \mathcal{P}) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$, $\mathcal{P} \subseteq \mathcal{P}'$, $\mathsf{Part}(G) \subseteq \mathcal{R} \subseteq \mathcal{R}'$, $\mathcal{P} \subseteq \mathcal{R}$ *and* $\mathcal{P}' \subseteq \mathcal{R}'$ *then* $(\mathsf{S}_{\mathcal{R}'}(G, \mathcal{P}') \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}'}(G, \mathcal{P}') \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$.

Proof (*see Page 268*). By induction on the structure of G. There are two base cases and five inductive cases:

Base Cases: The cases are $G = \mathsf{end}$ and $G = \mathbf{t}$. We only show $G = \mathsf{end}$, as the other is similar. Let $\mathcal{P}, \mathcal{P}'$, $\mathcal{R}$, and $\mathcal{R}'$ be sets of participants satisfying satisfying the following assumptions:

Case $G = \mathsf{end}$**:**

(1)	$\mathsf{Part}(G) \subseteq \mathcal{R} \subseteq \mathcal{R}'$	(Assumption)
(2)	$\mathcal{P} \subseteq \mathcal{R}$	(Assumption)
(3)	$\mathcal{P}' \subseteq \mathcal{R}'$	(Assumption)
(4)	$(\mathsf{S}_{\mathcal{R}}(\mathsf{end}, \mathcal{P}) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}}(\mathsf{end}, \mathcal{P}) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$	(Assumption)
(5)	$\mathcal{P} \subseteq \mathcal{P}'$	(Assumption)

Then, we distinguish cases depending on whether p, q belong to $\mathcal{P}'$ or not.

(a) $\mathsf{p} \in \mathcal{P}'$ and $\mathsf{q} \in \mathcal{P}'$.
(b) $\mathsf{p} \notin \mathcal{P}'$ and $\mathsf{q} \notin \mathcal{P}'$.
(c) $\mathsf{p} \in \mathcal{P}'$ and $\mathsf{q} \notin \mathcal{P}'$.
(d) $\mathsf{p} \notin \mathcal{P}'$ and $\mathsf{q} \in \mathcal{P}'$.

All the cases proceed similarly, hence we only show (a):

Sub-Case (a):

(i)	$\forall \mathsf{r} \in \mathcal{P}.(\mathsf{S}_{\mathcal{R}'}(\mathsf{end}, \mathcal{P}')$ $= \mathsf{S}_{\mathcal{R}'}(\mathsf{end}, \mathcal{P}') = \mathsf{r}_{\{1,n\}}\!\uparrow\!\langle S_d, \emptyset\rangle.\mathsf{end}$	(Fig. 10.11)
(ii)	$(\mathsf{r}_{\{1,n\}}\!\uparrow\!\langle S_d, \emptyset\rangle.\mathsf{end} \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} =$ $(!S_d.\mathsf{end}) \upharpoonright \mathsf{q} = !S_d.\mathsf{end}$	(Fig. 10.12, Fig. 10.16, (i))
(iii)	$(\mathsf{r}_{\{1,n\}}\!\uparrow\!\langle S_d, \emptyset\rangle.\mathsf{end} \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p} = (!S_d) \upharpoonright \mathsf{p} = !S_d.\mathsf{end}$	(Fig. 10.12, Fig. 10.16, (i))
(iv)	$!S_d.\mathsf{end} \bowtie !S_d.\mathsf{end}$	(Def. 10.30, (ii), (iii))

Inductive Cases: There are five cases. As in the base cases, let $\mathcal{P}, \mathcal{P}'$, $\mathcal{R}$, and $\mathcal{R}'$ be sets of participants satisfying satisfying the following assumptions:

(1)	$\mathsf{Part}(G) \subseteq \mathcal{R} \subseteq \mathcal{R}'$	(Assumption)
(2)	$\mathcal{P} \subseteq \mathcal{R}$	(Assumption)
(3)	$\mathcal{P}' \subseteq \mathcal{R}'$	(Assumption)
(4)	$(\mathsf{S}_{\mathcal{R}}(G, \mathcal{P}) \restriction \mathsf{p}) \restriction \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}}(G, \mathcal{P}) \restriction \mathsf{q}) \restriction \mathsf{p}$	(Assumption)
(5)	$\mathcal{P} \subseteq \mathcal{P}'$	(Assumption)

Then, we distinguish cases depending on the shape of G. The IH states that the property holds for all the sub-terms G' of type G.

Case $G = \mathsf{r}{\uparrow}\langle S, \Pi\rangle.G'$**:**

IH: $(\mathsf{S}_{\mathcal{R}}(\mathsf{r}{\uparrow}\langle S, \Pi\rangle.G', \mathcal{P}) \restriction \mathsf{p}) \restriction \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}}(\mathsf{r}{\uparrow}\langle S, \Pi\rangle.G', \mathcal{P}) \restriction \mathsf{q}) \restriction \mathsf{p}$ implies that $(\mathsf{S}_{\mathcal{R}'}(\mathsf{r}{\uparrow}\langle S, \Pi\rangle.G', \mathcal{P}') \restriction \mathsf{p}) \restriction \mathsf{q} \bowtie (\mathsf{S}_{\mathcal{R}'}(\mathsf{r}{\uparrow}\langle S, \Pi\rangle.G', \mathcal{P}') \restriction \mathsf{q}) \restriction \mathsf{p}$ for every $\mathcal{R}', \mathcal{P}'$ such that $\mathcal{R} \subseteq \mathcal{R}' \wedge \mathcal{P} \subseteq \mathcal{P}'$.

We then further distinguish two cases depending on whether $\mathsf{r} \in \mathcal{P}$ or $\mathsf{r} \notin \mathcal{P}$:

Case $\mathsf{r} \in \mathcal{P}$**:** By Fig. 10.11, $\mathsf{S}_{\mathcal{R}}(\mathsf{r} \uparrow \langle S, \Pi\rangle.G', \mathcal{P}') = \mathsf{r} \uparrow \langle S, \Pi\rangle.\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{r}\})$. We then further distinguish eight sub-cases depending on the conditions satisfied by p and q:

Sub-case $\mathsf{r} = \mathsf{p} \wedge \mathsf{q} \in \Pi$**:** Then $G = \mathsf{p}{\uparrow}\langle S, \Pi\rangle.G'$:

(i)	$\mathsf{p}{\uparrow}\langle S, \Pi\rangle.\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\}) \restriction \mathsf{p}$ $= !S.(\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\})) \restriction \mathsf{p}$	(Fig. 10.12)
(ii)	$\mathsf{p}{\uparrow}\langle S, \Pi\rangle.\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\}) \restriction \mathsf{p} \restriction \mathsf{q}$ $= !S.((\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\})) \restriction \mathsf{p}) \restriction \mathsf{q}$	(Fig. 10.16)
(iii)	$\mathsf{p}{\uparrow}\langle S, \Pi\rangle.\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\}) \restriction \mathsf{q} \restriction \mathsf{p}$ $= ?(\mathsf{q}, S).((\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\})) \restriction \mathsf{q}) \restriction \mathsf{p}$	(Fig. 10.12 and Fig. 10.16)

By (1) and (5), $\mathcal{R} \subseteq \mathcal{R}'$ and $\mathcal{P} \subseteq \mathcal{P}'$. Thus, we distinguish two more sub-cases: $\mathsf{p} \notin \mathcal{P}'$ and $\mathsf{p} \in \mathcal{P}$:

Sub-case $\mathsf{p} \notin \mathcal{P}'$**:** Notice that:
(i) $\mathsf{p} \uparrow \langle S, \Pi\rangle.\mathsf{S}_{\mathcal{R}'}(G', \mathcal{P}' \cup \{\mathsf{p}\}) \restriction \mathsf{p} \restriction \mathsf{q} = !S.((\mathsf{S}_{\mathcal{R}'}(G', \mathcal{P}' \cup \{\mathsf{p}\})) \restriction \mathsf{p}) \restriction \mathsf{q}$ by Fig. 10.12 and Fig. 10.16. Also, by assumption, Fig. 10.12 and Fig. 10.16 we have (ii) $\mathsf{p} \uparrow \langle S, \Pi\rangle.\mathsf{S}_{\mathcal{R}'}(G', \mathcal{P} \cup \{\mathsf{p}\}) \restriction \mathsf{q} \restriction \mathsf{p} = ?(\mathsf{q}, S).((\mathsf{S}_{\mathcal{R}}(G', \mathcal{P}' \cup \{\mathsf{p}\})) \restriction \mathsf{q}) \restriction \mathsf{p}$. Since by assumption we have that:

$$((\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\})) \restriction \mathsf{p}) \restriction \mathsf{q} \bowtie ((\mathsf{S}_{\mathcal{R}}(G', \mathcal{P} \cup \{\mathsf{p}\})) \restriction \mathsf{q}) \restriction \mathsf{p}$$

and that $\mathcal{R}' \supseteq \mathcal{R} \wedge \mathcal{P}' \supseteq \mathcal{P}$, we can apply the inductive hypothesis to obtain (iii)

$$((\mathsf{S}_{\mathcal{R}'}(G', \mathcal{P}' \cup \{\mathsf{p}\})) \restriction \mathsf{p}) \restriction \mathsf{q} \bowtie ((\mathsf{S}_{\mathcal{R}'}(G', \mathcal{P}' \cup \{\mathsf{p}\})) \restriction \mathsf{q}) \restriction \mathsf{p}$$

and then we can conclude by Def. 10.30.

Sub-case $\mathsf{p} \in \mathcal{P}'$**:** Notice that (i) $\mathsf{S}_{\mathcal{R}}(\mathsf{p} \uparrow \langle S, \Pi\rangle.G', \mathcal{P}') = \mathsf{r}_{\{1,n\}} \uparrow \langle S_d, \emptyset\rangle.\mathtt{tick}.\mathsf{p} \uparrow \langle S, \Pi\rangle.\mathsf{S}_{\mathsf{Part}(G')}(G', \{\mathsf{r}\})$ by Fig. 10.11. Then, we have that (ii) $\mathsf{r}_{\{1,\mathsf{n}\}}\uparrow\langle S_d,\emptyset\rangle.\mathtt{tick}.\mathsf{p}\uparrow\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G')}}(G',\{\mathsf{p}\})\restriction\mathsf{p} = \mathtt{tick}.!S.(\mathsf{S}_{\mathsf{Part(G')}}(G,\{\mathsf{p}\})\restriction\mathsf{p})$ by Fig. 10.12 and (iii):

$$\begin{aligned}&\mathtt{tick}.!S.(\mathsf{S}_{\mathsf{Part(G')}}(G,\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q}\\&=\mathtt{tick}.!S.((\mathsf{S}_{\mathsf{Part(G')}}(G,\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q}\end{aligned}$$

by Fig. 10.16 and Assumption. Also, by assumption, Fig. 10.12 and Fig. 10.16 we have (iv):

$$\begin{aligned}&(\mathsf{r}_{\{1,\mathsf{n}\}}\uparrow\langle S_d,\emptyset\rangle.\mathtt{tick}.\mathsf{p}\uparrow\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G')}}(G',\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q}\\&=\mathtt{tick}.?(\mathsf{p},S).((\mathsf{S}_{\mathsf{Part(G')}}(G',\{\mathsf{p}\})\restriction\mathsf{q})\restriction\mathsf{p}\end{aligned}$$

Finally, since $\{\mathsf{p}\}\subseteq\mathcal{P}'$ and the length of G' has decreased we can apply the IH and Def. 10.30 to show that duality holds.

Sub-case $\mathsf{r}=\mathsf{p}\wedge\mathsf{q}\notin\Pi$: This proof proceeds similarly as above, while considering that q is not a receptor from the broadcast done by p. Then, the case will conclude by Def. 10.30 and IH, since an output can be dual to any type.

Sub-case $\mathsf{r}=\mathsf{q}\wedge\mathsf{p}\in\Pi$: Symmetrical to Case $\mathsf{r}=\mathsf{p}\wedge\mathsf{q}\in\Pi$.

Sub-case $\mathsf{r}=\mathsf{q}\wedge\mathsf{p}\notin\Pi$: Symmetrical to Case $\mathsf{r}=\mathsf{p}\wedge\mathsf{q}\notin\Pi$.

Sub-case $\mathsf{p}\in\Pi\wedge\mathsf{q}\in\Pi$: In this case $\mathsf{r}\neq\mathsf{p}\wedge\mathsf{r}\neq\mathsf{q}$, by Definition. Notice also, that in this case both participants are receptors, and hence, by Fig. 10.16, the input prefix obtained by Fig. 10.12 will disappear, hence the case will conclude by the IH.

Sub-case $\mathsf{p}\notin\Pi\wedge\mathsf{q}\in\Pi$: Analogous to the case above.

Sub-case $\mathsf{p}\in\Pi\wedge\mathsf{q}\notin\Pi$: Analogous to the case above.

Sub-case $\mathsf{p}\notin\Pi\wedge\mathsf{q}\notin\Pi$: Analogous to the case above.

Case $\mathsf{r}\in\mathcal{P}$: We have that:

$$\mathsf{S}_{\mathcal{R}}(\mathsf{r}\uparrow\langle S,\Pi\rangle.G',\mathcal{P}')=\mathsf{r}_{\{1,\mathsf{n}\}}\uparrow\langle S_d,\emptyset\rangle.\mathtt{tick}.\mathsf{r}\uparrow\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G')}}(G',\{\mathsf{r}\})$$

by Fig. 10.11. The proof proceeds as shown above, the only difference being the fact that we are adding a tick to the global type by projection. Hence, our IH will consider $\mathsf{Part}(G')$ and $\{\mathsf{r}\}$ as parameters.

Case $G=\mathtt{pause}.G'$: This case is straightforward by applying the IH, since the size of the parameters of the saturation function will only affect the projections up to the number of outputs, which are dual with any type.

Case $G=\mu\mathbf{t}.G'$: There are two cases, but they are straightforward by applying the IH, notice that projections do not affect the recursion, so they only work internally in the body of the recursive type.

Case $G=\mathtt{watch}\ ev\ \mathtt{do}\ G'\ \mathtt{else}\ G''$: This case again is straightforward by applying the inductive hypothesis. Similarly to the recursive type, we have to consider that the IH is applied to both the main body and alternative body. In the alternative body, it is necessary to consider that the size of the parameters set will not affect duality.

□

Proposition 10.34. *Let G be a global type and* $\mathsf{p} \neq \mathsf{q}$. *Then* $(G\lfloor_{\mathsf{p}}) \upharpoonright \mathsf{q} \bowtie (G\lfloor_{\mathsf{q}}) \upharpoonright \mathsf{p}$.

Proof (*see Page 269*). By induction on G. Let $\mathsf{p}, \mathsf{q} \in \mathsf{Part}(G)$ and let $\mathsf{Part}(G) = \mathcal{R}$.

Case $G = \mathsf{end}$**:**

(1)	$(\mathsf{end}\lfloor_{\mathsf{p}}) \upharpoonright \mathsf{q} = (\mathsf{S}_{\mathcal{R}}(\mathsf{end}, \emptyset) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q}$	(Fig. 10.12, Fig. 10.11)
(2)	$(\mathsf{end}\lfloor_{\mathsf{q}}) \upharpoonright \mathsf{p} = (\mathsf{S}_{\mathcal{R}}(\mathsf{end}, \emptyset) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$	(Fig. 10.12 and Fig. 10.11)
(3)	$\mathsf{Part}(\mathsf{end}) = \mathcal{R} = \emptyset$	(Definition of $\mathsf{Part}(G)$)
(4)	$\mathcal{R} \setminus \mathcal{P} = \emptyset$	(algebra of sets)
(5)	$\mathsf{S}_{\mathcal{R}}(\mathsf{end}, \emptyset) = \mathsf{S}_{\emptyset}(\mathsf{end}, \emptyset) = \mathsf{end}$	(Fig. 10.11)
(6)	$(\mathsf{S}_{\emptyset}(\mathsf{end}, \emptyset) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} = (\mathsf{end} \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} = \mathsf{end} \upharpoonright \mathsf{q} = \mathsf{end}$	((5), Fig. 10.12, Fig. 10.16)
(7)	$(\mathsf{S}_{\emptyset}(\mathsf{end}, \emptyset) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p} = (\mathsf{end} \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p} = \mathsf{end} \upharpoonright \mathsf{p} = \mathsf{end}$	((5), Fig. 10.12, Fig. 10.16)
(8)	$\mathsf{end} \bowtie \mathsf{end}$	(Def. 10.30)

Case $G = \mathbf{t}$**:** As above.

Case $G = \mathsf{r}\!\uparrow\!\langle S, \Pi\rangle.G'$**:** We have that $(G'\lfloor_{\mathsf{p}}) \upharpoonright \mathsf{q} \bowtie (G'\lfloor_{\mathsf{q}}) \upharpoonright \mathsf{p}$ by IH. Moreover,

$$(\mathsf{S}_{\mathsf{Part}(G')}(G', \emptyset) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (\mathsf{S}_{\mathsf{Part}(G')}(G', \emptyset) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$$

by Fig. 10.12 and Fig. 10.11. Then, We distinguish cases depending on r and the memberships of p, q in Π. There are eight cases:

Case $\mathsf{r} = \mathsf{p} \wedge \mathsf{q} \in \Pi$**:** Then $G = \mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.G'$:

(i) By Fig. 10.11:

$$\mathsf{S}_{\mathsf{Part}(G)}(\mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.G', \emptyset) = \mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\})$$

(ii) By Fig. 10.12, $\mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{p} = !S.(\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{p})$.

(iii) By Fig. 10.16, $!S.(\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} = !S.((\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q})$.

(iv) By Fig. 10.12:

$$\mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{q} = ?(\mathsf{p}, S).(\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{q})$$

(v) By Fig. 10.16, $?(\mathsf{p}, S).(\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p} = ?S.((\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p})$.

We conclude by Lem. 10.33 and IH that

$$(\mathsf{S}_{\mathsf{Part}(G')}(G', \emptyset) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q} \bowtie (\mathsf{S}_{\mathsf{Part}(G')}(G', \emptyset) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p}$$

implies $!S.((\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{p}) \upharpoonright \mathsf{q}) \bowtie ?S.((\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \upharpoonright \mathsf{q}) \upharpoonright \mathsf{p})$ which is the same as saying $\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\}) \bowtie \mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\})$.

Case $\mathsf{r} = \mathsf{p} \wedge \mathsf{q} \notin \Pi$**:** Then $G = \mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.G'$:

(i) By Fig. 10.11:

$$\mathsf{S}_{\mathsf{Part}(G)}(\mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.G', \emptyset) = \mathsf{p}\!\uparrow\!\langle S, \Pi\rangle.\mathsf{S}_{\mathsf{Part}(G)}(G', \{\mathsf{p}\})$$

(ii) By Fig. 10.12, $\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{p} = !S.(\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{p})$.

(iii) By Fig. 10.16, $!S.(\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q} = !S.((\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q})$.

(iv) By Fig. 10.12, $\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q} = \mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q}$.

(v) By Fig. 10.16, $(\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q})\restriction\mathsf{p} = (\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q})\restriction\mathsf{p}$.

We then conclude by Lem. 10.33 since

$$(\mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset)\restriction\mathsf{p})\restriction\mathsf{q} \bowtie (\mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset)\restriction\mathsf{q})\restriction\mathsf{p}$$

implies

$$!S.((\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q}) \bowtie ((\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q})\restriction\mathsf{p})$$

which means that $\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\}) \bowtie \mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})$.

Case $\mathsf{r}=\mathsf{q}\wedge\mathsf{p}\in\Pi$**:** Symmetrical to Case $\mathsf{r}=\mathsf{p}\wedge\mathsf{q}\in\Pi$.

Case $\mathsf{r}=\mathsf{q}\wedge\mathsf{p}\notin\Pi$**:** Symmetrical to Case $\mathsf{r}=\mathsf{p}\wedge\mathsf{q}\notin\Pi$.

Case $\mathsf{p}\in\Pi\wedge\mathsf{q}\in\Pi$**:** In this case $\mathsf{r}\neq\mathsf{p}\wedge\mathsf{r}\neq\mathsf{q}$, hence:

(i) By Fig. 10.11:

$$\mathsf{S}_{\mathsf{Part(G)}}(\mathsf{r}\!\uparrow\!\langle S,\Pi\rangle.G',\emptyset) = \mathsf{r}\!\uparrow\!\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{r}\})$$

(ii) By Fig. 10.12, $\mathsf{S}_{\mathsf{Part(G)}}(\mathsf{r}\!\uparrow\!\langle S,\Pi\rangle.G',\emptyset) = \mathsf{r}\!\uparrow\!\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{r}\})$.

(iii) By Fig. 10.16, $?(\mathsf{r},S).(\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q} = (\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{p})\restriction\mathsf{q}$.

(iv) By Fig. 10.12, $\mathsf{p}\!\uparrow\!\langle S,\Pi\rangle.\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q} = ?(\mathsf{r},S).(\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q})$.

(v) By Fig. 10.16, $?(\mathsf{p},S).(\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q})\restriction\mathsf{p} = (\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q})\restriction\mathsf{p}$.

We then conclude by Lem. 10.33, since

$$(\mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset)\restriction\mathsf{p})\restriction\mathsf{q} \bowtie (\mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset)\restriction\mathsf{q})\restriction\mathsf{p}$$

implies

$$((\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{r}\})\restriction\mathsf{p})\restriction\mathsf{q}) \bowtie ((\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{p}\})\restriction\mathsf{q})\restriction\mathsf{p})$$

which means that $\mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{r}\}) \bowtie \mathsf{S}_{\mathsf{Part(G)}}(G',\{\mathsf{r}\})$.

Case $\mathsf{p}\notin\Pi\wedge\mathsf{q}\in\Pi$**:** Analogous to the case above – concludes by Lem. 10.33.

Case $\mathsf{p}\in\Pi\wedge\mathsf{q}\notin\Pi$**:** Analogous to the case above – concludes by Lem. 10.33.

Case $\mathsf{p}\notin\Pi\wedge\mathsf{q}\notin\Pi$**:** Analogous to the case above – concludes by Lem. 10.33.

Case $G=\mathsf{pause}.G'$**:**

(1) $(\mathsf{pause}.G'\lfloor_{\mathsf{p}})\restriction\mathsf{q} = (\mathsf{S}_{\mathcal{R}}(\mathsf{pause}.G',\emptyset)\restriction\mathsf{p})\restriction\mathsf{q}$ (Fig. 10.12, Fig. 10.11)

(2) $(\mathsf{pause}.G'\lfloor_{\mathsf{q}})\restriction\mathsf{p} = (\mathsf{S}_{\mathcal{R}}(\mathsf{pause}.G',\emptyset)\restriction\mathsf{q})\restriction\mathsf{p}$ (Fig. 10.12, Fig. 10.11)

(3) $\mathsf{S}_{\mathsf{Part(G)}}(\mathsf{pause}.G',\emptyset) = \mathsf{r}_{\{1,n\}}\!\uparrow\!\langle S_d,\emptyset\rangle.\mathsf{pause}.\mathsf{S}_{\mathsf{Part(G')}}(G',\emptyset)$ (Fig. 10.11)

We then distinguish cases depending on the presence of p,q in $\{\mathsf{r}_1,\ldots\mathsf{r}_n\}$. There are four cases:

Case $\mathsf{p}, \mathsf{q} \in \{\mathsf{r}_1, \dots \mathsf{r}_n\}$**:** This case follows from the Lem. 10.33.

Case $\mathsf{p} \in \{\mathsf{r}_1, \dots \mathsf{r}_n\} \wedge \mathsf{q} \notin \{\mathsf{r}_1, \dots \mathsf{r}_n\}$**:** This case follows from the Lem. 10.33.

Case $\mathsf{p} \notin \{\mathsf{r}_1, \dots \mathsf{r}_n\} \wedge \mathsf{q} \in \{\mathsf{r}_1, \dots \mathsf{r}_n\}$**:** This case follows from the Lem. 10.33.

Case $\mathsf{p}, \mathsf{q} \notin \{\mathsf{r}_1, \dots \mathsf{r}_n\}$**:**

Case $G = \mu \mathbf{t}.G'$**:** There are two cases, but they proceed straightforward by the IH. Note that $\mathsf{Part}(\mu \mathbf{t}.G') = \mathsf{Part}(G')$.

Case $G = \mathtt{watch}\ ev\ \mathtt{do}\ G'\ \mathtt{else}\ G''$**:** This case follows by Lem. 10.33. This is because the duality of a watch operator depends on the duality of its components.

□

F.3 Properties of the Type System

Lemma 10.35. *If* $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ *then* $\Gamma \vdash M \triangleright \Theta$.

Proof (*see Page 272*). By induction on the height of the typing derivation

$$\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$$

Base Cases:

Case $\lfloor$RVAR$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$RVAR$\rfloor$. Then, by inversion, we have $\Gamma \vdash M \triangleright \Theta$, concluding the proof.

Case $\lfloor$INACT$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$INACT$\rfloor$. Then, by inversion, we have $\Gamma \vdash M \triangleright \Theta$, concluding the proof.

Case $\lfloor$MINIT$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$MINIT$\rfloor$. Notice that the memories are empty. Hence, the judgment $\Gamma \vdash M \triangleright \Theta$ holds by Rule $\lfloor$EMPTYMEM$\rfloor$ in Fig. 10.15.

Case $\lfloor$MACC$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$MACC$\rfloor$. Notice that the memories are empty. Hence, the judgment $\Gamma \vdash M \triangleright \Theta$ holds by Rule $\lfloor$EMPTYMEM$\rfloor$ in Fig. 10.15.

Inductive Cases:

Case $\lfloor$WEAK$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$WEAK$\rfloor$. By inversion, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta, c : \mathsf{end} \diamond \Theta\rangle$. Then, by IH, $\Gamma \vdash M \triangleright \Theta$.

Case $\lfloor$CONTR$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$CONTR$\rfloor$. By inversion, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \setminus c : \mathsf{end} \diamond \Theta\rangle$. Then, by IH, $\Gamma \vdash M \triangleright \Theta$.

Case $\lfloor$CONC$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$CONC$\rfloor$. By inversion, $P = P_1 \mid P_2$, $\Gamma \vdash \langle P_1, M, E\rangle \triangleright \langle \Delta_1, c : \mathsf{end} \diamond \Theta\rangle$, and $\Gamma \vdash \langle P_1, M, E\rangle \triangleright \langle \Delta_1, c : \mathsf{end} \diamond \Theta\rangle$, with $\Delta = \Delta_1, \Delta_2$. Then, by IH, $\Gamma \vdash M \triangleright \Theta$, concluding the proof.

Case $\lfloor$EMIT$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$EMIT$\rfloor$. As above, this case concludes by applying inversion and the IH.

Case $\lfloor$PAUSE$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$PAUSE$\rfloor$. By inversion, we have that $\Gamma \vdash M \triangleright \Theta$, which is what we wanted to prove.

Case $\lfloor$REC$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$REC$\rfloor$. By inversion, (1) $\Gamma, X : \mathtt{pause}.T \vdash \langle P, M^{\emptyset}, \emptyset\rangle \triangleright \langle c : T \diamond \Theta^{\emptyset}\rangle$ and (2) $\Gamma, X : \mathtt{pause}.T \vdash \langle P, M, E\rangle \triangleright \langle c : T \diamond \Theta\rangle$. Then, by applying the IH on (2), we conclude that $\Gamma, X : \mathtt{pause}.T \vdash M \triangleright \Theta$. Since $X$is not used in typing M it implies that $\Gamma \vdash M \triangleright \Theta$, which allows us to conclude this case.

Case $\lfloor$SENDFIRST$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. From the rule it can be deduced that $M = M' \cup s[p] : \epsilon$, $\Theta = \Theta', s[\mathsf{p}] : \mathtt{void}$. Then, by inversion on the Rule, (1) $\Gamma \vdash \langle P, M' \cup s[\mathsf{p}] : (d_S, \emptyset), E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta', s[\mathsf{p}] : (S, \emptyset)\rangle$. From IH applied to (1), we have that (2) $\Gamma \vdash M' \cup s[\mathsf{p}] : (d_S, \emptyset) \triangleright \Theta', s[\mathsf{p}] : (S, \emptyset)$. Notice that the judgment in (2) can only be deduced from Rule $\lfloor$MERGEMEM$\rfloor$ in Fig. 10.15, thus, by inversion we have that (3) $\Gamma \vdash M' \triangleright \Theta'$. Then, applying Rule $\lfloor$VOIDMSG$\rfloor$, we have that (4) $\Gamma \vdash s[\mathsf{p}] : \epsilon \triangleright s[\mathsf{p}] : \mathtt{void}$. Thus, applying Rule $\lfloor$MERGEMEM$\rfloor$, we can conclude that $\Gamma \vdash M' \cup s[\mathsf{p}] : \epsilon \triangleright \Theta', s[\mathsf{p}] : \mathtt{void}$, which concludes the proof.

Case $\lfloor$SENDMORE$\rfloor$, $\lfloor$RCVFIRST$\rfloor$, $\lfloor$RCVMORE$\rfloor$, *and* $\lfloor$RCVNEXT$\rfloor$**:** The proof for each of these cases proceeds similarly as above, by using inversion, applying the IH and using the rules in Fig. 10.15, to deduce that the memory is typable.

Case $\lfloor$WATCH$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with

$$P = \mathtt{watch}\, ev\ \mathtt{do}\ Q_1\{Q_2\}$$

and $\langle \Delta \diamond \Theta\rangle = \langle T_{Q_1} \star_{ev} T_{Q_2} \diamond \Theta\rangle$. By inversion on Rule $\lfloor$WATCH$\rfloor$, we have that $\Gamma \vdash \langle Q_1, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_{Q_1} \diamond \Theta\rangle$. Then, we can conclude by applying the IH to $\Gamma \vdash \langle Q_1, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_{Q_1} \diamond \Theta\rangle$.

Case $\lfloor$IF$\rfloor$**:** By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ with Rule $\lfloor$IF$\rfloor$. The proof proceeds by inversion and applying the IH.

□

Lemma 10.41 (Subject Congruence). *If* $\Gamma \vdash C \triangleright \langle \Delta \diamond \Theta\rangle$ *and* $C \equiv C'$ *then* $\Gamma \vdash C' \triangleright \langle \Delta \diamond \Theta\rangle$.

Proof (see Page 274). By induction on the structural congruence relation in Fig. 10.5. There are two cases:

Case $P \equiv Q \Rightarrow \langle P, M, E\rangle \equiv \langle Q, M, E\rangle$**:** Assume $C = \langle P, M, E\rangle$ and $C' = \langle Q, M, E\rangle$. We proceed by case analysis on the hypothesis $P \equiv Q$. There are 4 rules by which it may be deduced:

Case $R \mid \mathbf{0} \equiv R$. Suppose $P = R \mid \mathbf{0}$ and $Q = R$. By assumption, we have that $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. This judgment is necessarily deduced using Rule $\lfloor$CONC$\rfloor$. By inversion, (i) $\Gamma \vdash \langle R, M, E\rangle \triangleright \langle \Delta_1 \diamond \Theta\rangle$, and (ii) $\Gamma \vdash \langle \mathbf{0}, M, E\rangle \triangleright \langle \Delta_2 \diamond \Theta\rangle$. Furthermore, the judgment in (ii) has to be necessarily deduced using Rule $\lfloor$INACT$\rfloor$. Thus, applying inversion, (iv) $\Delta_2 = \Delta^{end}$. Thus, we conclude by applying Rule $\lfloor$WEAK$\rfloor$ to add Δ_2 in (i), obtaining $\Gamma \vdash \langle R, M, E\rangle \triangleright \langle \Delta_1, \Delta_2 \diamond \Theta\rangle$.

Let us now consider the inverse case, i.e., $P = R$ and $Q = R \mid \mathbf{0}$. By assumption, we have $\Gamma \vdash \langle R, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$, and we have to prove that $\Gamma \vdash \langle R \mid \mathbf{0}, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. By Lem. 10.35, (i) $\Gamma \vdash M \triangleright \Theta$. Then, observe that $\Delta_0 = \emptyset$ implies that $\Delta_0 = \Delta^{end}$ by Def. 10.21. We may then apply Rule $\lfloor \text{Inact} \rfloor$, using $\Delta_0 = \emptyset$ and (i) to derive $\Gamma \vdash \langle \mathbf{0}, M, E\rangle \triangleright \langle \emptyset \diamond \Theta\rangle$, and subsequently apply Rule $\lfloor \text{Conc} \rfloor$ to obtain $\Gamma \vdash \langle R \mid 0, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$.

Case $\texttt{watch}\, ev\, \texttt{do}\, \mathbf{0}\{R\} \equiv \mathbf{0}$. Suppose $P = \texttt{watch}\, ev\, \texttt{do}\, \mathbf{0}\{R\}$ and $Q = \mathbf{0}$. By assumption, $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. This judgement is deduced with Rule $\lfloor \text{Watch} \rfloor$, hence $\Delta = \{s[\mathsf{p}] : T_\mathbf{0} \star_{ev} T_R\}$. By inversion on Rule $\lfloor \text{Watch} \rfloor$, we have (i) $\Gamma \vdash M \triangleright \Theta$ and (ii) $\Gamma \vdash \langle \mathbf{0}, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_\mathbf{0} \diamond \Theta\rangle$. Since (ii) is deduced by Rule $\lfloor \text{Inact} \rfloor$, it must be $T_\mathbf{0} = \mathsf{end}$. Hence also $T_\mathbf{0} \star_{ev} T_R = \mathsf{end}$ (cf. Def. 10.26), and we may conclude that $\Gamma \vdash \langle Q, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$.

Conversely, suppose $P = \mathbf{0}$ and $Q = \texttt{watch}\, ev\, \texttt{do}\, \mathbf{0}\{R\}$. By assumption, $\Gamma \vdash \langle \mathbf{0}, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. This judgment is deduced by Rule $\lfloor \text{Inact} \rfloor$. By inversion we have (i) $\Delta = \Delta^{end}$. Hence, we distinguish two cases: (1) $\Delta = \emptyset$ and (2) $\Delta \neq \emptyset$. In Case (1), we apply Rule $\lfloor \text{Weak} \rfloor$ to obtain $\Gamma \vdash \langle \mathbf{0}, M, E\rangle \triangleright \langle c : \mathsf{end} \diamond \Theta\rangle$. Then, we apply Rule $\lfloor \text{Watch} \rfloor$ to deduce $\Gamma \vdash \langle \texttt{watch}\, ev\, \texttt{do}\, \mathbf{0}\{R\}, M, E\rangle \triangleright \langle s[\mathsf{p}] : \mathsf{end} \diamond \Theta\rangle$. Finally, we apply Rule $\lfloor \text{Contr} \rfloor$ to conclude $\Gamma \vdash \langle \texttt{watch}\, ev\, \texttt{do}\, \mathbf{0}\{R\}, M, E\rangle \triangleright \langle \emptyset \diamond \Theta\rangle$.

For Case (2), consider an arbitrary $s[\mathsf{p}] \in dom(\Delta)$. By (i), $s[\mathsf{p}] : \mathsf{end} \in \Delta$. Then, using the contraction rule $\lfloor \text{Contr} \rfloor$, we may deduce (ii) $\Gamma \vdash \langle \mathbf{0}, M, E\rangle \triangleright \langle s[\mathsf{p}] : \mathsf{end} \diamond \Theta\rangle$. We may now apply Rule $\lfloor \text{Watch} \rfloor$ to the judgement (ii) to obtain $\Gamma \vdash \langle \texttt{watch}\, ev\, \texttt{do}\, \mathbf{0}\{R\}, M, E\rangle \triangleright \langle s[\mathsf{p}] : \mathsf{end} \diamond \Theta\rangle$. Then, by iteratively applying Rule $\lfloor \text{Weak} \rfloor$ to add $\Delta \setminus s[\mathsf{p}] : \mathsf{end}$ to the session environment, we finally obtain $\Gamma \vdash \langle \texttt{watch}\, ev\, \texttt{do}\, \mathbf{0}\{R\}, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$.

Case $R_1 \mid R_2 \equiv R_2 \mid R_1$. The proof of this case is straightforward by inversion, since the hypotheses $\Gamma \vdash \langle R_1, M, E\rangle \triangleright \langle \Delta_1 \diamond \Theta\rangle$ and $\Gamma \vdash \langle R_2, M, E\rangle \triangleright \langle \Delta_2 \diamond \Theta\rangle$ are not ordered in Rule $\lfloor \text{Conc} \rfloor$.

Case $(R_1 \mid R_2) \mid R_3 \equiv R_1 \mid (R_2 \mid R_3)$. Again, the proof is straightforward by inversion, considering that
$(dom(\Delta_1) \cap dom(\Delta_2)) \cap dom(\Delta_3) = \emptyset$ if and only if $dom(\Delta_1) \cap (dom(\Delta_2) \cap dom(\Delta_3)) = \emptyset$.

Case $C_1 \equiv C_2 \Rightarrow (\nu s)C_1 \equiv (\nu s)C_2$: Suppose that $C = (\nu s)C_1$ and $C' = (\nu s)C_2$. By assumption, $\Gamma \vdash C \triangleright \langle \emptyset \diamond \emptyset\rangle$. This judgment is derived with Rule $\lfloor \text{CRes} \rfloor$. Then, by inversion, $\Gamma \vdash C_1 \triangleright \langle \Delta \diamond \Theta\rangle$ and $Co\,\langle \Delta \diamond \Theta\rangle$. By induction $\Gamma \vdash C_2 \triangleright \langle \Delta_1 \diamond \Theta_1\rangle$, and we can apply Rule $\lfloor \text{CRes} \rfloor$ to this statement and to $Co\,\langle \Delta_1 \diamond \Theta_1\rangle$ to obtain $\Gamma \vdash (\nu s)C_2 \triangleright \langle \emptyset \diamond \emptyset\rangle$, namely $\Gamma \vdash C' \triangleright \langle \emptyset \diamond \emptyset\rangle$, which is what we wanted to prove.

□

Lemma 10.42 (Substitution Lemma). *If* $\Gamma, x : S \vdash C \triangleright \langle \Delta \diamond \Theta\rangle$ *and* $\Gamma \vdash v : S$ *then* $\Gamma \vdash C\{^v/_x\} \triangleright \langle \Delta \diamond \Theta\rangle$.

Proof (see Page 274). By induction on the height of the type derivation. The base cases are rules $\lfloor \text{Inact} \rfloor$ and $\lfloor \text{Minit} \rfloor$. The thesis is easily derived observing that x does not occur free neither in **0** nor in the initiator.

For the inductive cases, the interesting ones are those of rules $\lfloor\text{SendFirst}\rfloor$ and $\lfloor\text{SendMore}\rfloor$. All other cases follow by the fact that the process under consideration either do not contain free occurrences of x or the conclusion follows by a direct application of the inductive hypothesis.

Let us consider the case of rule $\lfloor\text{SendFirst}\rfloor$ (the treatment for rule $\lfloor\text{SendMore}\rfloor$ will be the same). We know that:

$$\Gamma, x : S \vdash \langle s[\mathsf{p}]!\langle e\rangle.P, M \cup s[\mathsf{p}] : \epsilon, E\rangle \triangleright \langle s[\mathsf{p}] :!S.T \diamond \Theta, s[\mathsf{p}] : \mathtt{void}\rangle$$

By inversion we have: (1) $\Gamma, x : S \vdash e : S$ and (2) $\Gamma, x : S \vdash \langle P, M \cup s[\mathsf{p}] : (d_S, \emptyset), E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta, s[\mathsf{p}] : (S, \emptyset)\rangle$.

By an easy induction on the height of the type derivation of expressions, Fig. 10.14, and from (1) we deduce $\Gamma, v : S \vdash e\{^v/_x\} : S$. From (2), by inductive hypothesis we conclude $\Gamma, v : S \vdash \langle P\{^v/_x\}, M \cup s[\mathsf{p}] : (d_S, \emptyset), E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta, s[\mathsf{p}] : (S, \emptyset)\rangle$.

Finally using rule $\lfloor\text{SendFirst}\rfloor$ we conclude this case obtaining:

$$\Gamma, v : S \vdash \langle (s[\mathsf{p}]!\langle e\rangle.P)\{^v/_x\}, M \cup s[\mathsf{p}] : \epsilon, E\rangle \triangleright \langle s[\mathsf{p}] :!S.T \diamond \Theta, s[\mathsf{p}] : \mathtt{void}\rangle$$

□

Lemma 10.43 (Reduction Lemma). *Let* $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ *and* $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$ *via some reduction rule different from* [Cont] *and* [Struct]. *Then*

$$\langle \Delta \diamond \Theta\rangle \;\Rightarrow\; \langle \Delta' \diamond \Theta'\rangle$$

and $\Gamma \vdash \langle P', M', E'\rangle \triangleright \langle \Delta' \diamond \Theta'\rangle$. *Moreover, if* $\langle \Delta, \Delta_0 \diamond \Theta, \Theta_0\rangle$ *is coherent, then* $\langle \Delta', \Delta_0 \diamond \Theta', \Theta_0\rangle$ *is coherent.*

Proof (*see Page 274*). By induction on length of the reduction

$$\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$$

with a case analysis on the last applied rule. Notice that Case [Init] is not considered here, as we are only interested in reductions that affect the processes inside a reachable non initial configuration.

Case [Out] : Let $P = s[\mathsf{p}]!\langle e\rangle.P'$, and $M = M'' \cup \{s[\mathsf{p}] : \varepsilon\}$ for some M''. P is typed with rule $\lfloor\text{SendFirst}\rfloor$ and by hypothesis we have: $e \downarrow v$, $M' = M'' \cup \{s[\mathsf{p}] : (v, \emptyset)\}$, $\Delta = s[\mathsf{p}] :!S.T$ and $\Theta = \Theta'', s[\mathsf{p}] : \mathtt{void}$.

By inversion on rule $\lfloor\text{SendFirst}\rfloor$ we know that $\Gamma \vdash e : S$ and $\Gamma \vdash \langle P', M'' \cup \{s[\mathsf{p}] : (d_S, \emptyset)\}, E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta'', s[\mathsf{p}] : (S, \emptyset)\rangle$.

Now since $\Gamma \vdash e : S$ and $e \downarrow v$ we know that $\Gamma \vdash v : S$. Hence we can conclude that

$$\Gamma \vdash \langle P', M'' \cup \{s[\mathsf{p}] : (v, \emptyset)\}, E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta'', s[\mathsf{p}] : (S, \emptyset)\rangle$$

and by using item 1 of Def. (Def. 10.36) we have that

$$\langle s[\mathsf{p}] :!S.T \diamond \Theta'', s[\mathsf{p}] : \mathtt{void}\rangle \;\Rightarrow\; \langle s[\mathsf{p}] : T \diamond \Theta'', s[\mathsf{p}] : (S, \emptyset)\rangle$$

Finally by assumption we have, $Co\,\langle s[\mathsf{p}] :!S.T, \Delta_0 \diamond \Theta'', s[\mathsf{p}] : \mathtt{void}, \Theta_0\rangle$. We know that $Co\,\langle s[\mathsf{p}] : T, \Delta_0 \diamond \Theta'', \{s[\mathsf{p}] : (S, \emptyset), \Theta_0\}\rangle$. We only have to consider the impact of $s[\mathsf{p}]$ as the other members of the environment are unchanged. Since $s[\mathsf{p}] \in \mathit{vdom}(\Theta'', s[\mathsf{p}] : (S, \emptyset), \Theta_0)$ Condition 1 of Def. 10.32 holds. For Condition 2, duality is preserved, as the generalized type of $s[\mathsf{p}]$ in Δ will lift the necessary output from Θ to re-construct the correct type.

Case [In]**:** Let $P = s[\mathsf{q}]?(\mathsf{p}, x).P''$ for some P'' and $M = M'' \cup \{s[\mathsf{p}] : (v, \Pi)\}$ for some M'', v, Π such that $\mathsf{q} \notin \Pi$.

P is typed with rule $\lfloor$RcvFirst$\rfloor$. By hypothesis we have: $P' = P''\{^v/_x\}$, $M' = M'' \cup \{s[\mathsf{p}] : (v, \Pi \cup \mathsf{q})\}$, $\Delta = s[\mathsf{q}] :?(\mathsf{p}, S).T$ and $\Theta = \Theta'', s[\mathsf{p}] : (S, \Pi)$.

By inversion on $\lfloor$RcvFirst$\rfloor$ we know that

$$\Gamma, x : S \vdash \langle P'', M'' \cup s[\mathsf{p}] : (v, \Pi \cup \{\mathsf{q}\}), E\rangle \triangleright \langle s[\mathsf{q}] : T \diamond \Theta'', \{s[\mathsf{p}] : (S, \Pi \cup \{\mathsf{q}\})\}\rangle$$

Now by Lem. 10.42 we can conclude that

$$\Gamma \vdash \langle P''\{^v/_x\}, M'' \cup s[\mathsf{p}] : (v, \Pi \cup \{\mathsf{q}\}), E\rangle \triangleright \langle s[\mathsf{q}] : T \diamond \Theta'', \{s[\mathsf{p}] : (S, \Pi \cup \{\mathsf{q}\})\}\rangle$$

and by item 2 in Def. 10.36

$$\langle s[\mathsf{q}] :?(\mathsf{p}, S).T \diamond \Theta'', s[\mathsf{p}] : (S, \Pi)\rangle \;\Rightarrow\; \langle s[\mathsf{q}] : T \diamond \Theta'', s[\mathsf{p}] : (S, \Pi \cup \{\mathsf{q}\})\rangle$$

Finally, we prove $Co\,\langle s[\mathsf{q}] : T, \Delta_0 \diamond \Theta'', s[\mathsf{p}] : (S, \Pi), \Theta_0\rangle$. Let us check the first condition for coherence; notice that $Co\,\langle \Delta, \Delta_0 \diamond \Theta, \Theta_0\rangle$ implies that either $s[\mathsf{q}] \in \mathit{vdom}(\Theta, \Theta_0)$ or $\mathsf{OG}(?(\mathsf{p}, S).T)$. In both cases coherence holds by assumption. Now, to check duality, let $\Delta_0 = \Delta_0', s[\mathsf{p}] : T'$ and we know that $s[\mathsf{p}] : (S, \Pi) \in \Theta$. Hence, by assumption $\langle \Delta, \Delta_0 \diamond \Theta, \Theta_0\rangle(s[\mathsf{q}]) =?(\mathsf{p}, S).T$ and $\langle \Delta, \Delta_0 \diamond \Theta, \Theta_0\rangle(s[\mathsf{p}]) =!S.T'$. Furthermore, $?(\mathsf{p}, S).T \bowtie !S.T'$ implies $T \bowtie T'$ by Def. 10.30 and therefore, $T \bowtie !s.T'$, which in turn, ensures that duality is preserved in $\langle \Delta', \Delta_0 \diamond \Theta', \Theta_0\rangle$.

Case [Emit]**:** Immediate, since this rule does not change the memory nor the configuration environment.

Case [Watch]**:** Let $P = \mathtt{watch}\, ev\ \mathtt{do}\ P'\{Q\}$. By assumption we know

$$\langle P, M, E\rangle \longrightarrow \langle \mathtt{watch}\, ev\ \mathtt{do}\ P'\{Q\}, M', E'\rangle$$

and $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle T_P' \star_{ev} T_Q \diamond \Theta\rangle$. By inversion on rule $\lfloor$Watch$\rfloor$ we have $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle T_{P'} \diamond \Theta\rangle$, $\Gamma \vdash \langle Q, M^{\emptyset}, E\rangle \triangleright \langle T_Q \diamond \Theta^{\emptyset}\rangle$ and $\mathsf{OG}(T_Q)$. By inductive hypothesis we have $\Gamma \vdash \langle P', M', E'\rangle \triangleright \langle T_P \diamond \Theta'\rangle$ and $\langle T_{P'} \diamond \Theta\rangle \;\Rightarrow\; \langle T_{P'}' \diamond \Theta'\rangle$. Hence, by item 3 in Def. 10.36 we conclude
$\langle T_{P'} \star_{ev} T_Q \diamond \Theta\rangle \;\Rightarrow\; \langle T_{P'}' \star_{ev} T_Q \diamond \Theta'\rangle$.

Finally, assume $Co\,\langle T_{P'} \star_{ev} T_Q, \Delta_0 \diamond \Theta, \Theta_0\rangle$. This implies $Co\,\langle T_{P'}, \Delta_0 \diamond \Theta, \Theta_0\rangle$ and hence, by inductive hypothesis we have $Co\,\langle T_{P'}', \Delta_0 \diamond \Theta', \Theta_0\rangle$, which in turn implies $Co\,\langle T_{P'}' \star_{ev} T_Q, \Delta_0 \diamond \Theta', \Theta_0\rangle$.

Case [Rec]**:** The proof proceeds as above by using the inductive hypothesis.

□

Lemma 10.44 (Tick Reduction of Configuration Environments Preserves Duality). *If* $\langle\Delta\diamond\Theta\rangle \curvearrowright_E \langle\Delta'\diamond\Theta'\rangle$ *then* $\Delta' = [\Delta]_E$, $dom(\Delta') \subseteq dom(\Delta)$ *and* $\Theta' = \Theta^{\emptyset}$. *Moreover, if* $\langle\Delta\diamond\Theta\rangle$ *satisfies duality then also* $\langle\Delta'\diamond\Theta'\rangle$ *satisfies duality and for any* $s[\mathsf{p}], s[\mathsf{q}] \in dom(\Delta')$, *if* $s[\mathsf{p}] : T_{\mathsf{p}} \in \Delta$ *and* $s[\mathsf{q}] : T_{\mathsf{q}} \in \Delta$, *then* $\langle\Delta\diamond\Theta\rangle(s[\mathsf{p}]) \restriction \mathsf{q} \bowtie \langle\Delta\diamond\Theta\rangle(s[\mathsf{q}]) \restriction \mathsf{p}$ *if and only if* $[T_{\mathsf{p}}]_E \restriction \mathsf{q} \bowtie [T_{\mathsf{q}}]_E \restriction \mathsf{p}$.

Proof (*see Page 275*). By induction on the definition of $\curvearrowright$. There is only one basic case, corresponding to Rule (Pause).

Case (*Pause*)**:** In this case $\langle\Delta\diamond\Theta\rangle \curvearrowright_E \langle\Delta'\diamond\Theta'\rangle$ is deduced by Rule 1. of Def. 10.37 and we have $\Delta = \mathsf{pause}(\Delta')$, $E = \emptyset$, $\Delta' = [\Delta]_E$ and $\Theta' = \Theta^{\emptyset}$. Moreover $dom(\Delta') = dom(\Delta)$.

Assume now $s[\mathsf{p}] : T_{\mathsf{p}} \in \Delta$ and $s[\mathsf{q}] : T_{\mathsf{q}} \in \Delta$. Then, letting $T'_{\mathsf{p}} = [T_{\mathsf{p}}]_E$ and $T'_{\mathsf{q}} = [T_{\mathsf{q}}]_E$, we have:

$$T_{\mathsf{p}} = \mathsf{pause}.T'_{\mathsf{p}} \qquad T_{\mathsf{q}} = \mathsf{pause}.T'_{\mathsf{q}} \qquad s[\mathsf{p}] : T'_{\mathsf{p}} \in \Delta' \qquad s[\mathsf{q}] : T'_{\mathsf{q}} \in \Delta'$$

We want to prove the duality of $\langle\Delta'\diamond\Theta^{\emptyset}\rangle$ assuming the duality of $\langle\Delta\diamond\Theta\rangle$. Note that $\langle\Delta'\diamond\Theta^{\emptyset}\rangle(s[\mathsf{p}]) = T'_{\mathsf{p}}$ and $\langle\Delta'\diamond\Theta^{\emptyset}\rangle(s[\mathsf{q}]) = T'_{\mathsf{q}}$. On the other hand, there are two possibilities for $\langle\Delta\diamond\Theta\rangle(s[\mathsf{p}])$: either $\langle\Delta\diamond\Theta\rangle(s[\mathsf{p}]) = T_{\mathsf{p}}$, in case $s[\mathsf{p}] \notin vdom(\Theta)$, or $\langle\Delta\diamond\Theta\rangle(s[\mathsf{p}]) = !S_{\mathsf{p}}.T_{\mathsf{p}}$, in case $s[\mathsf{p}] : (S_{\mathsf{p}}, \Pi_{\mathsf{p}}) \in \Theta$. Similarly, either $\langle\Delta\diamond\Theta\rangle(s[\mathsf{q}) = T_{\mathsf{q}}$ or $\langle\Delta\diamond\Theta\rangle(s[\mathsf{q}]) = !S_{\mathsf{q}}.T_{\mathsf{q}}$.

Let now $\tau_{\mathsf{p}} = T_p \restriction \mathsf{q}$, $\tau_{\mathsf{q}} = T_q \restriction \mathsf{p}$, and $\tau'_{\mathsf{p}} = T'_p \restriction \mathsf{q}$, $\tau'_{\mathsf{q}} = T'_q \restriction \mathsf{p}$. Since projection preserves pause, we have $\tau_{\mathsf{p}} = \mathsf{pause}.\tau'_{\mathsf{p}}$ and $\tau_{\mathsf{q}} = \mathsf{pause}.\tau'_{\mathsf{q}}$. Then:

$$\langle\Delta'\diamond\Theta^{\emptyset}\rangle(s[\mathsf{p}]) \restriction \mathsf{q} = T'_p \restriction \mathsf{q} = \tau'_{\mathsf{p}} \qquad \langle\Delta'\diamond\Theta^{\emptyset}\rangle(s[\mathsf{q}]) \restriction \mathsf{p} = T'_q \restriction \mathsf{p} = \tau'_{\mathsf{q}}$$

Moreover, we have:

$$\begin{array}{lll} \text{either} & \langle\Delta\diamond\Theta\rangle(s[\mathsf{p}) \restriction \mathsf{q} = \tau_{\mathsf{p}} & \text{or} \quad \langle\Delta\diamond\Theta\rangle(s[\mathsf{p}]) \restriction \mathsf{q} = !S_{\mathsf{p}}.\tau_{\mathsf{p}} \\ \text{either} & \langle\Delta\diamond\Theta\rangle(s[\mathsf{q}) \restriction \mathsf{p} = \tau_{\mathsf{q}} & \text{or} \quad \langle\Delta\diamond\Theta\rangle(s[\mathsf{q}]) \restriction \mathsf{p} = !S_{\mathsf{q}}.\tau_{\mathsf{q}} \end{array}$$

We want to show $\tau'_{\mathsf{p}} \bowtie \tau'_{\mathsf{q}}$. By assumption duality holds for $\langle\Delta\diamond\Theta\rangle$, so we are in one of the four cases:

$$\begin{array}{rcl@{\qquad}c@{\qquad}rcl} \tau_{\mathsf{p}} & \bowtie & \tau_{\mathsf{q}} & \text{i.e.} & \mathsf{pause}.\tau'_{\mathsf{p}} & \bowtie & \mathsf{pause}.\tau'_{\mathsf{q}} \\ !S_{\mathsf{p}}.\tau_{\mathsf{p}} & \bowtie & \tau_{\mathsf{q}} & \text{i.e.} & !S_{\mathsf{p}}.\mathsf{pause}.\tau'_{\mathsf{p}} & \bowtie & \mathsf{pause}.\tau'_{\mathsf{q}} \\ \tau_{\mathsf{p}} & \bowtie & !S_{\mathsf{q}}.\tau_{\mathsf{q}} & \text{i.e.} & \mathsf{pause}.\tau'_{\mathsf{p}} & \bowtie & !S_{\mathsf{q}}.\mathsf{pause}.\tau'_{\mathsf{q}} \\ !S_{\mathsf{p}}.\tau_{\mathsf{p}} & \bowtie & !S_{\mathsf{q}}.\tau_{\mathsf{q}} & \text{i.e.} & !S_{\mathsf{p}}.\mathsf{pause}.\tau'_{\mathsf{p}} & \bowtie & !S_{\mathsf{q}}.\mathsf{pause}.\tau'_{\mathsf{q}} \end{array}$$

Now, it is easy to see that any of the four statements on the right-hand side implies $\tau'_{\mathsf{p}} \bowtie \tau'_{\mathsf{q}}$. Hence we have shown duality of $\langle[\Delta]_E\diamond\Theta^{\emptyset}\rangle$ and more specifically that $\langle\Delta\diamond\Theta\rangle(s[\mathsf{p}]) \restriction \mathsf{q} \bowtie \langle\Delta\diamond\Theta\rangle(s[\mathsf{q}]) \restriction \mathsf{p}$ if and only if $[T_{\mathsf{p}}]_E \restriction \mathsf{q} \bowtie [T_{\mathsf{q}}]_E \restriction \mathsf{p}$.

Case (*Tick*)**:** Analogous to the previous case.

Case (*In*)**:** This case is vacuously true, since $\Delta = s[\mathsf{q}] :?(\mathsf{p}, S).T$ and hence it does not have any other type to be compared to.

Case (*Par*)**:** In this case $\langle \Delta \diamond \Theta \rangle \curvearrowright_E \langle \Delta' \diamond \Theta' \rangle$ is deduced by Rule 4. of Def. 10.37. Therefore $\Delta = (\Delta_1, \Delta_2)$ and $\langle \Delta_i \diamond \Theta \rangle \curvearrowright_{E_i} \langle \Delta'_i \diamond \Theta' \rangle$, for $i = 1, 2$. By induction $\Delta'_i = [\Delta_i]_E$, $dom(\Delta'_i) \subseteq dom(\Delta_i)$ and $\Theta' = \Theta^{\emptyset}$. Now, suppose $\langle \Delta_1, \Delta_2 \diamond \Theta \rangle$ satisfies duality. We want to show that also $\langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset} \rangle = \langle [\Delta_1]_{E_1}, [\Delta_2]_E \diamond \Theta^{\emptyset} \rangle$ satisfies duality. Note that since the Δ_i have disjoint domains, the duality of $\langle \Delta_1, \Delta_2 \diamond \Theta \rangle$ entails the duality of each $\langle \Delta_i \diamond \Theta \rangle$. Then by induction also each $\langle [\Delta_i]_E \diamond \Theta^{\emptyset} \rangle$ satisfies duality. Hence, to check duality of $\langle [\Delta_1]_E, [\Delta_2]_E \diamond \Theta^{\emptyset} \rangle$ we only have to look at pairs $s[\mathsf{p}_1], s[\mathsf{p}_2]$ such that $s[\mathsf{p}_1] \in dom([\Delta_1]_E)$ and $s[\mathsf{p}_2] \in dom([\Delta_2]_E)$, because the other cases (i.e., when pairs belong to only one single Δ_i) conclude by IH. Let $s[\mathsf{p}_1], s[\mathsf{p}_2]$ be such a pair. Recalling that $dom([\Delta_i]_E) \subseteq dom(\Delta_i)$, this means that for each $i = 1, 2$, there exists T_i such that $s[\mathsf{p}_i] : T_i \in \Delta_i$. By definition this implies $s[\mathsf{p}_i] : [T_i]_E \in [\Delta_i]_E$. Let $T'_i = [T_i]_{E_i}$. Thus we have:

$$\langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset} \rangle (s[\mathsf{p}_1]) = T'_1 \qquad \langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset} \rangle (s[\mathsf{p}_2]) = T'_2$$

As regards $\langle \Delta_1, \Delta_2 \diamond \Theta \rangle (s[\mathsf{p}_i])$, we have:

$$\begin{array}{ll} \text{either} \;\; \langle \Delta_1, \Delta_2 \diamond \Theta \rangle (s[\mathsf{p}_i]) = T_i & \text{if} \;\; s[\mathsf{p}_i] \notin vdom(\Theta) \\ \text{or} \;\; \langle \Delta_1, \Delta_2 \diamond \Theta \rangle (s[\mathsf{p}_i]) = !S_i.T_i & \text{if} \;\; s[\mathsf{p}_i] : (S_i, \Pi_i) \in \Theta \end{array}$$

Let now $\tau_1 = T_1 \upharpoonright \mathsf{p}_2$ and $\tau_2 = T_2 \upharpoonright \mathsf{p}_1$ and $\tau'_1 = T'_1 \upharpoonright \mathsf{p}_2$ and $\tau'_2 = T'_2 \upharpoonright \mathsf{p}_1$. We need to prove that $\tau_1 \bowtie \tau_2$ implies $\tau'_1 \bowtie \tau'_2$. There are then 4 cases to analyze:

$$\begin{array}{rcl} \tau_1 & \bowtie & \tau_2 \\ !S_1.\tau_1 & \bowtie & \tau_2 \\ \tau_1 & \bowtie & !S_2.\tau_2 \\ !S_1.\tau_1 & \bowtie & !S_2.\tau_2 \end{array}$$

this proceeds as follows:

Case $\tau_1 \bowtie \tau_2$**:** We need to apply induction on the structure of τ_1, base case is $\tau_1 = \mathsf{end}$:

Case $\tau_1 = \mathsf{end} \vee \tau_1 = \mathbf{t} \vee \tau_1 =!S.\tau''_1 \vee \tau''_1 =?S.\tau''_1$**:** Let $\tau_2 = \mu\mathbf{t}.\tau''_2$ and $\tau_2 = \mathsf{pause}.\tau''_2$. Then, the statement is vacuously true, since duality does not hold, and for the other case, duality proceeds simply by assumption, since the types do not change.

Case $\tau_1 = \mathsf{pause}.\tau''_1$**:** Now, we apply a case analysis on τ_2:

Case $\tau_2 = \mathsf{pause}.\tau''_2$**:** In this case we have that $\mathsf{pause}.\tau''_1 \bowtie \mathsf{pause}.\tau''_2$. Then we also can say that $[\mathsf{pause}.\tau''_1]_E = \tau''_1$ and $[\mathsf{pause}.\tau''_2]_E = \tau''_2$. Moreover, since we know that $T'_i = [T_i]_E$, then we know that $\tau''_1 = \tau'_1$ and $\tau''_2 = \tau'_2$, and hence it is clear that $\tau'_1 \bowtie \tau'_2$ by Def. 10.30. All the other cases are vacuously true.

Case $\tau_1 = \mu\mathbf{t}.\tau''_1$ **:** We can only proceed for the case when $\tau_1 = \mu\mathbf{t}'.\tau''_2$, other cases are vacuously true. In this case, we proceed by applying the IH, since we are looking at the bodies of the recursive types.

Case $\tau_1 = \langle\varphi_1, \psi_1\rangle^e$**:** For this case we only need to look at $\tau_2 = \langle\varphi_2, \psi_2\rangle^e$, all the other cases are vacuously true. For this case we have to consider two sub-cases depending on whether $e \in E$ or $e \notin E$:

Sub-case $e \in E$**:** In this case, by assumption, we have that $\langle\varphi_1, \psi_1\rangle^e \bowtie \langle\varphi_2, \psi_2\rangle^e$. Also, by Def. 10.38 we have that $[\langle\varphi_1, \psi_1\rangle^e]_E = \psi_1$ and $[\langle\varphi_2, \psi_2\rangle^e]_E = \psi_2$, hence $\tau_1' = \psi_1$ and $\tau_2 = \psi_2$. Notice that by Def. 10.30, $\langle\varphi_1, \psi_1\rangle^e \bowtie \langle\varphi_2, \psi_2\rangle^e$ implies $\varphi_1 \bowtie \varphi_2$ and $\psi_1 \bowtie \psi_2$ and therefore, we conclude by assumption.

Sub-case $e \notin E$**:** This case concludes by applying the IH hypothesis, since the reconditioning is applied to the first part of the watch type.

Case $!S.\tau_1 \bowtie \tau_2$**:** As above, we apply induction on the structure of τ_1 and obtain the same cases. This is possible if we consider the fact that the duality of $!S.\tau_1 \bowtie \tau_2$ implies the duality of $\tau_1 \bowtie \tau_2$.

Case $\tau_1 \bowtie !S.\tau_2$**:** As above.

Case $!S\tau_1 \bowtie !S'.\tau_2$**:** As above, notice that the reasoning is: By Def. 10.30, $!S\tau_1 \bowtie !S.\tau_2$ implies $\tau_1 \bowtie !S.\tau_2$, which in turn implies $!S\tau_1 \bowtie \tau_2$ and from here on, the reasoning is as the above mentioned cases.

Case (*Watch*)*:* In this case $\langle\Delta \diamond \Theta\rangle \curvearrowright_E \langle\Delta' \diamond \Theta'\rangle$ is deduced by Rule 6. of Def. 10.37. Then $\Delta = \Delta_1 \star_{ev} \Delta_2$. There are two cases two consider, depending on the result of the composition operator in Def. 10.26:

Case $\Delta_1 = \Delta^{end}$**:** This case is vacuously true since there is no reduction for a terminated environment.

Case $\Delta_1 \neq \Delta^{end}$**:** In this case, again we have two cases to consider, depending on whether $ev \in E$ or not:

Case $ev \in E$**:** In this case, we have by Def. 10.38 that $[\Delta_1 \star_{ev} \Delta_2]_E = \Delta_2$, since by assumption $ev \in E$, also, by assumption we have that $\Delta' = \Delta_2$, hence $\Delta' = [\Delta_1 \star_e \Delta_2]_E$, and applying the IH we have that $\langle\Delta \diamond \Theta\rangle \curvearrowright_E \langle\Delta' \diamond \Theta^{\emptyset}\rangle$, thus proving the first part of the lemma. Notice also that by Def. 10.26 we have that $dom(\Delta_1) = dom(\Delta_2)$ or the function would be undefined and hence $dom(\Delta') \subseteq dom(\Delta)$. Now, let us assume that $\langle\Delta_1 \star_{ev} \Delta_2 \diamond \Theta\rangle$ satisfies duality, we need to prove that $\langle[\Delta_1 \star_{ev} \Delta_2]_E \diamond \Theta^{\emptyset}\rangle$ also satisfies duality. For this we consider two arbitrary $s[\mathsf{p}] : T_{\mathsf{p}}, s[\mathsf{q}] : T_{\mathsf{q}} \in \Delta$. This means then that, as above we have two possibilities for each participant-type pair:

$$\langle\Delta_1 \star_{ev} \Delta_2 \diamond \Theta\rangle(s[\mathsf{p}]) = T_{\mathsf{p}} \vee \langle\Delta_1 \star_{ev} \Delta_2 \diamond \Theta\rangle(s[\mathsf{p}]) = !S.T_{\mathsf{p}}$$
$$\langle\Delta_1 \star_{ev} \Delta_2 \diamond \Theta\rangle(s[\mathsf{q}]) = T_{\mathsf{p}} \vee \langle\Delta_1 \star_{ev} \Delta_2 \diamond \Theta\rangle(s[\mathsf{q}]) = !S'.T_{\mathsf{q}}$$

Then, let $T_{\mathsf{p}} \upharpoonright \mathsf{q} = \tau_{\mathsf{p}}$ and $T_{\mathsf{q}} \upharpoonright \mathsf{p} = \tau_{\mathsf{q}}$, then, as above, we have the following cases:

$$\begin{array}{rcl} \tau_{\mathsf{p}} & \bowtie & \tau_{\mathsf{q}} \\ !S_{\mathsf{p}}.\tau_{\mathsf{p}} & \bowtie & \tau_{q} \\ \tau_{\mathsf{p}} & \bowtie & !S_{\mathsf{q}}.\tau_{\mathsf{q}} \\ !S_{\mathsf{p}}.\tau_{\mathsf{p}} & \bowtie & !S'_{\mathsf{q}}.\tau_{\mathsf{q}} \end{array}$$

Lastly, the proof proceeds as above by cases. Notice, however, that by Def. 10.26, one has that for all $c : T \in \Delta$ the type $T = \langle T_1, T_2\rangle^{ev}$. Furthermore, since duality for the watch type is defined pairwise (cf. Def. 10.30) and considering the fact that the reconditioning in the presence of ev does not change the environment Δ_2, we can conclude that $\langle \Delta_2 \diamond \Theta\rangle$ preserves duality, concluding this case.

Case $ev \notin E$**:** This case proceeds with a reasoning similar to the Case $\tau_1 = \mu\mathbf{t}.\tau_1''$ above: by IH and case analysis. We need to consider the full environment, and check only Δ_1, since Δ_2 remains untouched.

□

Lemma 10.45 (Suspension Lemma). *Let $\langle P, M, E\rangle\ddagger$ and $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. Then we have that $\langle \Delta \diamond \Theta\rangle \curvearrowright \langle [\Delta]_E \diamond \Theta^{\emptyset}\rangle$ and $\Gamma \vdash \langle [P]_E, M^{\emptyset}, \emptyset\rangle \triangleright \langle [\Delta]_E \diamond \Theta^{\emptyset}\rangle$. Moreover, if $\langle \Delta \diamond \Theta\rangle$ is coherent then $\langle [\Delta]_E \diamond \Theta^{\emptyset}\rangle$ is coherent.*

Proof (*see Page 275*). By induction on the definition of $\langle P, M, E\rangle\ddagger$. The basic cases correspond to the suspension rules $(pause)$, (out_s), (in_s) and (in_s^2), and the inductive cases to rules (par_s), $(watch_s)$ and (rec_s). We start with the basic cases.

Case $(pause)$**:** Assume $P = \mathsf{pause}.P'$. Hence $[P]_E = P'$. By assumption $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. This judgement is derived by Rule $\lfloor$Pause$\rfloor$, hence $\Delta = \mathsf{pause}.T$ and $[\Delta]_E = T$. By inversion on Rule $\lfloor$Pause$\rfloor$ we have $\Gamma \vdash M \triangleright \Theta$, $\mathsf{OG}(T)$, and $\Gamma \vdash \langle P', M^{\emptyset}, \emptyset\rangle \triangleright \langle T \diamond \Theta^{\emptyset}\rangle$. The latter is the required judgement $\Gamma \vdash \langle [P]_E, M^{\emptyset}, \emptyset\rangle \triangleright \langle [\mathsf{pause}.T]_E \diamond \Theta^{\emptyset}\rangle$. Moreover, $\langle \mathsf{pause}.T \diamond \Theta\rangle \curvearrowright \langle [\mathsf{pause}.T]_E \diamond \Theta^{\emptyset}\rangle = \langle T \diamond \Theta^{\emptyset}\rangle$ by Clause 1. of Def. 10.37. Assume now that $\langle \Delta \diamond \Theta\rangle$ is coherent. Then $\langle [\Delta]_E \diamond \Theta^{\emptyset}\rangle$ is coherent, since Condition 1. of Def. 10.32 follows from $\mathsf{OG}(T)$ and Condition 2. of Def. 10.32 follows from Lem. 10.44.

Case (out_s)**:** Assume $P = s[\mathsf{p}]!\langle e\rangle.P'$ and $M = M_0 \cup \{s[\mathsf{p}] : (w, \Pi)\}$. By assumption $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. This judgement is deduced by Rule $\lfloor$SendMore$\rfloor$, therefore, assuming $e \downarrow v$, $\Gamma \vdash v : S$ and $\Gamma \vdash w : S'$, it has the form:

$$\Gamma \vdash \langle s[\mathsf{p}]!\langle e\rangle.P', M_0 \cup \{s[\mathsf{p}] : (w, \Pi)\}, E\rangle \triangleright \langle s[\mathsf{p}] : \mathtt{tick}.!S.T \diamond \Theta_0, s[\mathsf{p}] : (!S', \Pi)\rangle$$

where $\Delta = s[\mathsf{p}] : \mathtt{tick}.!S.T$ and $\Theta = \Theta_0, s[\mathsf{p}] : (!S', \Pi)$. By reconditioning P and Δ we obtain $[P]_E = P = s[\mathsf{p}]!\langle e\rangle.P'$ and $[\Delta]_E = s[\mathsf{p}] :!S.T$. By inversion on Rule $\lfloor$SendMore$\rfloor$ we have $\Gamma \vdash \langle P', M_0^{\emptyset} \cup s[\mathsf{p}] : (v, \emptyset), \emptyset\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta_0^{\emptyset} \cup s[\mathsf{p}] : (S, \emptyset)\rangle$, where $\Gamma \vdash M_0 \triangleright \Theta_0$. This is the premise we need to apply Rule $\lfloor$SendFirst$\rfloor$ to $\langle s[\mathsf{p}]!\langle e\rangle.P', M_0^{\emptyset} \cup s[\mathsf{p}] : \epsilon, \emptyset\rangle = \langle [P]_E, M^{\emptyset}, \emptyset\rangle$. By this rule we deduce:

$$\Gamma \vdash \langle s[\mathsf{p}]!\langle e\rangle.P', M_0^{\emptyset} s[\mathsf{p}] : \epsilon, \emptyset\rangle \triangleright \langle s[\mathsf{p}] :!S.T \diamond \Theta_0^{\emptyset}, s[\mathsf{p}] : \mathtt{void}\rangle$$

that is, $\Gamma \vdash \langle [P]_E, M^\emptyset, \emptyset\rangle \triangleright \langle [\Delta]_E \diamond \Theta^\emptyset\rangle$, given that $M^\emptyset = M_0^\emptyset \cup s[\mathsf{p}] : \epsilon$, $[\Delta]_E = s[\mathsf{p}] : !S.T$ and $\Theta^\emptyset = \Theta_0^\emptyset, s[\mathsf{p}] : \mathtt{void}$ (the latter follows from $\Theta = \Theta_0, s[\mathsf{p}] : (!S', \Pi)$).

Again, we have $\langle \Delta \diamond \Theta\rangle \curvearrowright \langle [\Delta]_E \diamond \Theta^\emptyset\rangle$ by Clause 1. of Def. 10.37. What is left to show is coherence of $\langle [\Delta]_E \diamond \Theta^\emptyset\rangle$. It is easy to see that Condition 1. is satisfied, because $\mathsf{OG}(!S.T)$ implies $\mathsf{OG}([\Delta^{live}]_E)$. As for Condition 2., it follows again from Lem. 10.44.

Case (in_s)**:** Here $P = s[\mathsf{q}]?(\mathsf{p}, x).P'$ and $M = M_0 \cup s[\mathsf{p}] : \epsilon$. By assumption $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. This judgement is deduced by Rule $\lfloor\textsc{RcvNext}\rfloor$, hence it has the form:

$$\Gamma \vdash \langle s[\mathsf{q}]?(\mathsf{p}, x).P', M_0 \cup s[\mathsf{p}] : \epsilon, E\rangle \triangleright \langle s[\mathsf{q}] :?(\mathsf{p}, S).T \diamond \Theta_0, s[\mathsf{p}] : \mathtt{void}\rangle$$

We will call the derivation tree obtained by the previous hypothesis D_1.

Through reconditioning we obtain: $[P]_E = P = s[\mathsf{q}]?(\mathsf{p}, x).P'$ and we show that $\Gamma \vdash \langle P, M^\emptyset, E\rangle \triangleright \langle \Delta' \diamond \Theta^\emptyset\rangle$. This judgement is again deduced by Rule $\lfloor\textsc{RcvNext}\rfloor$, hence it has the form:

$$\Gamma \vdash \langle s[\mathsf{q}]?(\mathsf{p}, x).P', M^\emptyset \cup s[\mathsf{p}] : \epsilon, E\rangle \triangleright \langle s[\mathsf{q}] :?(\mathsf{p}, S).T' \diamond \Theta^\emptyset, s[\mathsf{p}] : \mathtt{void}\rangle$$

By inversion we have $\Gamma \vdash \langle P', M^\emptyset \cup s[\mathsf{p}] : (v, \{\mathsf{q}\}), \emptyset\rangle \triangleright \langle s[\mathsf{q}] : T' \diamond s[\mathsf{p}] : (S, \{q\})\rangle$ and $\Gamma \vdash v : S$. We also have that $\Gamma \vdash M^\emptyset \triangleright \Theta^\emptyset$, by applying the Rules in Fig. 10.15. The previously obtained derivation tree will be called D_2. We now proceed by induction on the structure of P': we will prove that $T' = \mathtt{trm}(T)^0_{\{p\}}$.

Case $P' = \mathbf{0}$**:** By using Rule $\lfloor\textsc{Inact}\rfloor$ we obtain $\Gamma \vdash \langle \mathbf{0}, M^\emptyset \cup s[\mathsf{p}] : (v, \{\mathsf{q}\}), \emptyset\rangle \triangleright \langle s[\mathsf{q}] : \mathtt{end} \diamond s[\mathsf{p}] : (S, \{q\})\rangle$ and by Fig. 10.19 we have that $\mathtt{end} = \mathtt{trm}(\mathtt{end})^0_{\{p\}}$.

Case $P' = \bar{a}[n]$ **or** $P' = a[\mathsf{p}](\alpha).P''$**:** This case is vacuously true as we are considering single session processes. Hence, there cannot be session initiations in a continuation.

Case $P' = s[\mathsf{q}]!\langle e\rangle.P''$**:** We distinguish the following two cases, depending on the memory M:

Case $M_0(s[\mathsf{q}]) \neq \epsilon$**:** We have that on D_1 we will have that $T = \mathtt{tick}.!S_1.T_1$, since this derivation is obtained by using Rule $\lfloor\textsc{SendMore}\rfloor$. For the D_2 we need to show then that $T' = \mathtt{trm}(T)^0_{\{\mathsf{p}\}}$; for this is enough to observe that $T' = \mathtt{trm}(T)^0_{\{p\}} =!S_1.T_1$, by Fig. 10.19, and hence, we can apply Rule $\lfloor\textsc{SendFirst}\rfloor$. Finally, we can conclude by IH.

Case $M_0(s[\mathsf{q}]) = \epsilon$**:** We have that on D_1 we will have that $T =!S_1.T_1$, since this derivation is obtained by using Rule $\lfloor\textsc{SendFirst}\rfloor$. By inversion we obtain:

$$\begin{aligned}\Gamma \vdash &\langle P'', M_0' \cup s[\mathsf{q}] : (v', \emptyset) \cup s[\mathsf{p}] : (v, \{\mathsf{q}\}), E\rangle \\ &\triangleright \langle s[\mathsf{q}] : T_1 \diamond \Theta', s[\mathsf{p}] : (v, \{\mathsf{q}\}, s[\mathsf{q}] : (S_1, \emptyset))\rangle\end{aligned}$$

with $M_0 = M_0' \cup s[\mathsf{q}] : (v', \emptyset)$ and $\Theta = \Theta', s[\mathsf{q}] : (S_1, \emptyset)$.

Notice now that on D_2 we have that the judgment is also deduced by Rule $\lfloor\textsc{SendFirst}\rfloor$, hence, by inversion: $\Gamma \vdash \langle P', M_1^\emptyset \cup s[\mathsf{q}] : (v', \emptyset) \cup$

$s[\mathsf{p}] : (v, \{\mathsf{q}\}), \emptyset\rangle \triangleright \langle s[\mathsf{q}] : T_1' \diamond \Theta_1^{\emptyset}, s[\mathsf{q}] : (S_1\emptyset), s[\mathsf{p}] : (S, \{q\})\rangle$, with $M^{\emptyset} = M_1^{\emptyset} \cup s[\mathsf{q}] : (v', \emptyset)$ and $\Theta^{\emptyset} = \Theta_1^{\emptyset}, s[\mathsf{q}] : (S_1\emptyset)$.

We know by Fig. 10.19 that $T' = \mathtt{trm}(T)^0_{\{p\}} = !S_1.T_1'$. We need then to show that $T_1' = \mathtt{trm}(T_1)^1_{\{\mathsf{p}\}}$; this is done by induction on P'':

Case $P'' = \mathbf{0}$**:** In this case again, by Rule $\lfloor$Inact$\rfloor$ on D_1 we obtain that $\Gamma \vdash \langle \mathbf{0}, M_1^{\emptyset} \cup s[\mathsf{q}] : (v', \emptyset) \cup s[\mathsf{p}] : (v, \{\mathsf{p}\}), \emptyset\rangle \triangleright \langle s[\mathsf{q}] : \mathtt{end} \diamond \Theta_1^{\emptyset}, s[\mathsf{q}] : (S_1\emptyset), s[\mathsf{p}] : (S, \{\mathsf{p}\})\rangle$ and since by Fig. 10.19, we have that $\mathtt{trm}(\mathtt{end})^1_{\{\mathsf{p}\}} = \mathtt{end}$, then we can conclude in D_2 with Rule $\lfloor$Inact$\rfloor$.

Case $P'' = \bar{a}[n]$ **or** $P' = a[\mathsf{p}](\alpha).P''$**:** This case is vacuously true as we are considering single session processes. Hence, there cannot be session initiations in a continuation.

Case $P'' = s[\mathsf{q}]!\langle e\rangle.P''$**:** In this case, on D_1 we use Rule $\lfloor$SendMore$\rfloor$, therefore, we have that $T_1 = \mathtt{tick}.!S_2.T_2$ and since $\mathtt{trm}(T_1)^1_{\{\mathsf{p}\}} = T_1 = \mathtt{tick}.!S_2.T_2$ we have, by Fig. 10.19, that $T_1' = \mathtt{trm}(T_1)^1_{\{\mathsf{p}\}}$. Therefore, we can conclude D_2 by applying Rule $\lfloor$SendMore$\rfloor$ and the IH.

Case $P'' = s[\mathsf{q}]?(\mathsf{r}, x).P''$**:** In this case, on D_1 we need to distinguish cases depending on r:

Case $\mathsf{r} \in \{\mathsf{p}\}$**:** In this case we have that $\mathsf{r} = \mathsf{p}$ and therefore, the judgment om D_1 is deduced from Rule $\lfloor$RcvMore$\rfloor$, which means that $T_1 = \mathtt{tick}.?(\mathsf{p}, S_2).T_2$. Hence,

$$T_1 = \mathtt{trm}(\mathtt{tick}.?(\mathsf{p}, S_2).T_2)^1_{\{\mathsf{p}\}} = \mathtt{tick}.?(\mathsf{p}, S_2).T_2$$

We can then conclude by applying IH.

Case $\mathsf{r} \notin \{\mathsf{p}\}$**:** In this case we need to distinguish cases depending on the memory M_0:

Case $M_0(s[\mathsf{r}]) = \epsilon$**:** In this case we have that on D_1 the judgment is deduced from Rule $\lfloor$RcvFirst$\rfloor$ and therefore $T_1 = ?(\mathsf{r}, S_2).T_2$ and $T_1' = ?(\mathsf{r}, S_2).T_2'$. By IH and Fig. 10.19 we have that $T_2' = \mathtt{trm}(T_2)^1_{\{\mathsf{p},\mathsf{r}\}}$ and therefore, $T_1' = \mathtt{trm}(?(\mathsf{r}, S_2).T_2)^1_{\{\mathsf{p}\}}$, which is what we wanted to prove.

Case $M_0(s[\mathsf{r}]) \neq \epsilon$**:** In this case we apply Rule $\lfloor$RcvMore$\rfloor$ on D_1 and therefore $T_1 = \mathtt{tick}.?(\mathsf{r}, S_2).T_2$. Observe then that in D_2 we will apply Rule $\lfloor$RcvFirst$\rfloor$ and $T_1' = ?(\mathsf{r}, S_2).T_2'$. Finally by IH and Fig. 10.19 we have that $\mathtt{trm}(T_1)^1_{\{\mathsf{p}\}} = ?(\mathsf{r}, S_2).T_2$ and $T_2' = T_2$, hence concluding the proof.

Case $P'' = \mathtt{rec}\, X\, .\, P'''$**:** This case concludes by inversion on Rule $\lfloor$Rec$\rfloor$ and applying the IH on the premises.

Case $P'' = \mathtt{pause}.\, P'''$**:** Note that since, by Fig. 10.19, function $\mathtt{trm}(\mathtt{pause}.T)^1_{\{\mathsf{p}\}} = \mathtt{pause}.T$, the case is straightforward by IH.

Case $P'' = P''' \mid Q$**,** $P'' = \mathtt{if}\; e\; \mathtt{then}\; P'''\; \mathtt{else}\; Q$**,** $P'' = \mathtt{emit}\; ev.\, P'''$ **and** $P'' = \mathtt{watch}\; ev\; \mathtt{do}\; P'''\{Q\}$**:** Again, these cases conclude by applying the IH on the premises of the Rules, obtained by inversion.

Case $P' = s[\mathsf{q}]?(\mathsf{r}, x).P''$: We distinguish cases depending on r:

Case $\mathsf{r} \in \{\mathsf{p}\}$: This case proceeds as above, by considering that in D_1 and D_2 we apply $\lfloor$RcvMore$\rfloor$. The equality is preserved, since this type is not changed by function $\mathtt{trm}(T)^0_{\{\mathsf{p}\}}$.

Case $\mathsf{r} \in \{\mathsf{p}\}$: We distinguish cases depending on the memory M_0:

Case $M_0(s[\mathsf{r}]) = \epsilon$: This case concludes by applying the IH, since we have that function $\mathtt{trm}(\cdot)^0_{\{\mathsf{p},\mathsf{r}\}}$ is applied recursively on the continuation T_1.

Case $M_0(s[\mathsf{r}]) \neq \epsilon$: This case concludes as Case $\mathsf{r} \in \{\mathsf{p}\}$, since the judgment on D_1 is deduced from $\lfloor$RcvMore$\rfloor$.

Case $P' = \mathtt{rec}\, X \,.\, P''$: This case proceeds by applying the IH on the premises obtained by inversion Rule $\lfloor$Rec$\rfloor$.

Case $P' = \mathtt{pause}.\, P''$: This case proceeds straightforward because $\mathtt{trm}(\cdot)^0_{\{\mathsf{p}\}}$ does not change type $T = \mathtt{pause}.T_1$ and both D_1, D_2 use Rule $\lfloor$Pause$\rfloor$ and conclude by IH.

Cases: $P' = P'' \mid Q$, $P' = \mathtt{if}\ e\ \mathtt{then}\ P''\ \mathtt{else}\ Q$, $P' = \mathtt{emit}\, ev.\, P''$, and $P' = \mathtt{watch}\, ev\ \mathtt{do}\ P''\{Q\}$. These three cases are concluded by applying the IH on the premises obtained by inversion in each of the respective Rules (i.e., $\lfloor$Conc$\rfloor$, $\lfloor$If$\rfloor$,$\lfloor$Emit$\rfloor$, $\lfloor$Watch$\rfloor$).

Note that once more, we have $\langle \Delta \diamond \Theta \rangle \curvearrowright \langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$ by Clause 3. of Def. 10.37. Moreover, $\langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$ is coherent: Condition 1. holds since, assuming that $\langle \Delta \diamond \Theta \rangle$ is coherent since, by assumption $\mathsf{OG}(?(\mathsf{p}, S)).T$ is true, and Condition 2. follows from Lem. 10.44.

Case (in^2_s)**:** Here $P = s[\mathsf{q}]?(\mathsf{p}, x).P'$ and $M = M_0 \cup \{s[\mathsf{p}] : (w, \Pi)\}$ with $\mathsf{q} \in \Pi$. By assumption $\Gamma \vdash \langle P, M, E \rangle \triangleright \langle \Delta \diamond \Theta \rangle$. This judgment is deduced by Rule $\lfloor$RcvMore$\rfloor$, hence it has the form:

$$\begin{aligned}\Gamma \vdash &\langle s[\mathsf{q}]?(\mathsf{p}, x).P', M_0 \cup s[\mathsf{p}] : (w, \Pi), E \rangle \\ &\triangleright \langle s[\mathsf{q}] : \mathtt{pause}.?(\mathsf{p}, S).T \diamond \Theta_0, s[\mathsf{p}] : (S', \Pi) \rangle\end{aligned}$$

Through reconditioning we obtain: $[P]_E = P = s[\mathsf{q}]?(\mathsf{p}, x).P'$ and $[\Delta]_E = s[\mathsf{p}] : ?(\mathsf{p}, S).T$. By inversion on Rule $\lfloor$RcvMore$\rfloor$ we have $\mathsf{q} \in \Pi$, $\Gamma \vdash v : S$, $\Gamma \vdash w : S'$, $\Gamma \vdash M_0 \triangleright \Theta_0$, $\mathsf{OG}(T)$, and $\Gamma, x : S \vdash \langle P', M^{\emptyset}_0 \cup s[\mathsf{p}] : (v, \{\mathsf{q}\}), \emptyset \rangle \triangleright \langle s[\mathsf{q}] : T \diamond \Theta^{\emptyset}_0, s[\mathsf{p}] : (S, \{\mathsf{q}\}) \rangle$.

These statements may now be used as premises for applying Rule $\lfloor$RcvNext$\rfloor$ to the configuration
$\langle [P]_E, M^{\emptyset}, \emptyset \rangle = \langle s[\mathsf{q}]?(\mathsf{p}, x).P', M^{\emptyset}_0 \cup s[\mathsf{p}] : \epsilon, \emptyset \rangle$. By this Rule we obtain:

$$\Gamma \vdash \langle s[\mathsf{q}]?(\mathsf{p}, x).P', M^{\emptyset}_0 \cup s[\mathsf{p}] : \epsilon, \emptyset \rangle \triangleright \langle s[\mathsf{q}] :?(\mathsf{p}, S).T \diamond \Theta^{\emptyset}_0,\ s[\mathsf{p}] : \mathtt{void} \rangle$$

This is the required judgment $\Gamma \vdash \langle [P]_E, M^{\emptyset}, \emptyset \rangle \triangleright \langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$. Note that once more, we have $\langle \Delta \diamond \Theta \rangle \curvearrowright \langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$ by Clause 1. of Def. 10.37. Moreover, $\langle [\Delta]_E \diamond \Theta^{\emptyset} \rangle$ is coherent: Condition 1. holds because $\mathsf{OG}(T)$, and Condition 2. follows from Lem. 10.44.

We now turn to the inductive cases, corresponding to rules (par_s), $(watch_s)$ and (rec_s):

Case (par_s)**:** Here $P = P_1 \mid P_2$, and $\langle P_1 \mid P_2, M, E\rangle\ddagger$ is deduced from $\langle P_1, M, E\rangle\ddagger$ and $\langle P_2, M, E\rangle\ddagger$. Notice that the two components P_1 and P_2 are run in the same memory M and set of events E. By assumption $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$. This judgment is deduced by Rule $\lfloor\text{Conc}\rfloor$, with $M_1 = M_2 = M$ and $E_1 = E_2 = E$, hence the deduction has the form:

$$\frac{\Gamma \vdash \langle P_i, M, E\rangle \triangleright \langle \Delta_i \diamond \Theta\rangle,\ i = 1, 2}{\Gamma \vdash \langle P_1 \mid P_2, M, E\rangle \triangleright \langle \Delta_1, \Delta_2 \diamond \Theta\rangle}$$

By induction, $\langle P_i, M, E\rangle\ddagger$ and $\Gamma \vdash \langle P_i, M, E\rangle \triangleright \langle \Delta_i \diamond \Theta\rangle$ imply $\Gamma \vdash \langle [P_i]_E, M^{\emptyset}, \emptyset\rangle \triangleright \langle [\Delta_i]_E \diamond \Theta^{\emptyset}\rangle$ for $i = 1, 2$. Moreover, $\langle \Delta_i \diamond \Theta\rangle \curvearrowright \langle [\Delta_i]_E \diamond \Theta^{\emptyset}\rangle$ and coherence of $\langle \Delta_i \diamond \Theta\rangle$ implies coherence of $\langle [\Delta_i]_E \diamond \Theta^{\emptyset}\rangle$. We want to show that $\Gamma \vdash \langle [P_1 \mid P_2]_E, M^{\emptyset}, \emptyset\rangle \triangleright \langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset}\rangle$, and that coherence of $\langle \Delta_1, \Delta_2 \diamond \Theta\rangle$ implies coherence of $\langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset}\rangle$. From Fig. 10.15, we have that $\Gamma \vdash M^{\emptyset} \triangleright \Theta^{\emptyset}$. We may then apply Rule $\lfloor\text{Conc}\rfloor$ to the premises $\Gamma \vdash M^{\emptyset} \triangleright \Theta^{\emptyset}$ and $\Gamma \vdash \langle [P_i]_E, M^{\emptyset}, \emptyset\rangle \triangleright \langle [\Delta_i]_E \diamond \Theta^{\emptyset}\rangle$ for $i = 1, 2$, to deduce $\Gamma \vdash \langle [P_1]_E \mid [P_2]_E, M^{\emptyset}, \emptyset\rangle \triangleright \langle [\Delta_1]_E, [\Delta_2]_E, \diamond \Theta^{\emptyset}\rangle$. By definition of reconditioning, $[P_1 \mid P_2]_E = [P_1]_E \mid [P_2]_E$ and $\langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset}\rangle = \langle [\Delta_1]_E, [\Delta_2]_E \diamond \Theta^{\emptyset}\rangle$. Hence we have the required judgment $\Gamma \vdash \langle [P_1 \mid P_2]_E, M^{\emptyset}, \emptyset\rangle \triangleright \langle [\Delta_1, \Delta_2]_E, \diamond \Theta^{\emptyset}\rangle$. Let us check now coherence preservation. Assuming $Co\,\langle \Delta_1, \Delta_2 \diamond \Theta\rangle$, we want to show $Co\,\langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset}\rangle$. To prove Condition 1., observe that $Co\,\langle \Delta_1, \Delta_2 \diamond \Theta\rangle$ implies $Co\,\langle \Delta_i \diamond \Theta\rangle$ for each $i = 1, 2$. Then by induction we have $Co\,\langle [\Delta_i]_E \diamond \Theta^{\emptyset}\rangle$.

This implies $\mathsf{OG}([\Delta_i^{live}]_E)$ for each i, from which we deduce that

$$\mathsf{OG}([\Delta_1^{live}]_E, [\Delta_2^{live}]_E)$$

holds, which implies $\mathsf{OG}([\Delta_1^{live}, \Delta_2^{live}]_E)$.

Moreover, from $\langle \Delta_i \diamond \Theta\rangle \curvearrowright \langle [\Delta_i]_E \diamond \Theta^{\emptyset}\rangle$ for $i = 1, 2$ we deduce $\langle \Delta_1, \Delta_2 \diamond \Theta\rangle \curvearrowright \langle [\Delta_1, \Delta_2]_E \diamond \Theta^{\emptyset}\rangle$ by Def. 10.37(Rule 4.). Then we may use Lem. 10.44 again to obtain Condition 2.

Case $(watch\_s)$**:** Here $P = \mathtt{watch}\,ev\ \mathtt{do}\ P_1\{P_2\}$ and $\langle \mathtt{watch}\,ev\ \mathtt{do}\ P_1\{P_2\}, M, E\rangle\ddagger$ is deduced from $\langle P_1, M, E\rangle\ddagger$. The judgment $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle \Delta \diamond \Theta\rangle$ is deduced by Rule $\lfloor\text{Watch}\rfloor$ and has the form:

$$\Gamma \vdash \langle \mathtt{watch}\,ev\ \mathtt{do}\ P_1\{P_2\}, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_1 \star_{ev} T_2 \diamond \Theta\rangle$$

By inversion on rule $\lfloor\text{Watch}\rfloor$ we obtain $\Gamma \vdash \langle P_1, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_1 \diamond \Theta\rangle$, as well as $\Gamma \vdash \langle P_2, M^{\emptyset}, E\rangle \triangleright \langle s[\mathsf{p}] : T_2 \diamond \Theta^{\emptyset}\rangle$ and $\mathsf{OG}(\Delta_2)$. Note that it is not the case that $T_1 = \mathsf{end}$ because in this case the configuration $\langle \mathtt{watch}\,ev\ \mathtt{do}\ P_1\{P_2\}, M, E\rangle$ would be terminated and not suspended.

Now, the form of $[P]_E$ and $[\Delta]_E$ will depend on the presence or absence of ev in E, namely:

1. If $ev \in E$ then $[P]_E = P_2$ and $[s[\mathsf{p}] : T_1 \star_{ev} T_2]_E = s[\mathsf{p}] : T_2$; in this case, by inversion on rule $\lfloor\text{Watch}\rfloor$ we obtain the required judgement $\Gamma \vdash \langle P_2, M^{\emptyset}, E\rangle \triangleright \langle s[\mathsf{p}] : T_2 \diamond \Theta^{\emptyset}\rangle$. Now, assume $Co\,\langle \Delta \diamond \Theta\rangle$. We want to show $Co\,\langle s[\mathsf{p}] : T_2 \diamond \Theta^{\emptyset}\rangle$. Condition 1. follows immediately from $\mathsf{OG}(T_2)$. As for Condition 2., notice that it follows from Lem. 10.44.

2. If $ev \notin E$, then $[P]_E = \mathtt{watch}\, ev\, \mathtt{do}\, [P_1]_E\{P_2\}$ and $[s[\mathsf{p}] : T_1 \star_{ev} T_2]_E = s[\mathsf{p}] : [T_1]_E \star_{ev} T_2$. By inversion on Rule $\lfloor$Watch$\rfloor$ we get $\Gamma \vdash \langle P_1, M, E\rangle \triangleright \langle s[\mathsf{p}] : T_1 \diamond \Theta\rangle$, and by applying the IH, we can obtain that $\Gamma \vdash \langle P_1, M, E\rangle \triangleright \langle s[\mathsf{p}] : [T_1]_E \diamond \Theta^{\emptyset}\rangle$ and we can use this hypothesis to conclude using Rule $\lfloor$Watch$\rfloor$. Now, for the second part, assume $Co\,\langle s[\mathsf{p}] : T_1 \star_{ev} T_2 \diamond \Theta\rangle$, we want to show that $Co\,\langle s[\mathsf{p}] : [T_1]_E \star_{ev} T_2 \diamond \Theta\rangle$. By inversion and IH we know then that $Co\,\langle [s[\mathsf{p}] : T_1]_E \diamond \Theta^{\emptyset}\rangle$ and since $T_1 = \mathsf{end}$ we also have that $\mathsf{OG}(T_2)$. Hence, it is true that $Co\,\langle s[\mathsf{p}] : [T_1]_E \star_{ev} T_2 \diamond \Theta^{\emptyset}\rangle$, satisfying Condition 1.. The second condition follows directly from Lem. 10.44.

Case (rec_s)**:** Here $P = (\mathtt{rec}\ X \,.\, P')$ and $\langle(\mathtt{rec}\ X \,.\, P'), M, E\rangle \ddagger$ is deduced from

$$\langle P'\{{}^{\mathsf{pause.\,rec}\, X\,.\,P'}/X\}), M, E\rangle \ddagger$$

The judgement $\Gamma \vdash \langle P, M, E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta\rangle$ is obtained from Rule $\lfloor$Rec$\rfloor$ and is as follows:

$$\Gamma \vdash \langle(\mathtt{rec}\ X \,.\, P), M, E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta\rangle$$

by inversion, we have that $\Gamma, X : \mathsf{pause}.T \vdash \langle P, M^{\emptyset}, E\rangle \triangleright \langle s[\mathsf{p}] : T \diamond \Theta^{\emptyset}\rangle$ and then by IH we have that $\Gamma, X : \mathsf{pause}.T \vdash \langle P, M^{\emptyset}, E\rangle \triangleright \langle [s[\mathsf{p}] : T]_E \diamond \Theta^{\emptyset}\rangle$. Then, it is enough to apply Rule $\lfloor$Rec$\rfloor$ with the previous hypothesis used twice to obtain:

$$\Gamma \vdash \langle(\mathtt{rec}\ X \,.\, P), M^{\emptyset}, E\rangle \triangleright \langle [s[\mathsf{p}] : T]_E \diamond \Theta^{\emptyset}\rangle$$

to proof the first part of the theorem. Supposing then $Co\,\langle\Delta \diamond \Theta\rangle$ and that $\langle\Delta \diamond \Theta\rangle \curvearrowright \langle[\Delta]_E \diamond \Theta^{\emptyset}\rangle$ then $\langle[\Delta]_E \diamond \Theta^{\emptyset}\rangle$ is coherent. This follows directly by IH and Lem. 10.44 as the proof is reduced to checking each type inside Δ.

□

Theorem 10.46 (Subject Reduction). *Let C be a reachable configuration and $C \rightsquigarrow^+ C'$. If $\Gamma \vdash C \triangleright \langle\Delta \diamond \Theta\rangle$ then $\Gamma \vdash C' \triangleright \langle\Delta' \diamond \Theta'\rangle$ and $\langle\Delta \diamond \Theta\rangle \Rightarrow^* \langle\Delta' \diamond \Theta'\rangle$ for some Δ', Θ'. Moreover if $\langle\Delta \diamond \Theta\rangle$ is coherent then $\langle\Delta' \diamond \Theta'\rangle$ is coherent.*

Proof (see Page 275). By induction on the length n of the reduction sequence $\rightsquigarrow^+$.

Case $n = 1$**:** We distinguish two more cases, depending on whether C is an initial configuration or not.

(a) $C = \langle P, M, E\rangle$ is initial. In this case

$$C = \langle a[1](\alpha_1).P_1 \mid \ldots \mid a[n](\alpha_n).P_n \mid \bar{a}[n], \emptyset, \emptyset\rangle$$

and $C \rightsquigarrow C'$ is a reduction deduced by Rule [Init]. Hence:

$$C' = (\nu s)\langle P_1\{{}^{s[\mathsf{1}]}/\alpha_1\} \mid \ldots \mid P_1\{{}^{s[\mathsf{n}]}/\alpha_n\}, M^{\emptyset}_s, \emptyset\rangle$$

Moreover, the type judgment $\Gamma \vdash C \triangleright \langle\Delta \diamond \Theta\rangle$ is deduced using Rules $\lfloor$MInit$\rfloor$, $\lfloor$MAcc$\rfloor$ and $\lfloor$Conc$\rfloor$ and thus it has the form (i) $\Gamma \vdash C \triangleright \langle\emptyset \diamond \emptyset\rangle$. From (i), using inversion on Rules $\lfloor$Conc$\rfloor$ and $\lfloor$MAcc$\rfloor$ we obtain (ii) $\Gamma \vdash \langle P_i\{{}^{s[\mathsf{i}]}/\alpha_i\}, M^{\emptyset}_s, \emptyset\rangle \triangleright \langle s[\mathsf{i}] : G_i\lfloor_{\,\mathsf{i}} \diamond \Theta^{\emptyset}_s\rangle$. By applying Rule $\lfloor$Conc$\rfloor$ to (ii) we deduce (iii) $\Gamma \vdash \langle P_1\{^{s[1]}/_{\alpha_1}\} \mid \ldots \mid P_n\{^{s[n]}/_{\alpha_n}\}, M_s^{\emptyset}, \emptyset\rangle \triangleright \langle \Delta \diamond \Theta_s^{\emptyset}\rangle$. By Prop. 10.34 we get (iv) $Co\langle \Delta \diamond \Theta_s^{\emptyset}\rangle$. We may then apply $\lfloor$Res$\rfloor$ to (iii) and (iv) to get $\Gamma \vdash (\nu c)\langle P_1\{^{s[1]}/_{\alpha_1}\} \mid \ldots \mid P_n\{^{s[n]}/_{\alpha_n}\}, M_s^{\emptyset}, \emptyset\rangle \triangleright \langle \emptyset \diamond \emptyset\rangle$. Since $\langle \Delta \diamond \Theta\rangle = \langle \Delta' \diamond \Theta'\rangle$ we trivially have $\langle \Delta \diamond \Theta\rangle \Rrightarrow^* \langle \Delta' \diamond \Theta'\rangle$ and there is nothing to prove about coherence.

(b) $C = (\nu s)\langle P, M, E\rangle$ and $C \rightsquigarrow C'$ is either a reduction deduced by Rule [Restr], or a tick transition deduced by Rule (tick). In both cases C' has the form $C' = (\nu s)\langle P', M', E'\rangle$ by Prop. 10.5. In the first case $C \longrightarrow C'$ is deduced by Rule [Restr] from $\langle P, M, E\rangle \longrightarrow \langle P', M', E'\rangle$ and we get the result by Lem. 10.43 (reduction lemma). In the second case $C \hookrightarrow_E C'$ is deduced by Rule (tick) from $\langle P, M, E\rangle\ddagger$, and we get the result by Lem. 10.45 (suspension lemma).

Case $n > 1$**:** Let

$$\begin{aligned}(C \rightsquigarrow (\nu s)\langle P_1, M_1, E_1\rangle \rightsquigarrow \cdots \rightsquigarrow (\nu s)\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle \\ \rightsquigarrow (\nu s)\langle P_n', M_n, E_n\rangle = C'\end{aligned}$$

By induction $\Gamma \vdash \langle P_{n-1}, M_{n-1}, E_{n-1}\rangle \triangleright \langle \Delta_{n-1} \diamond \Theta_{n-1}\rangle$ and $\langle \Delta \diamond \Theta\rangle \Rrightarrow^* \langle \Delta_{n-1} \diamond \Theta_{n-1}\rangle$. Moreover if $\langle \Delta \diamond \Theta\rangle$ is coherent then $\langle \Delta_{n-1} \diamond \Theta_{n-1}\rangle$ is coherent. Consider now the last reduction:

$$(\nu s)\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle \rightsquigarrow (\nu s)\langle P_n, M_n, E_n\rangle$$

There are two possibilities:

(1) $\rightsquigarrow = \hookrightarrow_E$. In this case:

$$(\nu s)\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle \hookrightarrow_E (\nu s)\langle [P_{n-1}]_E, M_{n-1}^{\emptyset}, \emptyset\rangle = (\nu s)\langle P_n, M_n, E_n\rangle$$

Since $(\nu s)\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle\ddagger$ if and only if $\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle\ddagger$ and

$$\Gamma \vdash \langle P_{n-1}, M_{n-1}, E_{n-1}\rangle \triangleright \langle \Delta_{n-1} \diamond \Theta_{n-1}\rangle$$

by Lem. 10.45 we deduce $\Gamma \vdash \langle [P_{n-1}]_{E_{n-1}}, M_{n-1}^{\emptyset}, \emptyset\rangle \triangleright \langle [\Delta_{n-1}]_{E_{n-1}} \diamond \Theta_{n-1}^{\emptyset}\rangle$, $\langle \Delta_{n-1} \diamond \Theta_{n-1}\rangle \curvearrowright_{E_{n-1}} \langle [\Delta_{n-1}]_{E_{n-1}} \diamond \Theta^{\emptyset}\rangle$ and $Co\,\langle [\Delta]_{E_{n-1}} \diamond \Theta^{\emptyset}\rangle$.

(2) $\rightsquigarrow = \longrightarrow$. In this case $(\nu s)\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle \longrightarrow (\nu s)\langle P_n, M_n, E_n\rangle$, and this reduction is deduced by Rule [Res] from $\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle \longrightarrow \langle P_n, M_n, E_n\rangle$, which in turn is deduced by the contextual Rules [Cont] and [Struct] from a reduction $\langle Q, M_{n-1}, E_{n-1}\rangle \longrightarrow \langle Q', M_n, E_n\rangle$ that is deduced by the computational rules only, where $P_{n-1} \equiv \mathcal{E}[Q]$ and $P_n \equiv \mathcal{E}[Q']$ for some evaluation context $\mathcal{E}$. This means that:

$$(\nu s)\langle P_{n-1}, M_{n-1}, E_{n-1}\rangle = (\nu s)\langle \mathcal{E}[Q], M_{n-1}, E_{n-1}\rangle$$

Then, assuming

$$\Gamma \vdash \langle Q, M_{n-1}, E_{n-1}\rangle \triangleright \langle \Delta_Q \diamond \Theta_Q\rangle$$

and $\Gamma \vdash \langle \mathcal{E}[\mathbf{0}], M_{n-1}, E_{n-1} \rangle \triangleright \langle \Delta'' \diamond \Theta'' \rangle$, by the typing rules [Conc] and [CRes] we have $\Delta_{n-1} = (\Delta_Q, \Delta'')$ and $\Theta_{n-1} = (\Theta_Q, \Theta'')$.

Since the reduction $\langle Q, M_{n-1}, E_{n-1} \rangle \longrightarrow \langle Q', M_n, E_n \rangle$ is deduced by computational rules only, by Lem. 10.43 we have $\langle \Delta_Q \diamond \Theta_Q \rangle \;\; \Rightarrow \;\; \langle \Delta'_Q \diamond \Theta'_Q \rangle$ and $\Gamma \vdash \langle Q', M_n, E_n \rangle \triangleright \langle \Delta'_Q \diamond \Theta'_Q \rangle$. Moreover, the same lemma states that if $\langle \Delta_Q, \Delta'' \diamond \Theta_Q, \Theta'' \rangle$ is coherent, then $\langle \Delta'_Q, \Delta'' \diamond \Theta'_Q, \Theta'' \rangle$ is coherent. From this we deduce that if $\langle \Delta_{n-1} \diamond \Theta_{n-1} \rangle$ is coherent then also $\langle \Delta' \diamond \Theta' \rangle$ is coherent.

□

Titles in the IPA Dissertation Series since 2016

Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems*. Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces*. Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network*. Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15