Bernardo Parente Coutinho Fernandes Toninho

B.Sc., M.Sc., Mestre em Engenharia Informática

A Logical Foundation for Session-based Concurrent Computation

Dissertação para obtenção do Grau de
Doutor em Engenharia Informática

Orientadores:	Frank Pfenning, Full Professor, Carnegie Mellon University Luís Caires, Prof. Catedrático, Universidade Nova de Lisboa

	Júri:
Presidente:	Prof. António Ravara (Universidade Nova de Lisboa)
Arguentes:	Prof. Vasco Vasconcelos (Universidade de Lisboa)
	Prof. Robert Harper (Carnegie Mellon University)
	Prof. Stephen Brookes (Carnegie Mellon University)
	Prof. Simon Gay (University of Glasgow)
Vogais:	Prof. Frank Pfenning (Carnegie Mellon University)
	Prof. Luís Caires (Universidade Nova de Lisboa)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Maio, 2015

A Logical Foundation for Session-based Concurrent Computation

For my parents, Isabel and Paulo.

ACKNOWLEDGEMENTS

I'd like to first and foremost thank my advisors (in no particular order), Luís Caires and Frank Pfenning. They have both been incredible mentors throughout this several year-long journey and it has been an honor and a privilege working with them, starting with Luis' classes on process calculi and Frank's classes on proof theory, teaching me about these seemingly unrelated things that turned out not to be unrelated at all; all the way up to our collaborations, discussions and conversations over the years, without which the work presented in this document would have never been possible. I'd also like to thank my thesis committee, for their participation, availability and their precious feedback.

I'd like to thank the many people on both sides of the Atlantic for discussions, talks and feedback over the years, be it directly related to this work or not. Some honorable mentions, from the West to the East: Robert Simmons, Dan Licata, Chris Martens, Henry DeYoung, Carsten Schürmann, Beniamino Accattoli, Yuxin Deng, Ron Garcia, Jason Reed, Jorge Pérez, Hugo Vieira, Luísa Lourenço, Filipe Militão, everyone in the CMU POP and FCT PLASTIC groups. Moving to Pittsburgh was a tremendously positive experience where I learned a great deal about logic, programming languages, proof theory and just doing great research in general, be it through classes, projects or discussions that often resulted in large strings of greek letters and odd symbols written on the walls. Moving back to Lisbon was a welcomed reunion with colleagues, friends and family, to which I am deeply grateful.

I thank the great teachers that helped me grow academically from an undergraduate learning programming to a graduate student learning proof theory (it takes a long time to go back to where you started). I especially thank Nuno Preguiça, João Seco and Robert Harper for their unique and invaluable contributions to my academic education.

I am deeply fortunate to have an exceptional group of friends that embrace and accept the things that make me who I am: Frederico, Mário, Cátia, Filipe, Tiago, Luísa, Hélio, Sofia, Pablo, Kattis. Thanks guys (and girls)! I must also mention all the friends and acquaintances I've made by coming back to my hobby from 1997, Magic: The Gathering. It never ceases to amaze me how this game of chess with variance can bring so many different people together. May you always draw better than your opponents, unless your opponent is me, of course.

Closer to home, I'd like to thank my family of grandparents, aunt and cousins, but especially to Mom and Dad. Without their love and support I'd never have gotten through graduate school across two continents over half a decade. A special note to my great-grandmother Clarice, the toughest woman to ever live, who could not see her great-grandson become a Doctor; to my grandfather António, who almost did; and to Fox, my near-human friend of 18 years, who waited for me to come home.

Thank you all, for everything…

ABSTRACT

Linear logic has long been heralded for its potential of providing a logical basis for concurrency. While over the years many research attempts were made in this regard, a Curry-Howard correspondence between linear logic and concurrent computation was only found recently, bridging the proof theory of linear logic and session-typed process calculus. Building upon this work, we have developed a theory of intuitionistic linear logic as a logical foundation for session-based concurrent computation, exploring several concurrency related phenomena such as value-dependent session types and polymorphic sessions within our logical framework in an arguably clean and elegant way, establishing with relative ease strong typing guarantees due to the logical basis, which ensure the fundamental properties of type preservation and global progress, entailing the absence of deadlocks in communication.

We develop a general purpose concurrent programming language based on the logical interpretation, combining functional programming with a concurrent, session-based process layer through the form of a contextual monad, preserving our strong typing guarantees of type preservation and deadlock-freedom in the presence of general recursion and higher-order process communication.

We introduce a notion of linear logical relations for session typed concurrent processes, developing an arguably uniform technique for reasoning about sophisticated properties of session-based concurrent computation such as termination or equivalence based on our logical approach, further supporting our goal of establishing intuitionistic linear logic as a logical foundation for session-based concurrency.

Keywords: Linear Logic, Session Types, Curry-Howard, Concurrency, Proofs-as-Programs

Resumo

Por muito tempo a lógica linear tem sido denotada pelo seu potencial de fornecer uma base lógica à concorrência. Apesar de ao longo dos anos muitos esforços de investigação terem sido efectuados para este efeito, uma correspondência Curry-Howard entre a lógica linear e computação concorrente apenas foi descoberta recentemente, ligando a teoria de prova da lógica linear e cálculos de processos com tipos de sessão. Construindo sobre este trabalho, desenvolvemos uma teoria de lógica linear intuicionista como uma fundação lógica para computação concorrente baseada em sessões, explorando vários fenómenos da teoria da concorrência, como tipos de sessão com dependências de valor e sessões polimórficas, no nosso sistema de uma forma limpa e elegante, estabelecendo com facilidade técnica garantias fortes sobre os programas bem tipificados devido à base lógica, assegurando as propriedades fundamentais de preservação de tipos e progresso global, resultando na ausência de deadlocks durante comunicação.

Desenvolvemos uma linguagem de programação concorrente de uso geral, baseada na interpretação lógica, combinando programação funcional com uma camada de processos concorrentes baseada em sessões através de uma mónade contextual, preservando as fortes garantias de preservação de tipos e ausência de deadlocks na presença de recursão generalizada e comunicação de processos de ordem superior.

Introduzimos uma noção de relações lógicas lineares para processos concorrentes com tipos de sessão, desenvolvendo uma técnica uniforme para estudar propriedades sofisticadas de computação concorrente como terminação ou equivalência baseadas na nossa abordagem lógica, suportando assim o nosso objectivo de estabelecer a lógica linear intuicionista como uma fundação lógica para a concorrência baseada em sessões.

Palavras-chave: Lógica Linear, Tipos de Sessão, Curry-Howard, Concorrência, Provas-como-Programas

CONTENTS

Contents **xiii**

List of Figures **xvii**

1 Introduction **1**

1.1 Modelling Message-Passing Concurrency 2

1.2 π-calculus and Session Types 5

1.3 Curry-Howard Correspondence 6

1.4 Linear Logic and Concurrency 7

1.5 Session-based Concurrency and Curry-Howard 8

1.6 Contributions 8

1.6.1 A Logical Foundation for Session-Based Concurrency 8

1.6.2 Towards a Concurrent Programming Language 10

1.6.3 Reasoning Techniques 10

I A Logical Foundation for Session-based Concurrency **11**

2 Linear Logic and Session Types **13**

2.1 Linear Logic and the Judgmental Methodology 13

2.2 Preliminaries 16

2.2.1 Substructural Operational Semantics 17

2.3 Propositional Linear Logic 18

2.3.1 Judgmental Rules 19

2.3.2 Multiplicatives 20

2.3.3 Additives 23

2.3.4 Exponential 25

2.3.5 Examples 27

2.4 Metatheoretical Properties 29

2.4.1 Correspondence of process expressions and π-calculus processes 30

2.4.2 Type Preservation 33

2.4.3 Global Progress 41

2.5 Summary and Further Discussion 44

3 Beyond Propositional Linear Logic 47

3.1 First-Order Linear Logic and Value-Dependent Session Types 48

3.1.1 Universal and Existential First-Order Quantification 48

3.2 Example 50

3.3 Metatheoretical Properties 51

3.3.1 Correspondence between process expressions and π-calculus processes . 52

3.3.2 Type Preservation 53

3.3.3 Global Progress 55

3.4 Value-Dependent Session Types – Proof Irrelevance and Affirmation 55

3.4.1 Proof Irrelevance 55

3.4.2 Affirmation 57

3.5 Second-Order Linear Logic and Polymorphic Session Types 60

3.5.1 Example 62

3.6 Metatheoretical Properties 62

3.6.1 Correspondence between process expressions and π-calculus processes . 63

3.6.2 Type Preservation 63

3.6.3 Global Progress 64

3.7 Summary and Further Discussion 65

II Towards a Concurrent Programming Language 67

4 Contextual Monadic Encapsulation of Concurrency 71

4.1 Typing and Language Syntax 72

4.2 Recursion 74

4.3 Examples 78

4.3.1 Stacks 78

4.3.2 A Binary Counter 80

4.3.3 An App Store 82

4.4 Connection to Higher-Order π-calculus 83

4.5 Metatheoretical Properties 84

4.5.1 Type Preservation 84

4.5.2 Global Progress 86

5 Reconciliation with Logic 87

5.0.3 Guardedness and Co-regular Recursion by Typing 89

5.0.4 Coregular Recursion and Expressiveness - Revisiting the Binary Counter 90

5.1 Further Discussion 93

III Reasoning Techniques 97

6 Linear Logical Relations 101

6.1 Unary Logical Relations . . . 101
6.1.1 Logical Predicate for Polymorphic Session Types . . . 104
6.1.2 Logical Predicate for Coinductive Session Types . . . 111
6.2 Binary Logical Relations . . . 122
6.2.1 Typed Barbed Congruence . . . 123
6.2.2 Logical Equivalence for Polymorphic Session Types . . . 127
6.2.3 Logical Equivalence for Coinductive Session Types . . . 132
6.3 Further Discussion . . . 136

7 Applications of Linear Logical Relations 139
7.1 Behavioral Equivalence . . . 139
7.1.1 Using Parametricity . . . 141
7.2 Session Type Isomorphisms . . . 142
7.3 Soundness of Proof Conversions . . . 144
7.4 Further Discussion . . . 150

8 Conclusion 153
8.1 Related Work . . . 154
8.2 Future Work . . . 156

Bibliography 159

A Proofs 165
A.1 Inversion Lemma . . . 165
A.2 Reduction Lemmas - Value Dependent Session Types . . . 170
A.3 Reduction Lemmas - Polymorphic Session Types . . . 171
A.4 Logical Predicate for Polymorphic Session Types . . . 172
A.4.1 Proof of Theorem 28 . . . 172
A.5 Proof Conversions . . . 173
A.5.1 Additional cases for the Proof of Theorem 48 . . . 174

List of Figures

1.1 π-calculus Syntax. . . . 3

1.2 π-calculus Labeled Transition System. . . . 4

1.3 Session Types 5

2.1 Intuitionistic Linear Logic Propositions 14

2.2 Translation from Process Expressions to π-calculus Processes 30

2.3 Process Expression Assignment for Intuitionistic Linear Logic 45

2.4 Process Expression SSOS Rules 46

3.1 A PDF Indexer 52

3.2 Type-Directed Proof Erasure. . . . 58

3.3 Type Isomorphisms for Erasure. . . . 58

4.1 The Syntax of Types 72

4.2 Typing Process Expressions and the Contextual Monad. . . . 75

4.3 Language Syntax. . . . 77

4.4 List Deallocation. . . . 80

4.5 A Concurrent Stack Implementation. . . . 80

4.6 A Bit Counter Network. . . . 81

4.7 Monadic Composition as Cut 83

6.1 π-calculus Labeled Transition System. . . . 102

6.2 Logical predicate (base case). . . . 107

6.3 Typing Rules for Higher-Order Processes 114

6.4 Logical Predicate for Coinductive Session Types - Closed Processes 115

6.5 Conditions for Contextual Type-Respecting Relations 126

6.6 Logical Equivalence for Polymorphic Session Types (base case). . . . 129

6.7 Logical Equivalence for Coinductive Session Types (base case) – abridged. . . . 134

7.1 A sample of process equalities induced by proof conversions 147

A.1 Process equalities induced by proof conversions: Classes (A) and (B) 174

A.2 Process equalities induced by proof conversions: Class (C) 175

A.3 Process equalities induced by proof conversions: Class (D). . . . 176

A.4 Process equalities induced by proof conversions: Class (E). 177

CHAPTER 1

INTRODUCTION

Over the years, computation systems have evolved from monolithic single-threaded machines to concurrent and distributed environments with multiple communicating threads of execution, for which writing correct programs becomes substantially harder than in the more traditional sequential setting. These difficulties arise as a result of several issues, fundamental to the nature of concurrent and distributed programming, such as: the many possible *interleavings of executions*, making programs hard to test and debug; *resource management*, since often concurrent programs must interact with (local or remote) resources in an orderly fashion, which inherently introduce constraints in the level of concurrency and parallelism a program can have; and *coordination*, since the multiple execution flows are intended to work together to produce some ultimate goal or result, and therefore must proceed in a coordinated effort.

Concurrency theory often tackles these challenges through abstract language-based models of concurrency, such as process calculi, which allow for reasoning about concurrent computation in a precise way, enabling the study of the behavior and interactions of complex concurrent systems. Much like the history of research surrounding the λ-calculus, a significant research effort has been made to develop type systems for concurrent calculi (the most pervasive being the π-calculus) that, by disciplining concurrency, impose desirable properties on well-typed programs such as deadlock-freedom or absence of race conditions.

However, unlike the (typed) λ-calculus which has historically been known to have a deep connection with intuitionistic logic, commonly known as the Curry-Howard correspondence, no equivalent connection was established between the π-calculus and logic until quite recently. One may then wonder why is such a connection important, given that process calculi have been studied since the early 1980s, with the π-calculus being established as the *lingua franca* of interleaving concurrency calculi in the early 1990s, and given the extensive body of work that has been developed based on these calculi, despite the lack of foundations based on logic.

The benefits of a logical foundation in the style of Curry-Howard for interleaving concurrency are various: on one hand, it entails a form of *canonicity* of the considered calculus by connecting it with proof theory and logic. Pragmatically, a logical foundation opens up the possibility of employing established techniques from logic to the field of concurrency theory, potentially providing elegant new means of reasoning and representing concurrent phenomena, which in turn enable the development of techniques to ensure stronger correctness and reliability of concurrent software. Fundamentally, a logical foundation allows for a compositional and incremental study of new language features, since the grounding in logic ensures that such extensions do not harm previously obtained results. This dissertation seeks to support the following:

Thesis Statement: Linear logic, specifically in its intuitionistic formulation, is a suitable logical foundation for message-passing concurrent computation, providing an elegant framework in which to express and reason about a multitude of naturally occurring phenomena in such a concurrent setting.

1.1 Modelling Message-Passing Concurrency

Concurrency is often divided into two large models: shared-memory concurrency and message-passing concurrency. The former enables communication between concurrent entities through modification of memory locations that are shared between the entities; whereas in the latter the various concurrently executing components communicate by exchanging messages, either synchronously or asynchronously.

Understanding and reasoning about concurrency is ongoing work in the research community. In particular, many language based techniques have been developed over the years for both models of concurrency. For shared-memory concurrency, the premier techniques are those related to (concurrent) separation logic [56], which enables formal reasoning about memory configurations that may be shared between multiple entities. For message-passing concurrency, which is the focus of this work, the most well developed and studied techniques are arguably those of *process calculi* [19, 68]. Process calculi are a family of formal language that enable the precise description of concurrent systems, dubbed *processes*, by modelling interaction through communication across an abstract notion of communication channels. An important and appealing feature of process calculi is their mathematical and algebraic nature: processes can be manipulated via certain algebraic laws, which also enable formal reasoning about process *behavior* [50, 70, 71].

While many such calculi have been developed over the years, the *de facto* standard process calculus is arguably the π-calculus [52, 68], a language that allows modelling of concurrent systems that communicate over channels that may themselves be generated dynamically and passed in communication. Typically, the π-calculus also includes replication (i.e. the ability to spawn an arbitrary number of parallel copies of a process), which combined with channel generation and passing makes the language a Turing complete model of concurrent, message-passing computation. Many variants of the π-calculus have been developed in the literature (a comprehensive study can be found in [68]), tailored to the study of specific features such as asynchronous communication, higher-order process communication [53], security protocol modelling [1], among many others.

We now give a brief introduction to *a* π-calculus, the syntax of which is given in Fig. 1.1 (for purposes that will be made clear in Chapter 2, we extend the typical π-calculus syntax with a

$$\begin{array}{llllllllllll} P & ::= & \mathbf{0} & \mid & P \mid Q & \mid & (\nu y)P & \mid & x\langle y\rangle.P & \mid & x(y).P \\ & & !x(y).P & \mid & [x \leftrightarrow y] & \mid & x.\mathsf{inl};P & \mid & x.\mathsf{inr};P & \mid & x.\mathsf{case}(P,Q) \end{array}$$

Figure 1.1: π-calculus Syntax.

forwarder $[x \leftrightarrow y]$ reminiscent of the forwarders used in the internal mobility π-calculus [67], or explicit fusions of [57]). The so-called static fragment of the π-calculus consists of the inactive process $\mathbf{0}$; the parallel composition operator $P \mid Q$, which composes in parallel processes P and Q; and the scope restriction $(\nu y)P$, binding channel y in P. The communication primitives are the output and input prefixed processes, $x\langle y\rangle.P$, which sends channel y along x and $x(y).P$ which inputs along x and binds the received channel to y in P. We restrict general unbounded replication to an input-guarded form $!x(y).P$, denoting a process that waits for inputs on x and subsequently spawns a replica of P when such an input takes place. The channel forwarding construct $[x \leftrightarrow y]$ equates the two channel names x and y. We also consider (binary) guarded choice $x.\mathsf{case}(P,Q)$ and the two corresponding selection constructs $x.\mathsf{inl};P$, which selects the left branch, and $x.\mathsf{inr};P$, which selects the right branch.

For any process P, we denote the set of *free names* of P by $\mathit{fn}(P)$. A process is *closed* if it does not contain free occurrences of names. We identify processes up to consistent renaming of bound names, writing $\equiv_\alpha$ for this congruence. We write $P\{x/y\}$ for the process obtained from P by capture avoiding substitution of x for y in P.

The operational semantics of the π-calculus are usually presented in two forms: reduction semantics, which describe how processes react internally; and labelled transition semantics, which specify how processes may interact with their environment. Both semantics are defined up-to a congruence $\equiv$ which expresses basic structural identities of processes, dubbed *structural congruence*.

Definition 1 (Structural Congruence)**.** Structural congruence *($P \equiv Q$), is the least congruence relation on processes such that*

$$\begin{array}{llll} P \mid \mathbf{0} \equiv P & (S\mathbf{0}) & P \equiv_\alpha Q \Rightarrow P \equiv Q & (S\alpha) \\ P \mid Q \equiv Q \mid P & (S|C) & P \mid (Q \mid R) \equiv (P \mid Q) \mid R & (S|A) \\ (\nu x)\mathbf{0} \equiv \mathbf{0} & (S\nu\mathbf{0}) & x \notin \mathit{fn}(P) \Rightarrow P \mid (\nu x)Q \equiv (\nu x)(P \mid Q) & (S\nu|) \\ (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & (S\nu\nu) & [x \leftrightarrow y] \equiv [y \leftrightarrow x] & (S\leftrightarrow) \end{array}$$

Definition 2 (Reduction)**.** Reduction *($P \rightarrow Q$), is the binary relation on processes defined by:*

$$\begin{array}{ll} x\langle y\rangle.Q \mid x.(z)P \rightarrow Q \mid P\{y/z\} & (RC) \\ x\langle y\rangle.Q \mid !x.(z)P \rightarrow Q \mid P\{y/z\} \mid !x.(z)P & (R!) \\ x.\mathsf{inl};P \mid x.\mathsf{case}(Q,R) \rightarrow P \mid Q & (RL) \\ x.\mathsf{inr};P \mid x.\mathsf{case}(Q,R) \rightarrow P \mid R & (RR) \\ Q \rightarrow Q' \Rightarrow P \mid Q \rightarrow P \mid Q' & (R|) \\ P \rightarrow Q \Rightarrow (\nu y)P \rightarrow (\nu y)Q & (R\nu) \\ (\nu x)(P \mid [x \leftrightarrow y]) \rightarrow P\{y/x\} \qquad (x \neq y) & (R\leftrightarrow) \\ P \equiv P' \wedge P' \rightarrow Q' \wedge Q' \equiv Q \Rightarrow P \rightarrow Q & (R\equiv) \end{array}$$

$$\frac{P \xrightarrow{\alpha} Q}{(\boldsymbol{\nu} y)P \xrightarrow{\alpha} (\boldsymbol{\nu} y)Q}(\mathsf{res}) \qquad \frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}(\mathsf{par}) \qquad \frac{P \xrightarrow{\overline{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}(\mathsf{com})$$

$$\frac{P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} y)(P' \mid Q')}\ (\mathsf{close}) \qquad \frac{P \xrightarrow{\overline{x\langle y\rangle}} Q}{(\boldsymbol{\nu} y)P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} Q}\ (\mathsf{open}) \qquad \overline{x}\langle y\rangle.P \xrightarrow{\overline{x\langle y\rangle}} P\ (\mathsf{out})$$

$$x(y).P \xrightarrow{x(z)} P\{z/y\}\ (\mathsf{in}) \quad !x(y).P \xrightarrow{x(z)} P\{z/y\} \mid !x(y).P\ (\mathsf{rep}) \quad x.\mathsf{inl};P \xrightarrow{\overline{x.\mathsf{inl}}} P\ (\mathsf{lout})$$

$$x.\mathsf{inr};P \xrightarrow{\overline{x.\mathsf{inr}}} P\ (\mathsf{rout}) \quad x.\mathsf{case}(P,Q) \xrightarrow{x.\mathsf{inl}} P\ (\mathsf{lin}) \quad x.\mathsf{case}(P,Q) \xrightarrow{x.\mathsf{inr}} Q\ (\mathsf{rin})$$

$$(\boldsymbol{\nu} x)(P \mid [x \leftrightarrow y]) \xrightarrow{\tau} P\{y/x\}\ (\mathsf{link})$$

Figure 1.2: π-calculus Labeled Transition System.

We adopt the so-called early transition system for the π-calculus [68] extended with the appropriate labels and transition rules for the choice constructs. A transition $P \xrightarrow{\alpha} Q$ denotes that process P may evolve to process Q by performing the action represented by the label α. Transition labels are given by

$$\alpha ::= \overline{x\langle y\rangle} \mid x(y) \mid \overline{(\nu y)x\langle y\rangle} \mid x.\mathsf{inl}\ \mid x.\mathsf{inr}\ \mid \overline{x.\mathsf{inl}}\ \mid \overline{x.\mathsf{inr}}\ \mid \tau$$

Actions are input $x(y)$, the left/right offers $x.\mathsf{inl}$ and $x.\mathsf{inr}$, and their matching co-actions, respectively the output $\overline{x\langle y\rangle}$ and bound output $\overline{(\nu y)x\langle y\rangle}$ actions, and the left/ right selections $\overline{x.\mathsf{inl}}$ and $\overline{x.\mathsf{inr}}$. The bound output $\overline{(\nu y)x\langle y\rangle}$ denotes extrusion of a fresh name y along (channel) x. Internal action is denoted by τ. In general an action α ($\overline{\alpha}$) requires a matching $\overline{\alpha}$ (α) in the environment to enable progress, as specified by the transition rules. For a label α, we define the sets $fn(\alpha)$ and $bn(\alpha)$ of free and bound names, respectively, as follows: in $\overline{x\langle y\rangle}$ and $x(y)$ both x and y are free; in $\overline{x.\mathsf{inl}}$, $\overline{x.\mathsf{inr}}$, $x.\mathsf{inl}$, and $x.\mathsf{inr}$, x is free; in $(\nu y)x\langle y\rangle$, x is free and y is bound. We denote by $s(\alpha)$ the subject of α (e.g., x in $x\langle y\rangle$).

Definition 3 (Labeled Transition System). *The relation* labeled transition *($P \xrightarrow{\alpha} Q$) is defined by the rules in Figure 6.1, subject to the side conditions: in rule* (res)*, we require $y \notin fn(\alpha)$; in rule* (par)*, we require $bn(\alpha) \cap fn(R) = \emptyset$; in rule* (close)*, we require $y \notin fn(Q)$; in rule* (link)*, we require $x \neq y$. We omit the symmetric versions of rules* (par)*,* (com)*,* (close) *and* (link)*.*

We can make precise the relation between reduction and τ-labeled transition [68], and closure of labeled transitions under $\equiv$ as follows. We write $\rho_1\rho_2$ for relation composition (e.g., $\xrightarrow{\tau}\equiv$).

Proposition 1.

1. *If $P \equiv\xrightarrow{\alpha} Q$, then $P \xrightarrow{\alpha}\equiv Q$*

2. *$P \to Q$ iff $P \xrightarrow{\tau}\equiv Q$.*

$$
\begin{array}{rcll}
A, B, C & ::= & A \multimap B & \text{input channel of type } A \text{ and continue as } B \\
& \mid & A \otimes B & \text{output fresh channel of type } A \text{ and continue as } B \\
& \mid & \mathbf{1} & \text{terminate} \\
& \mid & A \mathbin{\&} B & \text{offer choice between } A \text{ and } B \\
& \mid & A \oplus B & \text{provide either } A \text{ or } B \\
& \mid & !A & \text{provide replicable session of type } A
\end{array}
$$

Figure 1.3: Session Types

As in the history of the λ-calculus, many type systems for the π-calculus have been developed over the years. The goal of these type systems is to discipline communication in some way as to avoid certain kinds of errors. While the early type systems for the π-calculus focused on assigning types to values communicated on channels (e.g. the type of channel c states that only integers can be communicated along c), and on assigning input and output capabilities to channels (e.g. process P can only send integers on channel c and process Q can only receive), arguably the most important family of type systems developed for the π-calculus are *session types*.

1.2 π-calculus and Session Types

The core idea of session types is to structure communication between processes around the concept of a *session*. A (binary) session consists of a description of the interactive behavior between two components of a concurrent system, with an intrinsic notion of duality: When one component sends, the other receives; when one component offers a choice, the other chooses. Another crucial point is that a session is stateful insofar as it is meant to evolve over time (e.g. "input an integer and afterwards output a string") until all its codified behavior is carried out. A session *type* [41, 43] codifies the intended session that must take place over a given communication channel at the type level, equating type checking with a high-level form of communication protocol compliance checking.

The communication idioms that are typically captured in session types are input and output behavior, choice and selection, replication and recursive behavior. It is also common for session type systems to have a form of *session delegation* (i.e. delegating a session to another process via communication). Moreover, session types can provide additional guarantees on system behavior than just adhering to the ascribed sessions, such as *deadlock absence* and *liveness* [26]. We present a syntax for session types in Fig. 1.3, as well as the intended meaning of each type constructor. We use a slightly different syntax than that of the original literature on session types [41, 43], which conveniently matches the syntax of propositions in (intuitionistic) linear logic. We make this connection precise in Chapters 2 and 3.

While session types are indeed a powerful tool for the structuring of communication-centric programs, their theory is fairly complex, especially in systems that guarantee the absence of deadlocks, which require sophisticated causality tracking mechanisms. Another issue that arises in the original theory of session types is that it is often unclear how to consider new primitives or language features without harming the previously established soundness results. Moreover, the behavioral theory (in the sense of behavioral equivalence) of session typed systems is also quite intricate, requiring sophisticated bisimilarities which are hard to reason about, and from a technical standpoint, for which establishing the desirable property of contextuality or congruence (i.e. equivalence in any context) is a challenging endeavour.

1.3 Curry-Howard Correspondence

At the beginning of Chapter 1 we mentioned *en passant* the connection of typed λ-calculus and intuitionistic logic, commonly known as the Curry-Howard correspondence, and the historical lack of such a connection between the π-calculus and logic (until the recent work of [16]). For the sake of completeness, we briefly detail the key idea behind this correspondence and its significance.

The Curry-Howard correspondence consists of an identification between formal proof calculi and type systems for models of computation. Historically, this connection was first identified by Haskell Curry [23] and William Howard [44]. The former [23] observed that the types of the combinators of combinatory logic corresponded to the axiom schemes of Hilbert-style deduction systems for implication. The latter [44] observed that the proof system known as natural deduction for implicational intuitionistic logic could be directly interpreted as a typed variant of the λ-calculus, identifying implication with the simply-typed λ-calculus' arrow type and the dynamics of proof simplification with evaluation of λ-terms.

While we refrain from the full technical details of this correspondence, the key insight is the idea of identifying proofs in logic as programs in the λ-calculus, and vice-versa, thus giving rise to the fundamental concepts of proofs-as-programs and programs-as-proofs (a somewhat comprehensive survey can be found in [82]). The basic correspondence identified by Curry and Howard enabled the development of a new class of formal systems designed to act as both a proof system and as a typed programming language, such as Martin-Löf's intuitionistic type theory [49] and Coquand's Calculus of Constructions [22].

Moreover, the basis of this interpretation has been shown to scale well beyond the identification of implication and the function type of the λ-calculus, enabling an entire body of research devoted to identifying the *computational meaning* of extensions to systems of natural deduction such as Girard's System F [35] as a language of second-order propositional logic; modal necessity and possibility as staged computation [24] and monadic types for effects [80], among others. Dually, it also becomes possible to identify the *logical meaning* of computational phenomena.

Thus, in light of the substantial advances in functional type theories based on the Curry-Howard correspondence, the benefits of such a correspondence for concurrent languages become apparent. Notably, such a connection enables a new logically motivated approach to concurrency theory, as well as the compositional and incremental study of concurrency related phenomena, providing new techniques to ensure correctness and reliability of concurrent software.

1.4 Linear Logic and Concurrency

In the concurrency theory community, linearity has played a key role in the development of typing systems for the π-calculus. Linearity was used in early type systems for the π-calculus [47] as a way of ensuring certain desirable properties. Ideas from linear logic also played a key role in the development of session types, as acknowledged by Honda [41, 43].

Girard's linear logic [36] arises as an effort to marry the dualities of classical logic and the constructive nature of intuitionistic logic by rejecting the so-called structural laws of weakening ("I need not use all assumptions in a proof") and contraction ("If I assume something, I can use it multiple times"). Proof theoretically, this simple restriction turns out to have profound consequences in the meaning of logical connectives. Moreover, the resulting logic is one where assumptions are no longer persistent immutable objects but rather resources that interact, transform and are consumed during inference. Linear logic divides conjunction into two forms, which in linear logic terminology are called additive (usually dubbed "with", written &) and multiplicative (dubbed "tensor", and written $\otimes$), depending on how resources are used to prove the conjunction. Additive conjunction denotes a pair of resources where one must choose which of the elements of the pair one will use, although the available resources must be able to realize (i.e. prove) both elements. Multiplicative conjunction denotes a pair of resources where both resources *must* be used, since the available resources simultaneously realize both elements and all resources must be consumed in a valid inference (due to the absence of weakening and contraction). In classical linear logic there is also a similar separation in disjunction, while in the intuitionistic setting we only have additive disjunction $\oplus$ which denotes an alternative between two resources (we defer from a precise formulation of linear logic for now).

The idea of propositions as mutable resources that evolve independently over the course of logical inference sparked interest in using linear logic as a logic of concurrent computation. This idea was first explored in the work of Abramsky et. al [3], which developed a computational interpretation of linear logic proofs, identifying them with programs in a linear λ-calculus with parallel composition. In his work, Abramsky gives a faithful proof term assignment to classical linear logic sequent calculus, identifying proof composition with parallel composition. This proofs-as-processes interpretation was further refined by Bellin and Scott [8], which mapped classical linear logic proofs to processes in the synchronous π-calculus with prefix commutation as a structural process identity. Similar efforts were developed more recently in [42], connecting polarised proof-nets with an (I/O) typed π-calculus, and in [28] which develops ludics as a model for the finitary linear π-calculus. However, none of these interpretations provided a true Curry-Howard correspondence insofar as they either did not identify a type system for which linear logic propositions served as type constructors (lacking the propositions-as-types part of the correspondence); or develop a connection with only a very particular formulation of linear logic (such as polarised proof-nets).

Given the predominant role of session types as a typing system for the π-calculus and their inherent notion of evolving state, in hindsight, its connections with linear logic seem almost inescapable. However, it wasn't until the recent work of Caires and Pfenning [16] that this connection was made precise. The work of [16] develops a Curry-Howard correspondence between session types and *intuitionistic* linear logic, identifying (typed) processes with proofs, session types with linear logic propositions and process reduction with proof reduction. Following the discovery in the context of intuitionistic linear logic, classical versions of this correspondence have also been described [18, 81].

1.5 Session-based Concurrency and Curry-Howard

While we have briefly mentioned some of the conceptual benefits of a correspondence in the style of Curry-Howard for session-based concurrency in the previous sections, it is important to also emphasize some of the more pragmatic considerations that can result from the exploration and development of such a correspondence, going beyond the more foundational or theoretical benefits that follow from a robust logical foundation for concurrent calculi.

The identification of computation with proof reduction inherently bestows upon computation the good theoretical properties of proof reduction. Crucially, as discussed in [16], it enables a very natural and clean account of a form of *global progress* for session-typed concurrent processes, entailing the absence of *deadlocks* – “communication never gets stuck” – in typed communication protocols. Additionally, logical soundness entails that proof reduction is a finite procedure, which in turn enables us to potentially ensure that typed communication is not only *deadlock-free* but also *terminating*, a historically challenging property to establish in the literature.

Moreover, as the logical foundation scales beyond simple (session) types to account for more advanced typing disciplines such as polymorphism and coinductive types (mirroring the developments of Curry-Howard for functional calculi), we are expected to preserve such key properties, enabling the development of session-based concurrent programming languages where deadlocks are excluded as part of the typing discipline, even in the presence of sophisticated features such as mobile or polymorphic code.

1.6 Contributions

The three parts of this dissertation aim to support three fundamental aspects of our central thesis of using linear logic as a logical foundation for message-passing concurrent computation: (1) the development and usage of linear logic as a logical foundation for session-based concurrency, able to account for multiple relevant concurrency-theoretic phenomena; (2) the ability of using the logical interpretation as the basis for a concurrent programming language that cleanly integrates functional and session-based concurrent programming; and (3), developing effective techniques for reasoning about concurrent programs based on this logical foundation.

1.6.1 A Logical Foundation for Session-Based Concurrency

The first major contribution of Part I of the dissertation is the reformulation and refinement of the foundational work of [16], connecting linear logic and session types, which we use in this dissertation as a basis to explain and justify phenomena that arise naturally in message-passing concurrency, developing the idea of using linear logic as a logical foundation for message-passing concurrent computation in general. The second major contribution is the scaling of the interpretation of [16] beyond the confines of simple session types (and even the π-calculus), being able to also account for richer and more sophisticated phenomena such as dependent session types [75] and parametric polymorphism [17], all the while doing so in a technically elegant way and preserving the fundamental type safety properties of *session fidelity* and *global progress*, ensuring the safety of typed communication disciplines. While claims of elegance are by their very nature subjective, we believe it is possible to support these claims by showing how naturally and easily the interpretation can account for these phenomena that traditionally require quite intricate technical developments.

More precisely, in Chapter 2 we develop the interpretation of linear logic as session types that serves as the basis for the dissertation. The interpretation diverges from that of [16] in that it does not commit to the π-calculus *a priori*, developing a proof term assignment that can be used as a language for session-typed communication, but also maintaining its connections to the π-calculus. The goal of this "dual" development, amongst other technical considerations, is to further emphasize that the connection of linear logic and session-typed concurrency goes beyond that of having a π-calculus syntax and semantics. While we may use the π-calculus assignment when it is convenient to do so (and in fact we do precisely this in Part III), we are not necessarily tied to the π-calculus. This adds additional flexibility to the interpretation since it enables a form of "back and forth" reasoning between π-calculus and the faithful proof term assignment that is quite natural in proofs-as-programs approaches. We make the connections between the two term assignments precise in Section 2.4.1, as well as develop the metatheory for our assignment (Section 2.4), establishing properties of type preservation and global progress.

In Chapter 3 we show how to scale the interpretation to account for more sophisticated concurrent phenomena by developing the concepts of value dependent session types (Section 3.1) and parametric polymorphism (Section 3.5) as they arise by considering first and second-order intuitionistic linear logic, respectively. These are natural extensions to the interpretation that provide a previously unattainable degree of expressiveness within a uniform framework. Moreover, the logical nature of the development ensures that type preservation and global progress uniformly extend to these richer settings.

We note that the interpretation generalizes beyond these two particular cases. For instance, in [76] the interpretation is used to give a logically motivated account of parallel evaluation strategies on λ-terms through canonical embeddings of intuitionistic logic in linear logic. One remarkable result is that the resulting embeddings induce a form of sharing (as in futures, relaxing the sequentiality constraints of call-by-value and call-by-need) and copying (as in call-by-name) parallel evaluation strategies on λ-terms, the latter being reminiscent of Milner's original embedding of the λ-calculus in the π-calculus [51].

1.6.2 Towards a Concurrent Programming Language

A logical foundation must also provide the means of expressing concurrent computation in a natural way that preserves the good properties one obtains from logic. We defend this idea in Part II of the dissertation by developing the basis of a concurrent programming language that combines functional and session-based concurrent programs via a monadic embedding of session-typed process expressions in a λ-calculus (Chapter 4). One interesting consequence of this embedding is that it allows for process expressions to send, receive and execute *other* process expressions (Section 4.4), in the sense of *higher-order processes* [68], preserving the property of deadlock-freedom by typing (Section 4.5), a key open problem before this work.

For practicality and expressiveness, we consider the addition of general recursive types to the language (Section 4.2), breaking the connection with logic in that it introduces potentially divergent computations, but enabling us to showcase a wider range of interesting programs. To recover the connections with logic, in Chapter 5 we restrict general recursive types to coinductive session types, informally arguing for non-divergence of computation through the introduction of syntactic restrictions on recursive process definitions (the formal argument is developed in Part III), similar to those used in dependently typed programming languages with recursion such as Coq [74] and Agda [55], but with additional subtleties due to the concurrent nature of the language.

1.6.3 Reasoning Techniques

Another fundamental strength of a logical foundation for concurrent computation lies in its capacity to provide both the ability to express and also *reason* about such computations. To this end, we develop in Part III of the dissertation a theory of *linear logical relations* on the π-calculus assignment for the interpretation (Chapter 6), developing both unary (Section 6.1) and binary (Section 6.2) relations for the polymorphic and coinductive settings, which consists of a restricted form of the language of Chapter 4, following the principles of Chapter 5. We show how these relations can be applied to develop interesting results such as termination of well-typed processes in the considered settings, and concepts such as a notion of logical equivalence.

In Chapter 7, we develop applications of the linear logical relations framework, showing that our notion of logical equivalence is sound and complete with respect to the traditional process calculus equivalence of (typed) *barbed congruence* and may be used to perform reasoning on polymorphic processes in the style of parametricity for polymorphic functional languages (Section 7.1). We also develop the concept of session type isomorphisms (Section 7.2), a form of type compatibility or equivalence. Finally, in Section 7.3, by appealing to our notion of logical equivalence, we show how to give a concurrent justification for the proof conversions that arise in the process of cut elimination in the proof theory of linear logic, thus giving a full account of the proof-theoretic transformations of cut elimination within our framework.

Part I

A Logical Foundation for Session-based Concurrency

CHAPTER 2

LINEAR LOGIC AND SESSION TYPES

In this chapter, we develop the connections of propositional linear logic and session types, developing a concurrent proof term assignment to linear logic proofs that corresponds to a session-typed π-calculus. We begin with a brief introduction of the formalism of linear logic and the judgmental methodology used throughout this dissertation. We also introduce the concept of *substructural operational semantics* [62] which we use to give a semantics to our proof term assignment and later (Part II) to our programming language.

The main goal of this chapter is to develop and motivate the basis of the logical foundation for session-typed concurrent computation. To this end, our development of a concurrent proof term assignment for intuitionistic linear logic does not directly use π-calculus terms (as the work of [16]) but rather a fully syntax driven term assignment, faithfully matching the dynamics of proofs, all the while being consistent with the original π-calculus assignment. The specifics and the reasoning behind this new assignment are detailed in Section 2.2. The term assignment is developed incrementally in Section 2.3, with a full summary in Section 2.5 for reference. Some examples are discussed in Section 2.3.5. Finally, we establish the fundamental metatheoretical properties of our development in Section 2.4. Specifically, we make precise the correspondence between our term assignment and the π-calculus assignment and establish type preservation and global progress results, entailing a notion of session fidelity and deadlock-freedom.

2.1 Linear Logic and the Judgmental Methodology

In Section 1.4 we gave an informal description of the key ideas of linear logic and its connections to session-based concurrency. Here, we make the formalism of linear logic precise so that we may then carry out our concurrent term assignment to the rules of intuitionistic linear logic.

As we have mentioned, linear logic arises as an effort to marry the dualities of classical logic and the constructive nature of intuitionistic logic. However, the key aspect of linear logic that interests us is the idea that logical propositions are *stateful*, insofar as they may be seen as mutable

$$A, B, C \quad ::= \quad \mathbf{1} \mid A \multimap B \mid A \otimes B \mid A \mathbin{\&} B \mid A \oplus B \mid !A$$

Figure 2.1: Intuitionistic Linear Logic Propositions

resources that evolve, interact and are consumed during logical inference. Thus, linear logic treats evidence as *ephemeral* resources – using a resource consumes it, making it unavailable for further use. While linear logic can be presented in both classical and intuitionistic form, we opt for the intuitionistic formalism since it has a more natural correspondence with our intuition of resources and resource usage. Moreover, as we will see later on (Part II), intuitionistic linear logic seems to be more amenable to integration in a λ-calculus based functional language (itself connected to intuitionistic logic).

Seeing as linear logic is a logic of resources and their usage disciplines, linear logic *propositions* can be seen as resource combinators that specify how certain kinds of resources are meant to be used. Propositions (referred to by A, B, C – Fig. 2.1) in linear logic can be divided into three categories, depending on the discipline they impose on resource usage: the *multiplicative* fragment, containing $\mathbf{1}$, $A \otimes B$ and $A \multimap B$; the *additive* fragment, containing $A \mathbin{\&} B$ and $A \oplus B$; and the *exponential* fragment, containing $!A$. The multiplicative unit $\mathbf{1}$ is the distinguished proposition which denotes no information – no resources are used to provide this resource. Linear implication $A \multimap B$ denotes a form of resource transformation, where a resource A is transformed into resource B. Multiplicative conjunction $A \otimes B$ encodes simultaneous availability of resources A and B. Additive conjunction $A \mathbin{\&} B$ denotes the alternative availability of A and B (i.e., each is available but not both simultaneously). Additive disjunction $A \oplus B$ denotes the availability either A or B (i.e., only one is made available). Finally, the exponential $!A$ encodes the availability of an arbitrary number of instances of the resource A.

Our presentation of linear logic consists of a *sequent calculus*, corresponding closely to Barber's dual intuitionistic linear logic [7], but also to Chang et al.'s judgmental analysis of intuitionistic linear logic [21]. Our main judgment is written $\Delta \vdash A$, where Δ is a multiset of (linear) resources that are fully consumed to offer resource A. We write $\cdot$ for the empty context. Following [21], the exponential $!A$ internalizes validity in the logic – proofs of A using no linear resources, requiring an additional context of unrestricted or exponential resources Γ and the generalization of the judgment to $\Gamma; \Delta \vdash A$, where the resources in Γ need not be used.

In sequent calculus, the meanings of propositions are defined by so-called right and left rules (referring to the position of the principal proposition relative to the turnstyle in the rule). A right rule defines how to prove a given proposition, or in resource terminology, how to *offer* a resource. Dually, left rules define how to *use* an assumption of a particular proposition, or how to use an ambient resource. Sequent calculus also includes so-called structural rules, which do not pertain to specific propositions but to the proof system as a whole, namely the cut rule, which defines how to reason about a proof using lemmas, or how to construct auxiliary resources which are then consumed; and the identity rule, which specifies the conditions under which a linear assumption may discharge a proof obligation, or how an ambient resource may be offered outright. Historically, sequent calculus proof systems include cut and identity rules (and also explicit exchange, weakening and contraction rules as needed), which can *a posteriori* shown to be redundant via cut elimination and identity expansion. Alternatively (as in [21]), cut and identity may be omitted from the proof system and then shown as admissible principles of proof. We opt for the former since we wish to be able to express composition as primitive in the framework and reason explicitly about the computational content of proofs as defined by their dynamics during the cut elimination procedure.

To fully account for the exponential in our judgmental style of presentation, beyond the additional context region Γ, we also require two additional rules: an exponential cut principle, which specifies how to compose proofs that manipulate persistent resources; and a *copy* rule, which specifies how to use a persistent resource.

The rules of linear logic are presented in full detail in Section 2.3, where we develop the session-based concurrent term assignment to linear logic proofs and its correspondence to π-calculus processes. The major guiding principle for this assignment is the computational reading of the cut elimination procedure, which simplifies the composition of two proofs to a composition of smaller proofs. At the level of the proof term assignment, this simplification or reduction procedure serves as the main guiding principle for the operational semantics of terms (and of processes).

As a way of crystallizing the resource interpretation of linear logic and its potential for a session-based concurrent interpretation, consider the following valid linear judgment $A \multimap B, B \multimap C, A \vdash C$. According to the informal description above, the linear implication $A \multimap B$ can be seen as a form of resource transformation: provide it with a resource A to produce a resource B. Given this reading, it is relatively easy to see why the judgment is valid: we wish to provide the resource C by making full use of resources $A \multimap B$, $B \multimap C$ and A. To do this, we use the resource A in combination with the linear implication $A \multimap B$, consuming the A and producing B, which we then provide to the implication $B \multimap C$ to finally obtain C. While this resource manipulation reading of inference seems reasonable, where does the session-based concurrency come from? The key intuition consists of viewing a proposition as a denotation of the state of an *interactive behavior* (a session), and that such a behavior is resource-like, where we can view the interplay of the various interactive behaviors as *concurrent interactions* specified by the logical meaning of the propositional connectives. Thus, appealing to this session-based reading of linear logic propositions, $A \multimap B$ stands for the type of an object that implements the interaction of receiving a behavior A to then produce the behavior B.

From a session-based concurrent point of view, we may interpret the judgment $A \multimap B, B \multimap C, A \vdash C$ as follows: in an environment composed of three independent interactive behaviors offered by, say, P, Q and R, where P provides a behavior of type $A \multimap B$, Q one of type $B \multimap C$ and R one of type A, we can make use of these interactive behaviors to provide a behavior C. As will see later on, the realizer for this behavior C must therefore consist of plugging together R and P by sending P some information, after which S will be able to connect P with Q to obtain C.

Foreshadowing our future developments, we can extract from the inference of $A \multimap B, B \multimap C, A \vdash C$ the *code* for the concurrent process S as follows – we assign types to channel names along which communication takes place and take some syntactic liberties for readability at this stage in the presentation:

$$x{:}A \multimap B, y{:}B \multimap C, z{:}A \vdash S :: w{:}C \quad \text{where} \quad S = \mathsf{output}\ x\ z; \mathsf{output}\ y\ x; \mathsf{output}\ w\ y$$

The code for S captures the intuition behind the expected behavior of the process that transforms the available behaviors into an offer of C. It begins by linking z with x, sending the former along x. We are now in a setting where x is offering B and thus we send it to y which ultimately results in the C we were seeking. Moreover, provided with the appropriate processes P, Q and R (extractable from linear logic proofs), we can account for the closed concurrent system made up of the four processes executing concurrently (again, modulo some syntactic conveniences):

$$\vdash \mathsf{new}\ x, y, z.(P \parallel Q \parallel R \parallel S) :: w{:}C \quad \text{where}$$

$$\vdash P :: x{:}A \multimap B \qquad \vdash Q :: y{:}B \multimap C \qquad \vdash R :: z{:}A$$

As will be made clear throughout our presentation, this rather simple and arguably intuitive reading of linear logic turns out to provide an incredibly rich and flexible framework for defining and reasoning about session-based, message-passing programs.

2.2 Preliminaries

In this section we present some introductory remarks that are crucial for the full development of the logical interpretation of Section 2.3.

We point out the fundamental difference in approach to the foundational work of [16] on the connection of linear logic and session types. We opt for a different formulation of this logical interpretation for both stylistic purposes and to address a few shortcomings of the original interpretation. First, we use linear forwarders as an explicit proof term for the identity rule, which is not addressed in [16] (such a construct was first proposed for the interpretation in [75]). Secondly, we use a different assignment for the exponentials of linear logic (in line with [25, 77]), which matches proof reductions with process reductions in a faithful way. The final and perhaps most important distinction is that the rules we present are all syntax driven, not using the π-calculus proof term assignment outright (which requires typing up-to structural congruence). This proof term assignment forms a basis for a concurrent, session-typed language that can be mapped to session-typed π-calculus processes in a way that is consistent with the interpretation of [16], but that does not require structural congruence for typing nor for the operational semantics, by using a technique called *substructural operational semantics* [62] (SSOS in the sequel). We refer to these proof terms as *process expressions*, in opposition to π-calculus terms which I refer to simply as *processes*. Thus, our assignment does not require the explicit representation of the π-calculus ν-binder in the syntax, nor the explicit commutativity of parallel composition in the operational semantics, which are artefacts that permeate the foundational work of [16] and so require the extensive use of structural congruence of process terms for technical reasons. However, there are situations where using π-calculus terms and structural congruence turn out to be technically convenient, and abandoning the π-calculus outright would diminish the claims of providing a true logical foundation of session-based concurrent computation.

In our presentation we develop the proof term assignment *in tandem* with the π-calculus term assignment but without placing added emphasis on one or the other, enabling reasoning with techniques from proof theory and process calculi (and as we will see, combinations of both) and further emphasizing the back-and-forth from language to logic that pervades the works exploring logical correspondences in the sense of Curry-Howard. The presentation also explores the idea of identifying a process by the channel along which it offers its session behavior. This concept is quite important from a language design perspective (since it defines precise boundaries on the notion of a process), but is harder to justify in the π-calculus assignment. The identification of processes by a single channel further justifies the use of intuitionistic linear logic over classical linear logic, for which a correspondence with session types may also be developed (viz. [18, 81]), but where the classical nature of the system makes such an identification rather non-obvious.

2.2.1 Substructural Operational Semantics

As mentioned above, we define the operational semantics of process expressions in the form of a *substructural operational semantics* (SSOS) [62]. For the reader unfamiliar with this style of presentation, it consists of a compositional specification of the operational semantics by defining a predicate on the expressions of the language, through rules akin to those of multiset rewriting [20], where the pattern to the left of the $\multimap$ arrow (not to be confused with our object language usage of $\multimap$ as a type) describes a state which is consumed and transformed into the one to the right. The pattern on the right is typically made up of several predicates, which are grouped using a $\otimes$. The SSOS framework allows us to generate fresh names through existential quantification and crucially does not require an explicit formulation of structural congruence, as one would expect when defining the typical operational semantics for process calculi-like languages. This is due to the fact that the metatheory of SSOS is based on ordered logic programming where exchange is implicit for the linear fragment of the theory. In the remainder of this document we only make use of the linear fragment of SSOS.

To exemplify, we develop a simple SSOS specification of a destination-passing semantics for a linear λ-calculus using three linear predicates: $\mathsf{eval}\ M\ d$ which denotes a λ-term M that is to be evaluated on destination d; $\mathsf{return}\ V\ d$, denoting the return of value V to destination d; and $\mathsf{cont}\ x\ F\ w$, which waits on a value from destination x to carry out F and pass the result to w. Fresh destinations can be generated through existential quantification.

To evaluate function application $M\,N$ we immediately start the evaluation of M to a fresh destination x and generate a continuation which waits for the evaluation of M before proceeding:

$$\mathsf{eval}\ (M\,N)\ d \multimap \{\exists x.\mathsf{eval}\ M\ x \otimes \mathsf{cont}\ x\ (\_\,N)\ d\}$$

Since λ-expressions do not trigger evaluation, they are returned as values:

$$\mathsf{eval}\ (\lambda y.M)\ d \multimap \{\mathsf{return}\ (\lambda y.M)\ d\}$$

When a returned function meets its continuation, we may evaluate its body by instantiating the destination of the argument accordingly:

$$\mathsf{return}\ (\lambda y.M)\ d \otimes \mathsf{cont}\ x\ (\_N)\ w \multimap \{\exists z.\mathsf{eval}\ (M\{z/y\})\ w \otimes \mathsf{eval}\ N\ z\}$$

When we encounter a variable we have to wait for its value, forwarding it to the target destination:

$$\mathsf{eval}\ x\ w \multimap \{\mathsf{cont}\ x\ \_\ w\}$$

Finally, when a continuation waits for the return of a value, we return it on the destination of the continuation:

$$\mathsf{return}\ v\ x \otimes \mathsf{cont}\ x\ \_\ w \multimap \{\mathsf{return}\ v\ w\}$$

It is straightforward to see how these five rules combined define a rewriting system for evaluating linear λ-calculus terms using destinations (more examples can be found in [62], making full use of the ordered nature of the framework). We note that the operational semantics specified above are identical to futures, in the sense that evaluation of a function body and its argument can proceed in parallel.

To develop the SSOS for our language of process expressions we rely on the following two predicates (these will be extended further as we proceed in the presentation): the linear proposition exec P denotes the state of a linear process expression P and !exec P denotes the state of a persistent process (which must always be a replicating input).

We use $\Omega \longrightarrow \Omega'$ to refer to a single multiset rewriting, according to the specified SSOS rules, from execution state Ω to execution state Ω'. As usual, we use $\longrightarrow^{+}$ and $\longrightarrow^{*}$ for the transitive and reflexive transitive closures of $\longrightarrow$. We write $\longrightarrow^{n}$ to denote a sequence of n rewrite steps and $\longrightarrow^{n,m}$ to denote a sequence of n or m rewrite steps.

2.3 Propositional Linear Logic

We now develop our concurrent interpretation of intuitionistic linear logic, going through each sequent calculus rule and giving it both a π-calculus and a corresponding process expression assignment. For the rules of linear logic and our process expression assignment we employ the judgment $\Gamma;\Delta \vdash A$ and $\Gamma;\Delta \vdash P :: z{:}A$, respectively, whereas for the π-calculus assignment we use the judgment $\Gamma;\Delta \Rightarrow P :: z{:}A$. We note that typing for π-calculus processes is defined modulo structural congruence. For both assignments, we consistently label unrestricted and linear assumptions with channel names (u,v,w for the former and x,y,z for the latter) and we single out the distinguished right-hand side channel z, the channel along which process expression (or π-calculus process) P offers the behavior A when composed with processes offering the behaviors specified in Γ and Δ on the appropriate channels. We tacitly assume that all channel names in Γ, Δ and z are distinct.

The operational semantics of both assignments are driven by the principal cut reduction (when corresponding left and right rules of a connective meet in a cut) in linear logic. For the π-calculus assignment they match the expect reduction semantics modulo structural congruence, whereas for the process expression assignment, as discussed above, we introduce the semantics in SSOS-style, avoiding the use of structural congruence and explicit ν-binders.

2.3.1 Judgmental Rules

We begin our development with the basic judgmental principles of linear logic, namely the cut and identity principles (additional judgmental principles are required to justify the exponential !, but we leave those for the interpretation of the exponential). In linear logic a cut consists of a principle of proof composition, where some resources are used to prove a proposition A (also known as the cut formula) and the remainder are used to prove some proposition C, under the assumption that a proof of A exists:

$$\frac{\Gamma;\Delta_1 \vdash A \quad \Gamma;\Delta_2, A \vdash C}{\Gamma;\Delta_1,\Delta_2 \vdash C}\ (\textsc{cut})$$

At the level of process expressions and π-calculus processes, this is interpreted as a form of composition (we write P_x to denote that x occurs in P, writing $P_{x'}$ to denote the consistent renaming of x to x' in P):

$$\frac{\Gamma;\Delta_1 \vdash P_x :: x{:}A \quad \Gamma;\Delta_2, x{:}A \vdash Q_x :: z{:}C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathtt{new}\ x.(P_x \| Q_x) :: z{:}C}\ (\textsc{cut})$$

Note how x is bound in Q, since it is the only term which can actually make use of the session behavior A offered by P. Operationally, we make use of the following SSOS rule to capture the parallel composition nature of a cut:

$$(\textsc{cut})\ \mathsf{exec}(\mathtt{new}\ x.(P_x \| Q_x)) \multimap \{\exists x'.\mathsf{exec}(P_{x'}) \otimes \mathsf{exec}(Q_{x'})\}$$

The rule transforms the process expression assigned to a cut into the parallel execution of the two underlying process expressions, sharing a fresh name x' generated by the existential quantifier. In the π-calculus this corresponds to parallel composition plus name restriction:

$$\frac{\Gamma;\Delta_1 \vdash P :: x{:}A \quad \Gamma;\Delta_2, x{:}A \vdash Q :: z{:}C}{\Gamma;\Delta_1,\Delta_2 \vdash (\nu x)(P \mid Q) :: z{:}C}\ (\textsc{cut})$$

The $\boldsymbol{\nu}$-binder for x is necessary to ensure that the channel x is shared only between P and Q. To determine the correct operational behavior for our constructs we use as a guiding principle the computational content of the proof of cut elimination in linear logic. In particular, we consider the cut reduction steps when a right rule is cut with a left rule of a particular connective (also known as a principal cut).

The identity principle of linear logic states that we may use an assumption A to prove A, provided it is our only remaining (linear) resource in order to ensure linearity:

$$\frac{}{\Gamma; A \vdash A}\ (\textsc{id})$$

To develop the process expression assignment we must consider the rule above at the level sessions: we have access to an ambient session x that offers A and we wish to use it to offer A outright along channel z. Thus, the natural interpretation is to forward communication between the two channels z and x:

$$\dfrac{}{\Gamma; x{:}A \vdash \texttt{fwd}\ z\ x :: z{:}A}\ (\textsc{id})$$

or in the π-calculus syntax of Fig. 1.1,

$$\dfrac{}{\Gamma; x{:}A \Rightarrow [x \leftrightarrow z] :: z{:}A}\ (\textsc{id})$$

If we consider what happens during cut elimination we can see that a cut against identity simply erases the identity (but in a setting where we have labels attached to terms and assumptions, this requires renaming):

$$\dfrac{\Gamma; \Delta_1, \Delta_2 \vdash P_x :: x{:}A \quad \dfrac{}{\Gamma; x{:}A \vdash \mathsf{fwd}\ z\ x :: z{:}A}\ (\textsc{id})}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{new}\ x.(P_x \parallel \mathsf{fwd}\ z\ x) :: z{:}A}\ (\textsc{cut})$$

$$\Longrightarrow \Gamma; \Delta_1, \Delta_2 \vdash P\{z/x\} :: z{:}A$$

Another cut reduction arises when the identity is in the left premise of the cut:

$$\dfrac{\dfrac{}{\Gamma; y{:}A \vdash \mathsf{fwd}\ x\ y :: x{:}A}\ (\textsc{id}) \qquad \Gamma; \Delta_2, x{:}A \vdash P_x :: z{:}C}{\Gamma; y{:}A, \Delta_2 \vdash \texttt{new}\ x.(\mathsf{fwd}\ x\ y \parallel P_x) :: z{:}A}\ (\textsc{cut})$$

$$\Longrightarrow \Gamma; y{:}A, \Delta_2 \vdash P\{y/x\} :: z{:}C$$

Thus the SSOS rule for forwarding simply equates the two channel names:

$$(\textsc{fwd})\ \mathsf{exec}\ (\mathsf{fwd}\ x\ z) \multimap \{x = z\}$$

The corresponding π-calculus reduction is:

$$(\nu x)(P \mid [x \leftrightarrow z]) \longrightarrow P\{z/x\}$$

2.3.2 Multiplicatives

Multiplicative conjunction $A \otimes B$ denotes the simultaneous availability of both A and B, where both A and B are intended to be used. Thus, to offer $A \otimes B$ we must split our linear resources in two parts, one used to provide A and the other to provide B:

$$\dfrac{\Gamma; \Delta_1 \vdash A \quad \Gamma; \Delta_2 \vdash B}{\Gamma; \Delta_1, \Delta_2 \vdash A \otimes B}\ (\otimes\mathsf{R})$$

As highlighted in Fig. 1.3, the session type $A \otimes B$ denotes an output of a (fresh) session channel that offers A and a change of state to offer B along the original channel. The process expression assignment is:

$$\frac{\Gamma; \Delta_1 \vdash P_y :: y{:}A \quad \Gamma; \Delta_2 \vdash Q :: z{:}B}{\Gamma; \Delta_1, \Delta_2 \vdash \mathtt{output}\ z\ (y.P_y); Q :: z{:}A \otimes B}\ (\otimes\mathsf{R})$$

We thus have a process expression which can be identified by the channel z, along which an output of a fresh session channel is offered. The intended session behavior A for this fresh channel is implemented by the process expression P (note that y occurs free in P but not in the output process expression nor in Q), whereas the continuation Q offers $z{:}B$, independently. Using $A \otimes B$ warrants the use of both A and B, as captured in the following left rule:

$$\frac{\Gamma; \Delta, A, B \vdash C}{\Gamma; \Delta, A \otimes B \vdash C}\ (\otimes\mathsf{L})$$

At the process expression level, using a session that offers an output naturally consists of performing an input:

$$\frac{\Gamma; \Delta, y{:}A, x{:}B \vdash R_y :: z{:}C}{\Gamma; \Delta, x{:}A \otimes B \vdash y \leftarrow \mathtt{input}\ x; R_y :: z{:}C}\ (\otimes\mathsf{L})$$

The process expression consists of an input along the ambient session channel x, which is of type $A \otimes B$ and therefore outputs a fresh channel that offers A, which is bound to y in the continuation R. Moreover, the state of x changes after the input is performed, now offering B. The π-calculus process assignment consists of a bound output and an input for the right and left rules, respectively:

$$\frac{\Gamma; \Delta_1 \Rightarrow P :: y{:}A \quad \Gamma; \Delta_2 \Rightarrow Q :: z{:}B}{\Gamma; \Delta_1, \Delta_2 \Rightarrow (\nu y) z\langle y \rangle.(P \mid Q) :: z{:}A \otimes B}\ (\otimes\mathsf{R})$$

$$\frac{\Gamma; \Delta, y{:}A, x{:}B \Rightarrow R :: z{:}C}{\Gamma; \Delta, x{:}A \otimes B \Rightarrow x(y).R :: z{:}C}\ (\otimes\mathsf{L})$$

To determine the correct operational behavior of our process expression assignment, we examine the principal cut reduction for $\otimes$, which justifies the intuitive semantics of input and output:

$$\frac{\dfrac{\Gamma; \Delta_1 \vdash P :: y{:}A \quad \Gamma; \Delta_2 \vdash Q :: x{:}B}{\Gamma; \Delta_1, \Delta_2 \vdash \mathtt{output}\ x\ (y.P_y); Q :: x{:}A \otimes B}\ (\otimes\mathsf{R}) \quad \dfrac{\Gamma; \Delta_3, y{:}A, x{:}B \vdash R :: z{:}C}{\Gamma; \Delta_3, x{:}A \otimes B \vdash y \leftarrow \mathtt{input}\ x; R :: z{:}C}\ (\otimes\mathsf{L})}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \mathtt{new}\ x.((\mathtt{output}\ x\ (y.P_y); Q) \parallel y \leftarrow \mathtt{input}\ x; R) :: z{:}C}\ (\textsc{cut})$$

$$\Longrightarrow \quad \frac{\Gamma; \Delta_1 \vdash Q :: x{:}B \qquad \dfrac{\Gamma; \Delta_2 \vdash P :: y{:}A \quad \Gamma; \Delta_3, y{:}A, x{:}B \vdash R :: z{:}C}{\Gamma; \Delta_2, \Delta_3, x{:}B \vdash \mathtt{new}\ y.(P \parallel R) :: z{:}C}\ (\textsc{cut})}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \mathtt{new}\ x.(Q \parallel \mathtt{new}\ y.(P \parallel R)) :: z{:}C}\ (\textsc{cut})$$

The output $(\otimes\mathsf{R})$ synchronizes with the input $(\otimes\mathsf{L})$, after which we have a composition of three process expressions: P offering session A along the fresh channel, the continuation Q offering session B and the continuation R that uses both to provide C, resulting in the following SSOS rule:

$$(\textsc{scom})\ \mathsf{exec}\ (\mathsf{output}\, x\ (y.P_y); Q) \otimes \mathsf{exec}\ (y \leftarrow \mathsf{input}\, x; R_y) \multimap \{\exists y'.\, \mathsf{exec}\ (P_{y'}) \otimes \mathsf{exec}\ (Q) \otimes \mathsf{exec}\ (R_{y'})\}$$

The rule above specifies that whenever an input and an output action of the form above are available on the same channel, the communication fires and, after generating a fresh channel y', we transition to a state where the three appropriate continuations execute concurrently (with P and R sharing the fresh channel y' for communication).

At the level of π-calculus processes, this results in the familiar synchronization rule of reduction semantics:

$$(\nu y)x\langle y\rangle.P \mid x(z).Q \longrightarrow (\nu y)(P \mid R\{y/z\})$$

For the reader familiar with linear logic, it might seem somewhat odd that $\otimes$ which is a commutative operator is given a seemingly non-commutative interpretation. In the proof theory of linear logic, the commutative nature of $\otimes$ is made precise via a *type isomorphism* between $A \otimes B$ and $B \otimes A$. In our assignment a similar argument can be developed by appealing to observational equivalence. A precise notion of observational equivalence (and session type isomorphism) is deferred to Part III, where the necessary technical tools are developed. However, we highlight that it is the case that given a session of type $A \otimes B$ we can produce a realizer that offers $B \otimes A$ and vice-versa, such that the composition of the two is observationally equivalent to the identity.

The remaining multiplicative is linear implication $A \multimap B$, which can be seen as a resource transformer: given an A it will consume it to produce a B. To offer $A \multimap B$ we simply need to be able to offer B by using A. Dually, to use $A \multimap B$ we must provide a proof of A, which justifies the use of B.

$$\frac{\Gamma;\Delta, A \vdash B}{\Gamma;\Delta \vdash A \multimap B}\ (\multimap\mathsf{R}) \qquad \frac{\Gamma;\Delta_1 \vdash A \quad \Gamma;\Delta_2, B \vdash C}{\Gamma;\Delta_1,\Delta_2, A \multimap B \vdash C}\ (\multimap\mathsf{L})$$

The proof term assignment for linear implication is dual to that of multiplicative conjunction. To offer a session of type $x{:}A \multimap B$ we need to input a session channel of type A along x, which then enables the continuation to offer a session of type $x{:}B$. Using such a session requires the dual action, consisting of an output of a fresh session channel that offers A, which then warrants using x as a session of type B.

$$\frac{\Gamma;\Delta, x{:}A \vdash P_x :: z{:}B}{\Gamma;\Delta \vdash x \leftarrow \texttt{input}\ z; P_x :: z{:}A \multimap B}\ (\multimap\mathsf{R})$$

$$\frac{\Gamma;\Delta_1 \vdash Q :: y{:}A \quad \Gamma;\Delta_2, x{:}B \vdash R :: z{:}C}{\Gamma;\Delta_1,\Delta_2, x{:}A \multimap B \vdash \texttt{output}\ x\ (y.Q_y); R :: z{:}C}\ (\multimap\mathsf{L})$$

The π-calculus process assignment follows the expected lines:

$$\frac{\Gamma;\Delta, x{:}A \Rightarrow P :: z{:}B}{\Gamma;\Delta \Rightarrow z(x).P :: z{:}A \multimap B}\ (\multimap\mathsf{R})$$

$$\frac{\Gamma;\Delta_1 \Rightarrow Q :: y{:}A \quad \Gamma;\Delta_2, x{:}B \Rightarrow R :: z{:}C}{\Gamma;\Delta_1,\Delta_2, x{:}A \multimap B \Rightarrow (\nu y)x\langle y\rangle.(Q \mid R) :: z{:}C}\ (\multimap\mathsf{L})$$

The proof reduction obtained via cut elimination matches that of $\otimes$, and results in the same operational semantics.

Finally, we consider the multiplicative unit of linear logic, written **1**. Offering **1** requires no linear resources and so its corresponding right rule may only be applied with an empty linear context. Using **1** also adds no extra linear resources, and so we obtain the following rules:

$$\frac{}{\Gamma; \cdot \vdash \mathbf{1}}\ (\mathbf{1}\mathsf{R}) \qquad \frac{\Gamma; \Delta, \mathbf{1} \vdash C}{\Gamma; \Delta \vdash C}\ (\mathbf{1}\mathsf{L})$$

Thus, a session channel of type **1** denotes a session channel that is to be terminated, after which no subsequent communication is allowed. We explicitly signal this termination by closing the communication channel through a specialized process expression written `close`. Dually, since our language is synchronous, we must wait for ambient sessions of type **1** to close:

$$\frac{}{\Gamma; \cdot \vdash \mathtt{close}\ z :: z{:}\mathbf{1}}\ (\mathbf{1}\mathsf{R}) \qquad \frac{\Gamma; \Delta \vdash Q :: z{:}C}{\Gamma; \Delta, x{:}\mathbf{1} \vdash \mathtt{wait}\ x; Q :: z{:}C}\ (\mathbf{1}\mathsf{L})$$

Since no such primitives exist in the π-calculus, we can simply use a form of nullary communication for the multiplicative unit assignment:

$$\frac{}{\Gamma; \cdot \Rightarrow z\langle\rangle.\mathbf{0} :: z{:}\mathbf{1}}\ (\mathbf{1}\mathsf{R}) \qquad \frac{\Gamma; \Delta \Rightarrow Q :: z{:}C}{\Gamma; \Delta, x{:}\mathbf{1} \Rightarrow x().Q :: z{:}C}\ (\mathbf{1}\mathsf{L})$$

As before, our guiding principle of appealing to principal cut reductions validates the informally expected behavior of the process expression constructs specified above (as well as for the synchronization step on π-calculus processes):

$$\frac{\dfrac{}{\Gamma; \cdot \vdash \mathtt{close}\ x :: x{:}\mathbf{1}}\ (\mathbf{1}\mathsf{R}) \qquad \dfrac{\Gamma; \Delta \vdash Q :: z{:}C}{\Gamma; \Delta, x{:}\mathbf{1} \vdash \mathtt{wait}\ x; Q :: z{:}C}\ (\mathbf{1}\mathsf{L})}{\Gamma; \Delta \vdash \mathtt{new}\ x.(\mathtt{close}\ x \parallel (\mathtt{wait}\ x; Q)) :: z{:}C}\ (\textsc{cut}) \quad \Longrightarrow \Gamma; \Delta \vdash Q :: z{:}C$$

The SSOS rule that captures this behavior is:

$$(\textsc{close})\ \mathsf{exec}\ (\mathsf{close}\, x) \otimes \mathsf{exec}\ (\mathsf{wait}\, x; P) \multimap \{\mathsf{exec}\ (P)\}$$

Essentially, whenever a session closure operation meets a process waiting for the same session channel to close, the appropriate continuation is allowed to execute. The associated reduction rule on π-calculus terms is just the usual communication rule.

2.3.3 Additives

Additive conjunction in linear logic $A \mathbin{\&} B$ denotes the ability to offer either A or B. That is, the available resources must be able to provide A and must also be able to provide B, but not both simultaneously (as opposed to $\otimes$ which requires both to be available). Therefore, we offer $A \mathbin{\&} B$ by being able to offer A and B, without splitting the context:

$$\frac{\Gamma; \Delta \vdash A \quad \Gamma; \Delta \vdash B}{\Gamma; \Delta \vdash A \mathbin{\&} B}\ (\&\mathsf{R})$$

To use $A \mathbin{\&} B$, unlike $A \otimes B$ where we are warranted in using *both* A and B, we must chose which of the two resources will be used:

$$\frac{\Gamma; \Delta, A \vdash C}{\Gamma; \Delta, A \mathbin{\&} B \vdash C}\ (\&\mathsf{L}_1) \qquad \frac{\Gamma; \Delta, B \vdash C}{\Gamma; \Delta, A \mathbin{\&} B \vdash C}\ (\&\mathsf{L}_2)$$

Thus, $A \mathbin{\&} B$ denotes a form of alternative session behavior where a choice between the two behaviors A and B is offered.

$$\frac{\Gamma; \Delta \vdash P :: z{:}A \quad \Gamma; \Delta \vdash Q :: z{:}B}{\Gamma; \Delta \vdash z.\mathsf{case}(P, Q) :: z{:}A \mathbin{\&} B}\ (\&\mathsf{R})$$

The process expression above waits on session channel z for a choice between the left or right branches, which respectively offer $z{:}A$ and $z{:}B$. Using such a process expression is achieved by performing the appropriate selections:

$$\frac{\Gamma; \Delta, x{:}A \vdash R :: z{:}C}{\Gamma; \Delta, x{:}A \mathbin{\&} B \vdash x.\mathtt{inl}; R :: z{:}C}\ (\&\mathsf{L}_1) \qquad \frac{\Gamma; \Delta, x{:}B \vdash R :: z{:}C}{\Gamma; \Delta, x{:}A \mathbin{\&} B \vdash x.\mathtt{inr}; R :: z{:}C}\ (\&\mathsf{L}_2)$$

The π-calculus process assignment coincides precisely with the process expression assignment above.

The two principal cut reductions (one for each right-left rule pair) capture the expected behavior of choice and selection:

$$\frac{\dfrac{\Gamma; \Delta_1 \vdash P :: x{:}A \quad \Gamma; \Delta_1 \vdash Q :: x{:}B}{\Gamma; \Delta_1 \vdash x.\mathsf{case}(P, Q) :: x{:}A \mathbin{\&} B}\ (\&\mathsf{R}) \quad \dfrac{\Gamma; \Delta_2, x{:}A \vdash R :: z{:}C}{\Gamma; \Delta_2, x{:}A \mathbin{\&} B \vdash x.\mathtt{inl}; R :: z{:}C}\ (\&\mathsf{L}_1)}{\Gamma; \Delta_1, \Delta_2 \vdash \mathtt{new}\ x.(x.\mathsf{case}(P, Q) \parallel (x.\mathtt{inl}; R)) :: z{:}C}\ (\textsc{cut})$$

$$\Longrightarrow \quad \frac{\Gamma; \Delta_1 \vdash P :: x{:}A \quad \Gamma; \Delta_2, x{:}A \vdash R :: z{:}C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathtt{new}\ x.(P \parallel R) :: z{:}C}\ (\textsc{cut})$$

Resulting in the following SSOS rule (omitting the obvious rule for the right-branch selection):

$$(\textsc{choice})\ \mathsf{exec}\ (x.\mathtt{inl}; P) \otimes \mathsf{exec}\ (x.\mathsf{case}(Q, R)) \multimap \{\mathsf{exec}\ (P) \otimes \mathsf{exec}\ (Q)\}$$

The reduction rules induced on π-calculus terms are those expected for guarded binary choice:

$$\begin{array}{c} x.\mathsf{case}(P, Q) \mid x.\mathtt{inl}; R \longrightarrow P \mid R \\ x.\mathsf{case}(P, Q) \mid x.\mathtt{inr}; R \longrightarrow Q \mid R \end{array}$$

Additive disjunction $A \oplus B$ denotes an internal choice between A or B, in that offering $A \oplus B$ only requires being able to offer either A or B, but not necessarily both unlike additive conjunction. In this sense, additive disjunction is dual to additive conjunction: the "choice" is made when offering $A \oplus B$, and using $A \oplus B$ requires being prepared for both possible outcomes:

$$\frac{\Gamma; \Delta \vdash A}{\Gamma; \Delta \vdash A \oplus B}\ (\oplus\mathsf{R}_1) \qquad \frac{\Gamma; \Delta \vdash B}{\Gamma; \Delta \vdash A \oplus B}\ (\oplus\mathsf{R}_2) \qquad \frac{\Gamma; \Delta, A \vdash C \quad \Gamma; \Delta, B \vdash C}{\Gamma; \Delta, A \oplus B \vdash C}\ (\oplus\mathsf{L})$$

The duality between ⊕ and & is also made explicit by the process expression assignment (and the π-calculus process assignment):

$$\frac{\Gamma;\Delta \vdash P :: z{:}A}{\Gamma;\Delta \vdash z.\mathtt{inl}; P :: z{:}A \oplus B}\ (\oplus\mathsf{R}_1) \qquad \frac{\Gamma;\Delta \vdash P :: z{:}B}{\Gamma;\Delta \vdash z.\mathtt{inr}; P :: z{:}A \oplus B}\ (\oplus\mathsf{R}_2)$$

$$\frac{\Gamma;\Delta, x{:}A \vdash Q :: z{:}C \quad \Gamma;\Delta, x{:}B \vdash R :: z{:}C}{\Gamma;\Delta, x{:}A \oplus B \vdash x.\mathtt{case}(Q,R) :: z{:}C}\ (\oplus\mathsf{L})$$

The proof reductions obtained in cut elimination are identical to the cases for additive conjunction and so are omitted for brevity.

2.3.4 Exponential

To allow for controlled forms of weakening and contraction in linear logic Girard introduced the exponential $!A$, denoting a proposition A which need not satisfy linearity and can be weakened and contracted in a proof, thus being able to be used in an unrestricted way. In the formulation of linear logic presented here, using distinct contexts for unrestricted and linear propositions, we require additional judgmental principles. To ensure cut elimination, we need an additional cut principle that given an unrestricted proposition, places it in the unrestricted context (we write $\cdot$ for the empty context). Moreover, we need a copy rule that enables the use of unrestricted propositions:

$$\frac{\Gamma;\cdot \vdash A \quad \Gamma, A;\Delta \vdash C}{\Gamma;\Delta \vdash C}\ (\mathsf{cut}^!) \qquad \frac{\Gamma, A;\Delta, A \vdash C}{\Gamma, A;\Delta \vdash C}\ (\mathsf{copy})$$

At the level of session types, persistent or unrestricted sessions consist of replicated services that may be used an arbitrary number of times. The process expression assignment to the unrestricted version of cut consists of a replicated input that guards the process expression P implementing the persistent session behavior (which therefore cannot use any linear sessions), composed with the process expression Q that can use the replicated session. Such uses are accomplished by triggering a copy the replicated session, which binds a fresh session channel x that will be used for communication with the replicated instance.

$$\frac{\Gamma;\cdot \vdash P_x :: x{:}A \quad \Gamma, u{:}A;\Delta \vdash Q_{!u} :: z{:}C}{\Gamma;\Delta \vdash \mathtt{new}\ !u.(x \leftarrow \mathtt{input}\ !u\ P_x \parallel Q_{!u}) :: z{:}C}\ (\mathsf{cut}^!)$$

$$\frac{\Gamma, u{:}A;\Delta, x{:}A \vdash R_x :: z{:}C}{\Gamma, u{:}A;\Delta \vdash x \leftarrow \mathtt{copy}\ !u; R_x :: z{:}C}\ (\mathsf{copy})$$

For the π-calculus assignment, the unrestricted cut essentially consists of a combination of replicated input and parallel composition, whereas the copy rule consists of a (bound) output:

$$\frac{\Gamma;\cdot \Rightarrow P :: x{:}A \quad \Gamma, u{:}A;\Delta \Rightarrow Q :: z{:}C}{\Gamma;\Delta \Rightarrow (\nu u)(!u(x).P \mid Q) :: z{:}C}\ (\mathsf{cut}^!) \qquad \frac{\Gamma, u{:}A;\Delta, x{:}A \Rightarrow R :: z{:}C}{\Gamma, u{:}A;\Delta \Rightarrow (\nu x)u\langle x\rangle.R :: z{:}C}\ (\mathsf{copy})$$

During cut elimination, when a persistent cut meets an instance of the copy rule, we generate a (linear) cut. We show this below, using process expressions:

$$\dfrac{\Gamma;\cdot\vdash P_x :: x{:}A \qquad \dfrac{\Gamma,u{:}A;\Delta,x{:}A\vdash Q_x :: z{:}C}{\Gamma,A;\Delta\vdash x\leftarrow \texttt{copy}\ !u;Q_x :: z{:}C}\,(\mathsf{copy})}{\Gamma;\Delta\vdash \texttt{new}\ !u.(x\leftarrow \texttt{input}\ !u\ P_x \parallel (x\leftarrow\texttt{copy}\ !u;Q_x)) :: z{:}C}\,(\mathsf{cut}^!)$$

$$\Longrightarrow\quad \dfrac{\Gamma;\cdot\vdash P_x :: x{:}A \qquad \dfrac{\Gamma,A;\cdot\vdash P_x :: x{:}A \quad \Gamma,u{:}A;\Delta,x{:}A\vdash Q_x :: z{:}C}{\Gamma,A;\Delta\vdash \texttt{new}\ x.(P_x \parallel Q_x) :: z{:}C}\,(\mathsf{cut})}{\Gamma;\Delta\vdash \texttt{new}\ !u.(x\leftarrow \texttt{input}\ !u\ P_x \parallel (\texttt{new}\ x.(P;Q))) :: z{:}C}\,(\mathsf{cut}^!)$$

The reduction above formally captures the intuitive meaning of spawning a replica of the persistent session offered by P, which is then composed accordingly with the continuation Q.

$$\begin{array}{ll}(\textsc{ucut}) & \mathsf{exec}\,(\texttt{new}\ !u.(x\leftarrow\texttt{input}\ !u\ P_x;Q)) \multimap \{\exists !u.!\mathsf{exec}(x\leftarrow\texttt{input}\ !u\ P_x)\otimes\mathsf{exec}(Q_{!u})\}\\ (\textsc{copy}) & !\mathsf{exec}\,(x\leftarrow\mathsf{input}\,!u;P_x)\otimes\mathsf{exec}\,(x\leftarrow\mathsf{copy}\,!u;Q_x)\multimap\{\exists c'.\,\mathsf{exec}(P_{c'})\otimes\mathsf{exec}(Q_{c'})\}\end{array}$$

The rule (UCUT) takes a persistent cut and sets up the replicated input required to execute the persistent session implemented by P. Note the usage of !exec to indicate that the input is in fact replicated. The rule (COPY) defines how to use such a replicated input, triggering a replica which executes in parallel with the appropriate continuation Q, sharing a fresh session channel for subsequent communication. The persistent nature of the replicated input ensures that further replication of P is always possible.

For the π-calculus processes we obtain the following:

$$!u(x).P \mid (\nu x)u\langle x\rangle.Q \longrightarrow\ !u(x).P \mid (\nu x)(P \mid Q)$$

Having justified the necessary judgmental principles, we now consider the exponential $!A$. Offering $!A$ means that we must be able to realize A an arbitrary number of times and thus we must commit to not using any linear resources in order to apply the !R rule, since these must be used exactly once. To use $!A$, we simply need to move A to the appropriate unrestricted context, allowing it to be copied an arbitrary number of times.

$$\dfrac{\Gamma;\cdot\vdash A}{\Gamma;\cdot\vdash !A}\,(!\mathsf{R}) \qquad \dfrac{\Gamma,A;\Delta\vdash C}{\Gamma;\Delta,!A\vdash C}\,(!\mathsf{L})$$

The session interpretation of $!A$ is of a session offering the behavior A in a persistent fashion. Thus, the process expression assignment for $!A$ is:

$$\dfrac{\Gamma;\cdot\vdash P_x :: x{:}A}{\Gamma;\cdot\vdash \texttt{output}\ z\ !(x.P_x) :: z{:}!A}\,(!\mathsf{R}) \qquad \dfrac{\Gamma,u{:}A;\Delta\vdash Q_{!u} :: z{:}C}{\Gamma;\Delta,x{:}!A\vdash !u\leftarrow\texttt{input}\ x;Q_{!u} :: z{:}C}\,(!\mathsf{L})$$

The process expression for the !R rule performs an output of a (fresh) persistent session channel along z, after which its continuation will be prepared to receive requests along this channel by spawning new copies of P as required. Dually, the process expression for the $!\mathsf{L}$ rule inputs the persistent session channel along which such requests will be performed by Q.

For π-calculus process assignment, we offer a session of type $!A$ by performing the same output as in the process expression assignment, but we must make the replicated input explicit in the rules:

$$\frac{\Gamma; \cdot \Rightarrow P :: x{:}A}{\Gamma; \cdot \Rightarrow (\nu u)z\langle u\rangle.!u(x).P :: z{:}!A}\ (!\mathsf{R}) \qquad \frac{\Gamma, u{:}A; \Delta \Rightarrow Q :: z{:}C}{\Gamma; \Delta, x{:}!A \Rightarrow x(u).Q :: z{:}C}\ (!\mathsf{L})$$

The proof reduction we obtain from cut elimination is given below (using process expressions):

$$\frac{\dfrac{\Gamma; \cdot \vdash P_y :: y{:}A}{\Gamma; \cdot \vdash \texttt{output}\ x\ !(y.P) :: x{:}!A} \qquad \dfrac{\Gamma, u{:}A; \Delta \vdash Q_{!u} :: z{:}C}{\Gamma; \Delta, x{:}!A \vdash !u \leftarrow \texttt{input}\ x; Q_{!u} :: z{:}C}}{\Gamma; \Delta \vdash \texttt{new}\ x.(\texttt{output}\ x\ !(y.P_y) \parallel (!u \leftarrow \texttt{input}\ x; Q_{!u})) :: z{:}C}\ (\textsc{cut})$$

$$\Longrightarrow \frac{\Gamma; \cdot \vdash P_y :: y{:}A \quad \Gamma, u{:}A; \Delta \vdash Q_{!u} :: z{:}C}{\Gamma; \Delta \vdash \texttt{new}\ !u.(y \leftarrow \texttt{input}\ !u\ P_y \parallel Q_{!u})}\ (\mathsf{cut}^!)$$

Essentially, the cut is transformed into an unrestricted cut which then allows for the reductions of instances of the copy rule as previously shown. The SSOS rule that captures this behavior is:

$$(\textsc{repl})\ \mathsf{exec}\,(\mathsf{output}\,x\,!(y.P_y)) \otimes \mathsf{exec}\,(!u \leftarrow \mathsf{input}\,x; Q_{!u}) \multimap \{\mathsf{exec}\,(\texttt{new}\ !u.(y \leftarrow \mathsf{input}\,!u\ P_y \parallel Q_{!u}))\}$$

The associated π-calculus reduction rule is just an input/output synchronization. It should be noted that the term assignment for the exponential and its associated judgmental principles differs from that of [16]. In [16], the $!\mathsf{L}$ rule was silent at the level of processes and so despite there being a proof reduction in the proof theory, no matching process reduction applied. In the assignment given above this is no longer the case and so we obtain a tighter logical correspondence by forcing reductions on terms to match the (principal) reductions from the proof theory.

2.3.5 Examples

We now present some brief examples showcasing the basic interpretation of linear logic proofs as concurrent processes.

2.3.5.1 A File Storage Service

We consider a simple file storage service to illustrate the basic communication features of our interpretation. We want to encode a service which is able to receive files and serve as a form of persistent storage. To this end, the service must first receive a file and then offer a persistent service that outputs the stored file. We may encode this communication protocol with the following type (for illustration purposes, we use file as an abstraction of a persistent data type representing a file):

$$\mathsf{DBox} \triangleq \mathsf{file} \multimap !(\mathsf{file} \otimes \mathbf{1})$$

The type DBox specifies the required interaction from the perspective of the service: input a file and then persistently offer its output, encoding the file storage. A process expression offering such a service is (given that a file represents persistent data, we allow its usage while offering a persistent session directly):

$$\mathsf{Server} \triangleq f \leftarrow \mathsf{input}\ c; \mathsf{output}\ c\ !(y.\mathsf{output}\ y\ f; \mathsf{close}\ y) :: c{:}\mathsf{DBox}$$

Clients of the service must then behave accordingly: given access to the DBox service along a channel c, they may send it a file, after which they can acess the stored file arbitrarily often via a persistent (replicated) session u:

$$\mathsf{Client} \triangleq \mathsf{output}\ c\ \mathsf{thesis.pdf}; !u \leftarrow \mathsf{input}\ c; P$$

In the Client process expression above, we send to the DBox service the $\mathsf{thesis.pdf}$ file and then receive the handle u to the persistent session along which we may receive the file arbitrarily often. We abstract away this behavior in process expression P, which we assume to offer some type $z{:}C$. Using cut, we may compose the client and the DBox service accordingly:

$$\dfrac{\vdash \mathsf{Server} :: c{:}\mathsf{DBox} \qquad c{:}\mathsf{DBox} \vdash \mathsf{Client} :: z{:}C}{\vdash \mathsf{new}\ c.(\mathsf{Server} \parallel \mathsf{Client}) :: z{:}C}$$

2.3.5.2 On the Commutativity of Multiplicative Conjunction

While our interpretation of multiplicative conjunction appears to be non-commutative insofar as $A \otimes B$ denotes a session which sends a channel that offers A and continues as B, we can produce a process that coerces an ambient session of type $A \otimes B$ to $B \otimes A$ by simply considering the process expression assignment to the proof of $A \otimes B \vdash B \otimes A$, given below (omitting the empty unrestricted context region):

$$\dfrac{\dfrac{\dfrac{}{B \vdash B}\,\text{ID} \qquad \dfrac{}{A \vdash A}\,\text{ID}}{A, B \vdash B \otimes A}\,{\otimes}\mathsf{R}}{A \otimes B \vdash B \otimes A}\,{\otimes}\mathsf{L}$$

$$\dfrac{\dfrac{x{:}B \vdash \mathsf{fwd}\ x'\ x :: x'{:}B \qquad y{:}A \vdash \mathsf{fwd}\ z\ y :: z{:}A}{y{:}A, x{:}B \vdash \mathsf{output}\ z\ (x'.\mathsf{fwd}\ x'\ x); \mathsf{fwd}\ z\ y :: z{:}B \otimes A}}{x{:}A \otimes B \vdash y \leftarrow \mathsf{input}\ x; \mathsf{output}\ z\ (x'.\mathsf{fwd}\ x'\ x); \mathsf{fwd}\ z\ y :: z{:}B \otimes A}$$

Intuitively, the process expression above acts as a mediator between the ambient session channel x which offers the behavior $A \otimes B$ and the users of z which expect the behavior $B \otimes A$. Thus, we first input from x the channel (bound to y) which offers the behavior A, after which x will now offer B. In this state, we fulfil our contract of offering $B \otimes A$ by outputting along z a fresh name (bound to x') and then forwarding between $x{:}B$ and x' and $y{:}A$ and z.

In Section 7.2 we formally establish that indeed $A \otimes B$ and $B \otimes A$ are isomorphic, by showing how the composition of the coercions of $A \otimes B$ and $B \otimes A$ are observationally equivalent to the identity.

2.3.5.3 Embedding a Linear λ-calculus

One of the advantages of working within the realm of proof theory is that we obtain several interesting results "for free", by reinterpreting known proof-theoretic results using our concurrent term assignment.

One such result is an embedding of the linear λ-calculus which is obtained by considering the canonical proof theoretic translation of linear natural deduction (whose term assignment is the linear λ-calculus) to a sequent calculus (for which we have our process expression assignment). Intuitively, the translation maps introduction forms to the respective sequent calculus right rules, whereas the elimination forms are mapped to instances of the cut rule with the respective left rules. We write $[\![M]\!]_z$ for the translation of the linear λ-term M into a process expression P which implements the appropriate session behavior along z. The full details for this embedding can be found in [76], making use of the π-calculus assignment to linear sequents.

We consider a linear λ-calculus with the linear function space and the exponential $!M$, with its respective elimination form $\mathsf{let}\ !u = M\ \mathsf{in}\ N$ where the unrestricted variable u is bound in N (multiplicative or additive pairs and sums are just as easily embedded in our concurrent process expressions but are omitted).

Definition 4 (The $[\![\cdot]\!]_z$ Embedding)**.**

$$\begin{array}{lll}
[\![x]\!]_z & \triangleq & \mathsf{fwd}\ z\ x \\
[\![u]\!]_z & \triangleq & x \leftarrow \mathsf{copy}\ !u; \mathsf{fwd}\ z\ x \\
[\![\hat{\lambda} x.M]\!]_z & \triangleq & x \leftarrow \mathsf{input}\ z; [\![M]\!]_z \\
[\![M\, N]\!]_z & \triangleq & \mathsf{new}\ x.([\![M]\!]_x \mid (\mathsf{output}\ x\ (y.[\![N]\!]_y); \mathsf{fwd}\ z\ x)) \\
[\![!M]\!]_z & \triangleq & \mathsf{output}\ z\ !(x.[\![M]\!]_x) \\
[\![\mathsf{let}\ !u = M\ \mathsf{in}\ N]\!]_z & \triangleq & \mathsf{new}\ x.([\![M]\!]_x \mid !u \leftarrow \mathsf{input}\ x; [\![N]\!]_z)
\end{array}$$

We can understand the translation by considering the execution of a translated β-redex (we write $\longrightarrow$ for the state transformation generated by applying a single SSOS rule and $\longrightarrow^*$ for its reflexive transitive closure):

$$[\![(\hat{\lambda} x.M)\ N]\!]_z \quad = \quad \mathsf{new}\ x'.((x \leftarrow \mathsf{input}\ x'; [\![M]\!]_{x'}) \parallel (\mathsf{output}\ x'\ (y.[\![N]\!]_y); \mathsf{fwd}\ z\ x'))$$

$$\mathsf{exec}(\mathsf{new}\ x'.((x \leftarrow \mathsf{input}\ x'; [\![M]\!]_{x'}) \parallel (\mathsf{output}\ x'\ (y.[\![N]\!]_y); \mathsf{fwd}\ z\ x'))) \quad \longrightarrow^* \quad \exists y.\mathsf{exec}\ [\![M]\!]_z \otimes \mathsf{exec}\ [\![N]\!]_y$$

As expected, the execution of the process expression embedding of a β-redex will execute the embedding of the function body M in parallel with the embedding of the argument N, sharing a local communication channel y.

2.4 Metatheoretical Properties

One of the most important reasons for developing a correspondence between proofs and programs is that several properties of interest follow naturally from logical soundness. First, we note that a simulation between the reduction dynamics of proofs, as given by principal cut elimination steps,

P	$\leadsto$	$\hat{Q}$
$\mathsf{new}\ x.(P \parallel Q)$	$\leadsto$	$(\nu x)(\hat{P} \mid \hat{Q})$
$x \leftarrow \mathsf{input}\ z; Q_x$	$\leadsto$	$z(x).\hat{Q}_x$
$\mathsf{output}\ z\ (y.P_y); Q$	$\leadsto$	$(\nu y)z\langle y\rangle.(\hat{P}_y \mid \hat{Q})$
$\mathsf{close}\ z$	$\leadsto$	$z\langle\rangle.\mathbf{0}$
$\mathsf{wait}\ z; P$	$\leadsto$	$z().\hat{P}$
$z.\mathsf{case}(P, Q)$	$\leadsto$	$z.\mathsf{case}(\hat{P}, \hat{Q})$
$z.\mathsf{inl}; P$	$\leadsto$	$z.\mathsf{inl}; \hat{P}$
$z.\mathsf{inr}; P$	$\leadsto$	$z.\mathsf{inr}; \hat{P}$
$\mathsf{new}\ !u.(x \leftarrow \mathsf{input}\ !u\ P_x \parallel Q_{!u})$	$\leadsto$	$(\nu u)(!u(x).\hat{P}_x \mid \hat{Q}_u)$
$x \leftarrow \mathsf{copy}\ !u; R_x$	$\leadsto$	$(\nu x)u\langle x\rangle.\hat{R}_x$
$\mathsf{output}\ z\ !(x.P_x)$	$\leadsto$	$(\nu u)z\langle u\rangle.!u(x).\hat{P}_x$
$!u \leftarrow \mathsf{input}\ z; Q_{!u}$	$\leadsto$	$z(u).\hat{Q}_u$
$\mathsf{fwd}\ x\ y$	$\leadsto$	$[x \leftrightarrow y]$

Figure 2.2: Translation from Process Expressions to π-calculus Processes

and our process expression assignment is immediate (c.f. [18] for the π-calculus assignment). In the interpretation developed above, the two fundamental properties that follow from logical soundness are type preservation or *session fidelity*, and progress or *deadlock-freedom*. We can formulate the two properties formally using either the π-calculus and the process expression term assignment.

2.4.1 Correspondence of process expressions and π-calculus processes

We now make precise the informal correspondence between the process expression assignment and the π-calculus process assignment to linear logic proofs.

At a purely syntactic level, the correspondence is rather straightforward since we need only match the process expression assignment of each rule to its corresponding π-calculus process assignment. We write $P \leadsto \hat{Q}$ to denote that the well-typed process expression P translates to the π-calculus process $\hat{Q}$. More precisely, we translate *typing derivations* of process expressions to session typing derivations of π-calculus processes, following [16, 75].

Definition 5 (Translation from Process Expressions to π-calculus Processes)**.** *Well-typed process expression P translates to the π-calculus process $\hat{Q}$, written $P \leadsto \hat{Q}$ according to the rules of Fig. 2.2.*

We can thus state the following static correspondence theorem, relating typed process expressions with typed π-calculus processes.

Theorem 1 (Static Correspondence - Process Expressions to π-calculus)**.** *If $\Gamma; \Delta \vdash P :: z{:}A$ then there exists a π-calculus process $\hat{Q}$ such that $P \leadsto \hat{Q}$ with $\Gamma; \Delta \Rightarrow \hat{Q} :: z{:}A$.*

Proof. By induction on typing. □

This correspondence applies in both directions, in the sense that we may identify a typed π-calculus process that corresponds to a typed process expression (following Theorem 1), and conversely, we can identify a typed process expression that corresponds to a typed π-calculus process using the inverse of the translation presented in Def. 5, obtained by reading the table of Fig. 2.2 from right to left (we refer to this relation as $\leadsto^{-1}$). We note that since typing on π-calculus processes is defined modulo structural congruence, the translation $\leadsto^{-1}$ is not unique. In general, many (structurally equivalent) processes are mapped to the same process expression.

Theorem 2 (Static Correspondence - π-calculus to Process Expressions)**.** *If* $\Gamma; \Delta \Rightarrow P :: z{:}A$ *then there exists a process expression* $\hat{Q}$ *such that* $P \leadsto^{-1} \hat{Q}$ *with* $\Gamma; \Delta \vdash \hat{Q} :: z{:}A$.

Proof. Induction on typing. □

2.4.1.1 Identifying SSOS States with π-calculus Processes

To establish an operational correspondence between SSOS execution and π-calculus process reduction we must be able to identify SSOS execution states with π-calculus processes outright.

This mapping of execution states and π-calculus processes takes into account the fact that the global execution state can itself be typed by tracking the individual typing of individual process expressions, allowing us to appeal to our translation of process expressions to π-calculus processes. We first annotate each $\mathsf{exec}\, P$ with the channel z along which P offers its output and its type A. This exploits the observation that every process offers a session along exactly one channel, and for every channel there is exactly one process providing a session along it. This extended form is written $\mathsf{exec}\, P\, z\, A$. The SSOS rules given above can be updated in a straightforward fashion, and the original rules can be recovered by erasure. The annotations fix the role of every channel in a communication as either offered or used, and we can check if the whole process state Ω is well-typed according to a signature of linear channels Δ and unrestricted channels Γ.

We can type an execution state using the following rules:

$$\frac{\Gamma; \Delta_1 \models \Omega :: x{:}A \quad \Gamma; \Delta_2, x{:}A \vdash P :: z{:}C}{\Gamma; \Delta_1, \Delta_2 \models \Omega, \mathsf{exec}\ P\ z\ C :: z{:}C} \qquad \frac{\Gamma, u{:}A; \Delta \models \Omega :: z{:}C \quad \Gamma; \cdot \vdash P :: x{:}A}{\Gamma; \Delta \models \Omega, !\mathsf{exec}\ (x \leftarrow \mathsf{input}\ !u; P)\ !u\ A :: z{:}C}$$

$$\frac{}{\Gamma; \cdot \models (\cdot) :: \varnothing}$$

Thus, given a typed execution state in the sense specified above we can extract from it a π-calculus process via the following rules:

$$\frac{\Gamma; \Delta_1 \models \Omega \leadsto \hat{\Omega} :: x{:}A \quad \Gamma; \Delta_2, x{:}A \vdash P \leadsto \hat{P} :: z{:}C}{\Gamma; \Delta_1, \Delta_2 \models \Omega, \mathsf{exec}\ P\ z\ C \leadsto (\nu x)(\hat{\Omega} \mid \hat{P}) :: z{:}C}$$

$$\frac{\Gamma, u{:}A; \Delta \models \Omega \leadsto \hat{\Omega} :: z{:}C \quad \Gamma; \cdot \vdash P \leadsto \hat{P} :: x{:}A}{\Gamma; \Delta \models \Omega, !\mathsf{exec}\ (x \leftarrow \mathsf{input}\ !u; P)\ !u\ A \leadsto (\nu u)(!u(x).\hat{P} \mid \hat{\Omega}) :: z{:}C}$$

$$\frac{}{\Gamma; \cdot \models (\cdot) \leadsto \mathbf{0} :: \varnothing}$$

Theorem 3. *If* $\Gamma; \Delta \models \Omega :: z{:}A$ *and* $\Omega \leadsto \hat{\Omega}$ *then* $\Gamma; \Delta \Rightarrow \hat{\Omega} :: z{:}A$.

Proof. Straightforward induction, noting that typing for π-calculus processes is modulo structural congruence. □

Having lifted the translation procedure to execution states we can precisely state the operational correspondence between process expression executions and π-calculus reductions.

Theorem 4 (Simulation of SSOS Executions)**.** *Let* $\Gamma;\Delta \vdash P :: z{:}A$*, with* $\mathsf{exec}\ P\ z\ A \longrightarrow^{1,2} \Omega$*. We have that* $P \rightsquigarrow \hat{P}$*, such that the following holds:*

(i) Either, $\hat{P} \rightarrow Q$*, for some* Q*, with* $\Omega \rightsquigarrow\equiv Q$

(ii) Or, $\Omega \rightsquigarrow Q$*, for some* Q*, such that* $\hat{P} \equiv Q$*.*

Proof. We proceed by induction on typing. In order for the execution of P to produce an execution state that reduces, it must necessarily consist of a cut (linear or persistent). Since all execution of cuts generate an SSOS step that is not matched by a π-calculus process reduction, we need to consider what may happen to an execution state after this first mismatched reduction.

In a linear cut, this reduction results in an execution state Ω with the shape (for some Q, R, B, x):

$$\mathsf{exec}\ Q\ x\ B, \mathsf{exec}\ R\ z\ A$$

If the cut is *not* a principal cut (or an identity cut), Q and R cannot synchronize, although additional execution steps may be possible due to other cuts that are now exposed at the top-level of Q or R.

By the definition of $P \rightsquigarrow \hat{P}$, we have that $\hat{P} = (\nu x)(\hat{Q} \mid \hat{R})$. If neither $\mathsf{exec}\ Q\ x\ B$ nor $\mathsf{exec}\ R\ z\ A$ trigger execution steps, we have that $\Omega \rightsquigarrow\equiv (\nu x)(\hat{Q} \mid \hat{R})$, satisfying (ii).

If $\mathsf{exec}\ Q\ x\ B$ can trigger an execution step to some state Ω', then by i.h. we know that either (i) $\hat{Q} \rightarrow Q'$ with $\Omega' \rightsquigarrow\equiv Q'$ or (ii) $\Omega' \rightsquigarrow Q'$ with $\hat{Q} \equiv Q'$. In the either case we immediately satisfy (i) or (ii), respectively. The case when $\mathsf{exec}\ R\ z\ A$ executes is identical.

If the cut is a principal cut (or an identity cut) then $\Omega \longrightarrow \Omega'$ such that the two process expressions have synchronized (through the firing of one of the synchronization rules, or the renaming rule). In this case, observe that $(\nu x)(\hat{Q} \mid \hat{R}) \rightarrow P'$, where $\hat{Q}$ and $\hat{R}$ have also synchronized (since there is a one-to-one correspondence with the SSOS rules). It is then immediate that $\Omega' \rightsquigarrow S$ such that $S \equiv P'$ via appropriate α-conversion and applications of scope extrusion (appropriate typings may be extracted from the typed reductions of the previous section).

In an unrestricted cut the cases are the identical, except the process expression on the left premise of the cut cannot produce any executions (nor corresponding π-reductions) outright.

□

Theorem 5 (Simulation of π-calculus Reduction)**.** *Let* $\Gamma;\Delta \Rightarrow P :: z{:}A$ *with* $P \rightarrow Q$*. We have that* $P \rightsquigarrow^{-1} \hat{P}$ *such that* $\mathsf{exec}\ \hat{P}\ z\ A \longrightarrow^{+} \Omega$ *with* $\Omega \rightsquigarrow\equiv Q$*.*

Proof. We proceed by induction on typing, considering the cases where process P may exhibit a reduction. By inversion we know that the last rule must be a cut and thus $P \equiv (\nu x)(P_1 \mid P_2)$ for some P_1, P_2, x. Observe that $\hat{P} = \mathsf{new}\ x.(\hat{P_1} \parallel \hat{P_2})$.

If the cut is a linear cut, then one possibility is that its a principal cut and so we have that $P \rightarrow P'$ via a synchronization between P_1 and P_2. Observe that

$$\mathsf{exec}\ (\mathsf{new}\ x.(\hat{P}_1 \parallel \hat{P}_2))\ z\ A \longrightarrow \mathsf{exec}\ \hat{P}_1\ x\ B, \mathsf{exec}\ \hat{P}_2\ z\ A = \Omega'$$

for some B and existentially quantified x. By inversion we know that $\Omega' \longrightarrow \Omega''$ by the appropriate typed SSOS synchronization rule, with $\Omega'' \rightsquigarrow\equiv P'$.

If the top-level cut is not a principal cut then the reduction must arise due to another (principal) cut in either P_1 or P_2. Assume $(\nu x)(P_1 \mid P_2) \rightarrow (\nu x)(P_1' \mid P_2) = Q$ with $P_1 \rightarrow P_1'$. By i.h. we know that $\mathsf{exec}\ \hat{P}_1\ x\ B \longrightarrow^+ \Omega_1$ with $\Omega_1 \rightsquigarrow\equiv P_1'$. Since $\mathsf{exec}\ (\mathsf{new}\ x.(\hat{P}_1 \parallel \hat{P}_2))\ z\ A \longrightarrow \mathsf{exec}\ \hat{P}_1\ x\ B \otimes \mathsf{exec}\ \hat{P}_2\ z\ A \longrightarrow^+ \Omega_1 \otimes \mathsf{exec}\ \hat{P}_2\ z\ A$, with $\Omega_1 \rightsquigarrow\equiv P_1'$, it follows that $\Omega_1 \otimes \mathsf{exec}\ \hat{P}_2\ z\ A \rightsquigarrow\equiv (\nu x)(P_1' \mid P_2)$. For reductions in P_2 the reasoning is identical.

In an unrestricted cut, the reasoning is identical except that no internal reductions in P_1 are possible (nor corresponding executions in the SSOS states).

□

Theorems 4 and 5 precisely identify the correspondence between π-calculus reductions and process expression executions via the SSOS framework. Theorem 4 captures the fact that SSOS executions produce some additional administrative steps when compared to the π-calculus reduction semantics, characterizing these intermediate execution states accordingly.

Intuitively, the process expressions that correspond to the assignment of the cut rules always trigger an execution step that decomposes into the two underlying process expressions executing in parallel (rules (CUT) and (UCUT) of the SSOS), identifying the syntactic parallel composition construct with the meta-level composition of SSOS, whereas no such step is present on the π-calculus reductions (rather, the process states before and after a rewrite originating from either rules map to structurally equivalent processes). Thus, execution steps in the SSOS may arise due to these breakdowns of parallel compositions, which are not matched by process reductions but rather identified outright with the process in the image of the translation, up-to structural congruence (case (ii) of Thrm. 4), or due to actual synchronizations according to the SSOS specification for each principal cut, which are matched one-to-one with the π-calculus reduction semantics (case (i) of Thrm. 4). The lifting of $\rightsquigarrow$ to execution states identifies the SSOS-level parallel composition (the $\otimes$) with π-calculus composition.

Theorem 5 identifies a single π-calculus reduction with the multiple SSOS execution steps necessary to identify the corresponding process expression (due to rules (CUT) and (UCUT), which have no corresponding π-calculus reductions and may fire repeatedly due to nested occurrences).

2.4.2 Type Preservation

Having established the correspondence between the process expression assignment and the π-calculus process assignment, we now develop a type preservation result for the π-calculus assignment (which naturally maps back to the process expression language).

Up to this point we have mostly focused on the process expression assignment due to its positive aspects: the lack of a need for reasoning up-to structural congruence, no explicit ν-binders in the syntax and a clear distinction between syntax and semantics which are defined in terms of SSOS rules. While type preservation (and progress) can easily be developed for the process expression assignment by appealing to reductions on (typed) execution states (viz. [77]), which are intrinsically typed, we present our development for the π-calculus assignment mainly for two reasons: on one hand, reduction in the π-calculus is untyped and we wish to also develop type preservation (and later on, progress) without going through the intrinsically typed reductions of the process expression assignment, which include some administrative reductions that are absent from the π-calculus semantics; secondly, and more importantly, our technical development in Part III turns out to be more conveniently expressed using the technical devices of the π-calculus (specifically, labelled transitions) and so for the sake of uniformity of presentation we opt to also develop these results for the π-calculus assignment.

Our development of type preservation, which follows that of [16], relies on a series of reduction lemmas relating process reductions with the proof reduction obtained via composition through the cut rules, appealing to the labelled transition semantics of Fig. 6.1.

Lemma 1 (Reduction Lemma - $\otimes$)**.** *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}A \otimes B$ *with* $P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P'$

(b) $\Gamma; \Delta_2, x{:}A \otimes B \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x(y)} Q'$

Then:

(c) $(\boldsymbol{\nu} x)(P \mid Q) \to (\boldsymbol{\nu} x)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

Proof. By simultaneous induction on typing for P and Q. The possible cases for typing P are $\otimes$R, cut and cut$^!$. The possible cases for typing Q are $\otimes$L, cut, and cut$^!$.

Case: P from $\otimes$R and Q from $\otimes$L

$P = (\nu y)(x\langle y\rangle.(P_1^y \mid P_2^x)) \overset{\overline{(\nu y)x\langle y\rangle}}{\to} (\nu y)(P_1^y \mid P_2^x) = P'^x$ By inversion

$Q^z = x(y).Q'^z \overset{x(y)}{\to} Q'^z$ By inversion

$(\boldsymbol{\nu} x)(P \mid Q) = (\boldsymbol{\nu} x)((\nu y)(x\langle y\rangle.(P_1^y \mid P_2^x)) \mid x(y).Q'^z)$
$\to (\nu y)(P_1^y \mid (\nu x)(P_2^x \mid Q'^z))$ By principal cut reduction

$(\nu y)(P_1^y \mid (\nu x)(P_2^x \mid Q'^z)) \equiv (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(P_1^y \mid P_2^x \mid Q'^z) = R$
$\Gamma; \Delta_1, \Delta_1 \Rightarrow R :: z{:}C$ by typing

Case: P from cut and Q arbitrary.

$P_2^x \overset{\overline{(\nu y)x\langle y\rangle}}{\to} P_2'^x$ with
$P = (\boldsymbol{\nu} n)(P_1^n \mid P_2^x) \overset{\overline{(\nu y)x\langle y\rangle}}{\to} (\boldsymbol{\nu} n)(P_1^n \mid P_2'^x) = P'^x$ By inversion

$(\boldsymbol{\nu} x)(P \mid Q) = (\boldsymbol{\nu} x)((\boldsymbol{\nu} n)(P_1^n \mid P_2^x) \mid Q)$

$$\begin{array}{lr}
\equiv (\boldsymbol{\nu} n)(P_1^n \mid (\boldsymbol{\nu} x)(P_2^x \mid Q)) & \text{by } \equiv \\
\Gamma; \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(P_2'^x \mid Q'^z) :: z{:}C & (1) \text{ by i.h.} \\
(\boldsymbol{\nu} n)(P_1^n \mid (\boldsymbol{\nu} x)(P_2^x \mid Q)) \rightarrow (\boldsymbol{\nu} n)(P_1^n \mid (\boldsymbol{\nu} x)(P_2'^x \mid Q'^z)) & \text{by cut reduction} \\
(\boldsymbol{\nu} n)(P_1^n \mid (\boldsymbol{\nu} x)(P_2'^x \mid Q'^z)) \equiv (\boldsymbol{\nu} x)(\boldsymbol{\nu} n)(P_1^n \mid P_2'^x \mid Q'^z) = R & \text{by } \equiv \\
\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C & \text{by typing and } (1).
\end{array}$$

Case: P from cut$^!$ and Q arbitrary.

$$\begin{array}{lr}
P_2^x \overset{\overline{(\nu y)x\langle y\rangle}}{\rightarrow} P_2'^x \text{ with} & \\
P = (\boldsymbol{\nu} u)((!u(n).P_1^n) \mid P_2^x) \overset{\overline{(\nu y)x\langle y\rangle}}{\rightarrow} (\boldsymbol{\nu} u)((!u(n).P_1^n) \mid P_2'^x) = P'^x & \text{By inversion} \\
(\boldsymbol{\nu} x)(P \mid Q) = (\boldsymbol{\nu} x)((\boldsymbol{\nu} u)((!u(n).P_1^n) \mid P_2^x) \mid Q) & \\
\equiv (\boldsymbol{\nu} u)(!u(n).P_1^n \mid (\boldsymbol{\nu} x)(P_2^x \mid Q)) & \text{By } \equiv \\
\Gamma; \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(P_2'^x \mid Q'^z) :: z{:}C & (1) \text{ By i.h.} \\
(\boldsymbol{\nu} u)(!u(n).P_1^n \mid (\boldsymbol{\nu} x)(P_2^x \mid Q)) \rightarrow (\boldsymbol{\nu} u)(!u(n).P_1^n \mid (\boldsymbol{\nu} x)(P_2'^x \mid Q'^z)) & \text{by cut reduction} \\
\equiv (\boldsymbol{\nu} x)((\boldsymbol{\nu} u)((!u(n).P_1^n) \mid P_2'^x) \mid Q'^z) = R & \text{by } \equiv \\
\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C & \text{by typing and } (1)
\end{array}$$

Case: Q from cut with $Q = (\boldsymbol{\nu} n)(Q_1^n \mid Q_2^z)$ and $x \in fn(Q_1)$.

$$\begin{array}{lr}
Q_1^n \overset{x(y)}{\rightarrow} Q_1'^n \text{ with} & \\
Q = (\boldsymbol{\nu} n)(Q_1^n \mid Q_2^z) \overset{x(y)}{\rightarrow} (\boldsymbol{\nu} n)(Q_1'^n \mid Q_2^z) & \text{By inversion} \\
(\boldsymbol{\nu} x)(P \mid Q) = (\boldsymbol{\nu} x)(P \mid ((\boldsymbol{\nu} n)(Q_1^n \mid Q_2^z))) & \\
\equiv (\boldsymbol{\nu} n)((\boldsymbol{\nu} x)(P \mid Q_1^n) \mid Q_2^z) & \text{By } \equiv \text{ since } x \notin fn(Q_2) \\
\Gamma; \Delta_1, \Delta^* \Rightarrow (\boldsymbol{\nu} x)(P'^x \mid Q_1'^n) :: n{:}D & (1) \text{ By i.h. for some } D \\
(\boldsymbol{\nu} n)((\boldsymbol{\nu} x)(P \mid Q_1^n) \mid Q_2^z) \rightarrow (\boldsymbol{\nu} n)((\boldsymbol{\nu} x)(P'^x \mid Q_1'^n) \mid Q_2^z) & \text{by cut reduction} \\
\equiv (\boldsymbol{\nu} x)(P'^x \mid (\nu n)(Q_1'^n \mid Q_2^z)) = R & \\
\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C & \text{by typing and } (1)
\end{array}$$

Case: Q from cut with $Q = (\boldsymbol{\nu} n)(Q_1^n \mid Q_2^z)$ and $x \in fn(Q_2)$.

$$\begin{array}{lr}
Q_2^z \overset{x(y)}{\rightarrow} Q_2'^z \text{ with} & \\
Q = (\boldsymbol{\nu} n)(Q_1^n \mid Q_2^z) \overset{x(y)}{\rightarrow} (\boldsymbol{\nu} n)(Q_1^n \mid Q_2'^z) & \text{by inversion} \\
(\boldsymbol{\nu} x)(P \mid Q) = (\boldsymbol{\nu} x)(P \mid ((\boldsymbol{\nu} n)(Q_1^n \mid Q_2^z))) & \\
\equiv (\boldsymbol{\nu} n)(Q_1^n \mid (\boldsymbol{\nu} x)(P \mid Q_2^z)) & \text{By } \equiv \text{ since } x \notin fn(Q_1) \\
\Gamma; \Delta_1, \Delta^* \Rightarrow (\boldsymbol{\nu} x)(P'^x \mid Q_2'^z) :: z{:}C & (1) \text{ By i.h.} \\
(\boldsymbol{\nu} n)(Q_1^n \mid (\boldsymbol{\nu} x)(P \mid Q_2^z)) \rightarrow (\boldsymbol{\nu} n)(Q_1^n \mid (\boldsymbol{\nu} x)(P'^x \mid Q_2'^z)) & \text{by cut reduction} \\
\equiv (\boldsymbol{\nu} x)(P'^x \mid (\boldsymbol{\nu} n)(Q_1^n \mid Q_2'^z)) = R & \\
\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C & \text{by typing and } (1).
\end{array}$$

Case: Q from cut$^!$

$Q = (\nu u)((!u(n).Q_1^n) \mid Q_2^z)$ by inversion

$x \in FV(Q_2^z)$ by inversion

$Q_2^z \overset{x(y)}{\to} Q_2'^z$ with

$(\nu u)((!u(n).Q_1^n) \mid Q_2^z) \overset{x(y)}{\to} (\nu u)((!u(n).Q_1^n) \mid Q_2'^z)$ By inversion

$(\nu x)(P \mid Q) = (\nu x)(P \mid (\nu u)((!u(n).Q_1^n) \mid Q_2^z))$

$\equiv (\nu u)((!u(n).Q_1^n) \mid (\nu x)(P \mid Q_2^z))$ by $\equiv$

$\Gamma; \Delta_1, \Delta_2 \Rightarrow (\nu x)(P' \mid Q_2'^z) :: z{:}C$ (1) by i.h.

$(\nu u)((!u(n).Q_1^n) \mid (\nu x)(P \mid Q_2^z)) \to (\nu u)((!u(n).Q_1^n) \mid (\nu x)(P' \mid Q_2'^z))$

by cut reduction

$\equiv (\nu x)(P'^x \mid (\nu u)((!u(n).Q_1^n) \mid Q_2'^z)) = R$

$\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$ by typing and (1).

□

Lemma 1 above characterizes the fact that a process P that is prepared to offer the session behavior $A \otimes B$ along x may interact with a process Q that is prepared to use the given session behavior such that the corresponding process continuations remain well-typed, according to the ascribed typings. The following series of Lemmas follows the same line of reasoning, characterizing the interactive behavior between processes that offer and use a given session type. The proofs follow from the same reasoning as the proof above (the crucial case is that of the principal cut reduction, the rest follow by i.h.) and are omitted for conciseness.

Lemma 2 (Reduction Lemma - $\multimap$)**.** *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}A \multimap B$ *with* $P \xrightarrow{x(y)} P'$

(b) $\Gamma; \Delta_2, x{:}A \multimap B \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{(\nu y)x\langle y\rangle}} Q'$

Then:

(c) $(\nu x)(P \mid Q) \to (\nu x)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

Lemma 3 (Reduction Lemma - **1**)**.** *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}\mathbf{1}$ *with* $P \xrightarrow{\overline{x\langle\rangle}} P'$

(b) $\Gamma; \Delta_2, x{:}\mathbf{1} \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x()} Q'$

Then:

(c) $(\nu x)(P \mid Q) \to (\nu x)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

Lemma 4 (Reduction Lemma - &1)**.** *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}A \,\&\, B$ *with* $P \xrightarrow{x.\mathsf{inl}} P'$

(b) $\Gamma; \Delta_2, x{:}A \,\&\, B \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{x.\mathsf{inl}}} Q'$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

Lemma 5 (Reduction Lemma - &2). *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}A \,\&\, B$ *with* $P \xrightarrow{x.\mathsf{inr}} P'$

(b) $\Gamma; \Delta_2, x{:}A \,\&\, B \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{x.\mathsf{inr}}} Q'$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

Lemma 6 (Reduction Lemma - $\oplus$1). *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}A \oplus B$ *with* $P \xrightarrow{\overline{x.\mathsf{inl}}} P'$

(b) $\Gamma; \Delta_2, x{:}A \oplus B \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x.\mathsf{inl}} Q'$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

Lemma 7 (Reduction Lemma - $\oplus$2). *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}A \oplus B$ *with* $P \xrightarrow{\overline{x.\mathsf{inr}}} P'$

(b) $\Gamma; \Delta_2, x{:}A \oplus B \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x.\mathsf{inr}} Q'$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

Lemma 8 (Reduction Lemma - !). *Assume*

(a) $\Gamma; \Delta_1 \Rightarrow P :: x{:}!A$ *with* $P \xrightarrow{\overline{(\nu u)x\langle u \rangle}} P'$

(b) $\Gamma; \Delta_2, x{:}!A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x(u)} Q'$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu u)(P' \mid Q') \equiv R$ *such that* $\Gamma; \Delta_1, \Delta_2 \Rightarrow R :: z{:}C$

We also require a reduction lemma that characterizes process behavior for persistent sessions. Given a process P that offers behavior $y{:}A$ without using any linear resources and a process Q that can make use of a persistent behavior $u{:}A$ by performing the appropriate output action that will synchronize with the replicated input associated with persistent sessions, we may characterize the interaction between P and Q by identifying the appropriate well-typed process that results from the spawning of the replica of P: a composition of the persistent service P (replicated under an input guard), with a replica P composed with the continuation Q', both sharing the linear channel y.

Lemma 9 (Reduction Lemma - $\mathsf{cut}^!$)**.** *Assume*

(a) $\Gamma; \cdot \Rightarrow P :: y{:}A$

(b) $\Gamma, u{:}A; \Delta \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{(\nu y)u\langle y\rangle}} Q'$

Then:

(c) $(\nu u)(!u(y).P \mid Q) \rightarrow (\nu u)(!u(y).P \mid R)$, *for some* $R \equiv (\nu y)(P \mid Q')$

(d) $\Gamma, u{:}A; \Delta \Rightarrow R :: z{:}C$

Finally, we characterize the action labels that can be offered by processes offering or using a given session type as follows in Lemma 40.

Lemma 10. *Let* $\Gamma; \Delta \Rightarrow P :: x{:}C$.

1. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = \mathbf{1}$ *then* $\alpha = \overline{x\langle\rangle}$
2. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : \mathbf{1} \in \Delta$ *then* $\alpha = y()$.
3. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \otimes B$ *then* $\alpha = \overline{(\nu y)x\langle y\rangle}$.
4. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \otimes B \in \Delta$ *then* $\alpha = y(z)$.
5. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \multimap B$ *then* $\alpha = x(y)$.
6. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \multimap B \in \Delta$ *then* $\alpha = \overline{(\nu z)y\langle z\rangle}$.
7. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \,\&\, B$ *then* $\alpha = x.\mathsf{inl}$ *or* $\alpha = x.\mathsf{inr}$.
8. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \,\&\, B \in \Delta$ *then* $\alpha = \overline{y.\mathsf{inl}}$ *or* $\alpha = \overline{y.\mathsf{inr}}$.
9. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \oplus B$ *then* $\alpha = \overline{x.\mathsf{inl}}$ *or* $\alpha = \overline{x.\mathsf{inr}}$.
10. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \oplus B \in \Delta$ *then* $\alpha = y.\mathsf{inl}$ *or* $\alpha = y.\mathsf{inr}$.
11. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = !A$ *then* $\alpha = \overline{(\nu u)x\langle u\rangle}$.
12. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : !A \in \Delta$ *then* $\alpha = y(u)$.
13. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = u$ *and* $u : A \in \Gamma$ *then* $\alpha = \overline{(\nu z)u\langle z\rangle}$.

Proof. See Appendix A.1 □

Having established the necessary reduction lemmas, we may now show the desired subject reduction, or type preservation property, which states that process reductions preserve typing.

Theorem 6 (Type Preservation)**.** *If* $\Gamma; \Delta \Rightarrow P :: z{:}A$ *and* $P \rightarrow Q$ *then* $\Gamma; \Delta \Rightarrow Q :: z{:}A$.

Proof. By induction on the typing of P.

The possible cases are when the last typing rule is cut or cut$^!$. In all other cases P cannot offer a reduction step.

Case: P from cut$^!$

$P \equiv (\nu u)(!u(w).P_1 \mid P_2)$
$\Gamma; \cdot \Rightarrow P_1 :: w{:}C$
$\Gamma, w{:}C; \Delta \vdash P_2 :: z{:}A$ by inversion
From $P \rightarrow Q$ either
(1) $P_2 \rightarrow Q_2$ and $Q = (\nu u)(!u(w).P_1 \mid Q_2)$
(2) $P_2 \xrightarrow{\overline{(\nu w)u\langle w\rangle}} Q_2$ and $Q = (\nu u)(!u(w).P_1 \mid (\nu w)(P_1 \mid Q_2))$
Subcase (1):
$\Gamma, w{:}C; \Delta \Rightarrow Q_2 :: z{:}A$ by i.h.
$P \rightarrow (\nu u)(!u(w).P_1 \mid Q_2) = Q$
$\Gamma; \Delta \vdash Q :: z{:}A$ by cut$^!$
Subcase (2):
$(\nu u)(!u(w).P_1 \mid P_2) \rightarrow (\nu u)(!u(w).P_1 \mid R)$ with $R \equiv (\nu w)(P_1 \mid Q_2)$
$\Gamma, w{:}C; \Delta \Rightarrow R :: z{:}A$ by Lemma 9
$\Gamma; \Delta \Rightarrow Q :: z{:}A$ by cut$^!$

Case: P from cut

$P \equiv (\nu x)(P_1 \mid P_2)$
$\Delta = (\Delta_1, \Delta_2)$
$\Gamma; \Delta_1 \Rightarrow P_1 :: x{:}C$
$\Gamma; \Delta_2, x{:}C \Rightarrow P_2 :: z{:}A$ by inversion
Since $P \rightarrow Q$ there are four subcases:
(1) $P_1 \rightarrow Q_1$ and $Q = (\nu x)(Q_1 \mid P_2)$
(2) $P_2 \rightarrow Q_2$ and $Q = (\nu x)(P_1 \mid Q_2)$
(3) $P_1 \xrightarrow{\alpha} Q_1$ and $P_2 \xrightarrow{\overline{\alpha}} Q_2$
(4) $P_1 \xrightarrow{\overline{\alpha}} Q_1$ and $P_2 \xrightarrow{\alpha} Q_2$

Subcase (1): $P_1 \rightarrow Q_1$
$\Gamma; \Delta_1 \Rightarrow Q_1 :: x{:}C$ by i.h.
$(\nu x)(P_1 \mid P_2) \rightarrow (\nu x)(Q_1 \mid P_2)$ by reduction

$\Gamma; \Delta \vdash (\boldsymbol{\nu} x)(Q_1 \mid P_2) :: z{:}A$ by cut

Subcase (2): $P_2 \rightarrow Q_2$

Symmetric to **Subcase (1)**.

Subcase (3): $P_1 \xrightarrow{\alpha} Q_1$ and $P_2 \xrightarrow{\overline{\alpha}} Q_2$

Subsubcase: $C = \mathbf{1}$

not possible

Subsubcase: $C = C_1 \,\&\, C_2$

$\alpha = x.\mathsf{inl}$ or $\alpha = x.\mathsf{inr}$ By Lemma 40

$P \rightarrow R$ for some R

$\Gamma; \Delta \Rightarrow R :: z : A$

with $R \equiv (\boldsymbol{\nu} x)(Q_1 \mid Q_2) = Q$ by Lemmas 4 and 5

Subsubcase: $C = C_1 \oplus C_2$

not possible

Subsubcase: $C = C_1 \otimes C_2$

not possible

Subsubcase: $C = C_1 \multimap C_2$

$\alpha = x(y)$ and $\overline{\alpha} = \overline{(\nu y)x\langle y\rangle}$ By Lemma 40

$P \rightarrow R$ for some R

$\Gamma; \Delta \Rightarrow R :: z : A$

with $R \equiv (\nu x)(\nu y)(Q_1 \mid Q_2) = Q$ by Lemma 2

Subsubcase: $C = !C_1$

not possible

Subcase (4): $P_1 \xrightarrow{\overline{\alpha}} Q_1$ and $P_2 \xrightarrow{\alpha} Q_2$

Subsubcase: $C = \mathbf{1}$

$\overline{\alpha} = \overline{x\langle\rangle}$ and $\alpha = x()$ By Lemma 40

$P \rightarrow R$ for some R

$\Gamma; \Delta \Rightarrow R :: z{:}A$

with $R \equiv (\boldsymbol{\nu} x)(Q_1 \mid Q_2) = Q$ by Lemma 3

Subsubcase: $C = C_1 \,\&\, C_2$

not possible

Subsubcase: $C = C_1 \oplus C_2$

$\overline{\alpha} = \overline{x.\mathsf{inl}}$ or $\overline{\alpha} = \overline{x.\mathsf{inr}}$ By Lemma 40

$P \rightarrow R$ for some R

$\Gamma; \Delta \Rightarrow R :: z{:}A$

with $R \equiv (\nu x)(Q_1 \mid Q_2) = Q$ by Lemmas 6 and 7

Subsubcase: $C = C_1 \otimes C_2$

$\overline{\alpha} = \overline{(\nu y)x\langle y\rangle}$ and $\alpha = x(y)$ By Lemma 40

$P \rightarrow R$ for some R

$\Gamma; \Delta \Rightarrow R :: z : A$

with $R \equiv (\nu x)(\nu y)(Q_1 \mid Q_2) = Q$ by Lemma 1

Subsubcase: $C = C_1 \multimap C_2$

not possible

Subsubcase: $C = !C_1$

$\overline{\alpha} = \overline{(\nu u)x\langle u\rangle}$ and $\overline{\alpha} = x(u)$ By Lemma 40

$P \rightarrow R$ for some R

$\Gamma; \Delta \vdash R :: z : A$

with $R \equiv (\nu x)(\nu y)(Q_1 \mid Q_2) = Q$ by Lemma 8

□

Notice how Theorem 6 entails *session fidelity*, in the sense that a well-typed process is guaranteed to adhere to ascribed session behavior throughout its execution. Moreover, we can straightforwardly combine type preservation for the π-calculus assignment with the static and operational correspondence results for the process expression assignment, obtaining type preservation (and session fidelity) for both assignments outright.

2.4.3 Global Progress

In this section we establish the result of *global progress* (i.e., progress for multiple sessions) for the π-calculus assignment. Traditional session type systems typically fail to enforce a strong notion of progress [31, 43], or require extraneous techniques to ensure progress such as well-founded ordering on interaction events [26, 46]. In our work, the logical basis for our type systems intrinsically ensures that well-typed process compositions can always make progress, insofar as that they can either produce internal reductions, await for interaction from the environment as specified by their offered session type, or simply have no behavior left to offer (i.e., a terminated session).

In order to establish our global progress property, we first require the notion of a *live* process, defined below.

Definition 6 (Live Process). *A process P is considered live, written $live(P)$, if and only if the following holds:*

$$live(P) \triangleq P \equiv (\nu \overline{n})(\pi.Q \mid R) \textit{ for some } \pi.Q, R, \overline{n}$$

where $\pi.Q$ is a non-replicated *guarded process with $\pi \neq x\langle\rangle$ for any x.*

Our definition of a live process intuitively captures the shape of a process that has not yet fully carried out its intended session, and thus still has some sub-process $\pi.Q$ which is guarded by an action that is not terminating a session, composed with some process R which may potentially synchronize with the prefix π (or with some other prefix in Q), make progress outright or consist of terminated persistent sessions (these are implemented by input-guarded replicated processes and so are never fully consumed).

We first establish the following contextual progress property, from which Theorem 7 follows as a corollary.

Lemma 11 (Contextual Progress)**.** *Let* $\Gamma; \Delta \Rightarrow P :: z{:}C$. *If* $live(P)$ *then there is* Q *such that one of the following holds:*

(a) $P \rightarrow Q$,

(b) $P \xrightarrow{\alpha} Q$ *for some* α *where* $s(\alpha) \in z, \Gamma, \Delta$

Proof. By induction on the typing of P. In all right and left rules we satisfy condition (b) immediately with $s(\alpha) = z$ for right rules and $s(\alpha) = x \in \Delta$ for left rules. For the copy rule we satisfy (b) with $s(\alpha) = u \in \Gamma$. The interesting cases are those pertaining to cut and cut$^!$.

Case: cut

$\Delta = \Delta_1, \Delta_2$
$\Gamma; \Delta_1 \Rightarrow P_1 :: x{:}A$
$\Gamma; \Delta_2, x : A \Rightarrow P_2 :: z{:}C$
$P \equiv (\nu x)(P_1 \mid P_2)$ by inversion
$live(P_1)$ or $live(P_2)$ since $live(P)$

Case (1): $live(P_1)$ and $live(P_2)$.
There is P_1' such that either $P_1 \rightarrow P_1'$, or $P_1 \xrightarrow{\alpha_1} P_1'$
for some α_1 with $s(\alpha_1) \in x, \Gamma, \Delta_1$
There is P_2' such that either $P_2 \rightarrow P_2'$, or $P_2 \xrightarrow{\alpha_2} P_2'$
for some α_2 with $s(\alpha_2) \in x, \Gamma, \Delta_2, z$. by i.h.

Subcase (0.1): $P_1 \rightarrow P_1'$ or $P_2 \rightarrow P_2'$
$P \rightarrow Q$ [satisfying (a)]

Subcase (1.1): $s(\alpha_1) \neq x$
$P \xrightarrow{\alpha_1} Q \equiv (\nu x)(P_1' \mid P_2)$ with $\alpha_1 \in \Gamma, \Delta$ [satisfying (b)]

Subcase (1.2): $s(\alpha_2) \neq x$
$P \xrightarrow{\alpha_2} Q \equiv (\nu x)(P_1 \mid P_2')$ with $\alpha_2 \in z, \Gamma, \Delta$ [satisfying (b)]

Subcase (1.3): $s(\alpha_1) = s(\alpha_2) = x$
$\alpha_2 = \overline{\alpha_1}$ By Lemma 40
$P \rightarrow Q$ with $Q \equiv (\nu x)(\nu y)(P_1' \mid P_2')$ or $Q \equiv (\nu x)(P_1' \mid P_2')$ [satisfying (a)]

Case (2): not $live(P_1)$ and $live(P_2)$
There is P_2' such that either $P_2 \rightarrow P_2'$, or $P_2 \xrightarrow{\alpha_2} P_2'$
for some α_2 with $s(\alpha_2) \in x, \Gamma, \Delta_2, z$ by i.h.

Subcase (2.1): $P_2 \rightarrow P_2'$
$P \rightarrow Q$ with $Q \equiv (\nu x)(P_1 \mid P_2')$ [satisfying (a)]

Subcase (2.2): $P_2 \xrightarrow{\alpha_2} P_2'$

Subcase (2.2.1): $s(\alpha_2) \neq x$
$P \xrightarrow{\alpha_2} Q$ with $Q \equiv (\nu x)(P_1 \mid P_2')$ [satisfying (b)]

Subcase (2.2.2): $s(\alpha_2) = x$
$P_1 \equiv (\nu \overline{y})(!x(w).R_1' \mid R_1'')$ or $P_1 \equiv [y \leftrightarrow x]$, for some $y \in \Delta_1$,
or $P_1 \equiv (\boldsymbol{\nu} \overline{y})(x\langle\rangle.\mathbf{0} \mid R_1')$, with $x{:}\mathbf{1} \in \Delta_2$
Subcase (2.2.2.1): $P_1 \equiv (\nu \overline{y})(!x(w).R_1' \mid R_1'')$
Impossible, since not $live(P_1)$ would imply $P_1 \equiv (\nu \overline{y})(!x(w).R_1' \mid R_1'')$,
which cannot be the case since x is linear.

Subcase (2.2.2.2): $P_1 \equiv [y \leftrightarrow x]$, for some $y \in \Delta_1$.
$P \rightarrow Q$ with $Q \equiv P_2\{y/x\}$

Subcase (2.2.2.3): $P_1 \equiv (\boldsymbol{\nu} \overline{y})(x\langle\rangle.\mathbf{0} \mid R_1')$, with $x{:}\mathbf{1} \in \Delta_2$
$\alpha_2 = x()$ by Lemma 40
$P \rightarrow Q$ with $Q \equiv (\boldsymbol{\nu} x)((\boldsymbol{\nu} \overline{y})R_1' \mid P_2')$ [satisfying (a)]

Case (3): $live(P_1)$ and not $live(P_2)$
There is P_1' such that either $P_1 \rightarrow P_1'$, or $P_1 \xrightarrow{\alpha_1} P_1'$
for some α_1 with $s(\alpha_1) \in \Gamma, \Delta_1, x$ by i.h.
Subcase (3.1): $P_1 \rightarrow P_1'$
$P \rightarrow Q$ with $Q \equiv (\nu x)(P_1' \mid P_2)$ [satisfying (a)]

Subcase (3.1): $P_1 \xrightarrow{\alpha_1} P_1'$
for some α_1 with $s(\alpha_1) \in \Gamma, \Delta_1, x$
Subcase (3.1.1) $s(\alpha_1) = x$
$P_2 \equiv (\nu \overline{y})(!x(w).R_1' \mid R_1'')$ or $P_2 \equiv [x \leftrightarrow z]$ or $P_2 \equiv (\boldsymbol{\nu} \overline{y})(x\langle\rangle.\mathbf{0} \mid R_1')$, with $x{:}\mathbf{1}$
Subcase (3.1.1.1):
Impossible, since not $live(P_2)$ would imply $P_2 \equiv (\nu \overline{y})(!x(w).R_1' \mid R_1'')$,
which cannot be the case since x is linear.
Subcase (3.1.1.2):
$P \rightarrow Q$ with $Q \equiv P_1\{z/x\}$ [satisfying (a)]
Subcase (3.1.1.3):
Impossible since then P_2 would have to be live to consume $x{:}\mathbf{1}$.

Case: $\mathsf{cut}^!$

$\Gamma; \cdot \Rightarrow P_1 :: y{:}A$
$\Gamma, u{:}A; \Delta \Rightarrow P_2 :: z{:}C$
$P \equiv (\nu u)(!u(y).P_1 \mid P_2)$ by inversion
$live(P_2)$ since $live(P)$
There is P_2' such that either $P_2 \rightarrow P_2'$, or $P_2 \xrightarrow{\alpha_2} P_2'$
for some α_2 with $s(\alpha_2) \in u, \Gamma, \Delta, z$ by i.h.

Subcase (1): $P_2 \rightarrow P_2'$
$P \rightarrow Q$ with $Q \equiv (\nu u)(!u(y).P_1 \mid P_2')$ [satisfying (a)]

Subcase (2): $P_2 \xrightarrow{\alpha_2} P_2'$

Subcase (2.1): $s(\alpha_2) \neq u$

$P \xrightarrow{\alpha_2} Q$ with $Q \equiv (\nu u)(!u(y).P_1 \mid P_2')$

where $s(\alpha_2) \in \Gamma, \Delta, z$ [satisfying (b)]

Subcase (2.2): $s(\alpha_2) = u$

$P_2 \xrightarrow{\overline{(\nu y)u\langle y\rangle}} P_2'$ By Lemma 40

$!u(y).P_1 \xrightarrow{u(y)} (P_1 \mid !u(y).P_1)$

$P \rightarrow Q$ with $Q \equiv (\nu x)(\nu y)(P_1 \mid !u(y).P_1 \mid P_2')$ [satisfying (a)]

□

Having characterized the conditions under which a live process may reduce or interact with its environment in Lemma 11, our global progress result follows as a corollary by considering a closed *live* process offering behavior **1**. Note that such a process, in order to satisfy the liveness criteria and offer $x{:}\mathbf{1}$, must be made up of the composition of potentially multiple processes via cut and cut$^!$.

Theorem 7 (Global Progress)**.** *Let* $\cdot ; \cdot \Rightarrow P :: x{:}\mathbf{1}$. *If* $live(P)$ *then there is* Q *such that* $P \rightarrow Q$.

Proof. From Lemma 11 we know that $P \rightarrow Q$ or $P \xrightarrow{\alpha} Q$ for some α where $s(\alpha) = x$, since the linear and unrestricted context regions are empty. However, since $live(P)$ we know that no such α may exist and so $P \rightarrow Q$. □

A key aspect of global progress is that it entails *deadlock freedom* of well-typed processes (and consequently of well-typed process expressions) insofar as a well-typed process (resp. process expression) that is still meant to offer some behavior will be guaranteed by typing to do so.

2.5 Summary and Further Discussion

In this chapter we have developed the basis for our concurrent interpretation of linear logic as a concurrent programming language by developing a faithful assignment of concurrent process expressions to the rules of (intuitionistic) linear logic, for which the corresponding operational semantics is given as an SSOS specification that follows the computational content of the cut elimination proof. Moreover, we develop a π-calculus process assignment for the same logic, following [16], opting for a more logically faithful interpretation compared to that of [16]. For reference, we give in Fig. 2.3 a full summary of the typing rules for the process expression assignment; and in Fig. 2.4 the complete set of SSOS rules that define the operational semantics of process expressions.

We have developed a correspondence result between both the static and operational semantics of the process expression assignment and the π-calculus assignment (Section 2.4.1) and established type preservation (Section 2.4.2) and global progress (Section 2.4.3) results for our interpretation, which follow from the logical soundness of the framework.

$$\frac{\Gamma; \Delta_1 \vdash P_x :: x{:}A \quad \Gamma; \Delta_2, x{:}A \vdash Q_x :: z{:}C}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{new}\ x.(P_x \| Q_x) :: z{:}C}\ (\textsc{cut}) \qquad \frac{}{\Gamma; x{:}A \vdash \texttt{fwd}\ z\ x :: z{:}A}\ (\textsc{id})$$

$$\frac{\Gamma; \Delta_1 \vdash P_y :: y{:}A \quad \Gamma; \Delta_2 \vdash Q :: z{:}B}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{output}\ z\ (y.P_y); Q :: z{:}A \otimes B}\ (\otimes\mathsf{R})$$

$$\frac{\Gamma; \Delta, y{:}A, x{:}B \vdash R_y :: z{:}C}{\Gamma; \Delta, x{:}A \otimes B \vdash y \leftarrow \texttt{input}\ x; R_y :: z{:}C}\ (\otimes\mathsf{L})$$

$$\frac{\Gamma; \Delta, x{:}A \vdash P_x :: z{:}B}{\Gamma; \Delta \vdash x \leftarrow \texttt{input}\ z; P_x :: z{:}A \multimap B}\ (\multimap\mathsf{R})$$

$$\frac{\Gamma; \Delta_1 \vdash Q :: y{:}A \quad \Gamma; \Delta_2, x{:}B \vdash R :: z{:}C}{\Gamma; \Delta_1, \Delta_2, x{:}A \multimap B \vdash \texttt{output}\ x\ (y.Q_y); R :: z{:}C}\ (\multimap\mathsf{L})$$

$$\frac{}{\Gamma; \cdot \vdash \texttt{close}\ z :: z{:}\mathbf{1}}\ (\mathbf{1}\mathsf{R}) \qquad \frac{\Gamma; \Delta \vdash Q :: z{:}C}{\Gamma; \Delta, x{:}\mathbf{1} \vdash \texttt{wait}\ x; Q :: z{:}C}\ (\mathbf{1}\mathsf{L})$$

$$\frac{\Gamma; \Delta \vdash P :: z{:}A \quad \Gamma; \Delta \vdash Q :: z{:}B}{\Gamma; \Delta \vdash z.\texttt{case}(P, Q) :: z{:}A \mathbin{\&} B}\ (\&\mathsf{R})$$

$$\frac{\Gamma; \Delta, x{:}A \vdash R :: z{:}C}{\Gamma; \Delta, x{:}A \mathbin{\&} B \vdash x.\texttt{inl}; R :: z{:}C}\ (\&\mathsf{L}_1) \qquad \frac{\Gamma; \Delta, x{:}B \vdash R :: z{:}C}{\Gamma; \Delta, x{:}A \mathbin{\&} B \vdash x.\texttt{inr}; R :: z{:}C}\ (\&\mathsf{L}_2)$$

$$\frac{\Gamma; \Delta \vdash P :: z{:}A}{\Gamma; \Delta \vdash z.\texttt{inl}; P :: z{:}A \oplus B}\ (\oplus\mathsf{R}_1) \qquad \frac{\Gamma; \Delta \vdash P :: z{:}B}{\Gamma; \Delta \vdash z.\texttt{inr}; P :: z{:}A \oplus B}\ (\oplus\mathsf{R}_2)$$

$$\frac{\Gamma; \Delta, x{:}A \vdash Q :: z{:}C \quad \Gamma; \Delta, x{:}B \vdash R :: z{:}C}{\Gamma; \Delta, x{:}A \oplus B \vdash x.\texttt{case}(Q, R) :: z{:}C}\ (\oplus\mathsf{L})$$

$$\frac{\Gamma; \cdot \vdash P_x :: x{:}A \quad \Gamma, u{:}A; \Delta \vdash Q_{!u} :: z{:}C}{\Gamma; \Delta \vdash \texttt{new}\ !u.(x \leftarrow \texttt{input}\ !u\ P_x \parallel Q_{!u}) :: z{:}C}\ (\mathsf{cut}^{!})$$

$$\frac{\Gamma, u{:}A; \Delta, x{:}A \vdash R_x :: z{:}C}{\Gamma, u{:}A; \Delta \vdash x \leftarrow \texttt{copy}\ !u; R_x :: z{:}C}\ (\mathsf{copy})$$

$$\frac{\Gamma; \cdot \vdash P_x :: x{:}A}{\Gamma; \cdot \vdash \texttt{output}\ z\ !(x.P_x) :: z{:}!A}\ (!\mathsf{R}) \qquad \frac{\Gamma, u{:}A; \Delta \vdash Q_{!u} :: z{:}C}{\Gamma; \Delta, x{:}!A \vdash !u \leftarrow \texttt{input}\ x; Q_{!u} :: z{:}C}\ (!\mathsf{L})$$

Figure 2.3: Process Expression Assignment for Intuitionistic Linear Logic

$$(\textsc{cut})\ \mathsf{exec}(\mathtt{new}\ x.(P_x \| Q_x)) \multimap \{\exists x'.\mathsf{exec}(P_{x'}) \otimes \mathsf{exec}(Q_{x'})\}$$

$$(\textsc{fwd})\ \mathsf{exec}\,(\mathsf{fwd}\,x\,z) \multimap \{x = z\}$$

$$(\textsc{scom})\ \mathsf{exec}\,(\mathsf{output}\,x\,(y.P_y); Q) \otimes \mathsf{exec}\,(y \leftarrow \mathsf{input}\,x; R_y) \multimap \{\exists y'.\,\mathsf{exec}\,(P_{y'}) \otimes \mathsf{exec}\,(Q) \otimes \mathsf{exec}\,(R_{y'})\}$$

$$(\textsc{close})\ \mathsf{exec}\,(\mathsf{close}\,x) \otimes \mathsf{exec}\,(\mathsf{wait}\,x; P) \multimap \{\mathsf{exec}\,(P)\}$$

$$(\textsc{choice1})\ \mathsf{exec}\,(x.\mathtt{inl}; P) \otimes \mathsf{exec}\,(x.\mathsf{case}(Q, R)) \multimap \{\mathsf{exec}\,(P) \otimes \mathsf{exec}\,(Q)\}$$

$$(\textsc{choice2})\ \mathsf{exec}\,(x.\mathtt{inr}; P) \otimes \mathsf{exec}\,(x.\mathsf{case}(Q, R)) \multimap \{\mathsf{exec}\,(P) \otimes \mathsf{exec}\,(Q)\}$$

$$(\textsc{ucut})\ \mathsf{exec}\,(\mathtt{new}\ !u.(x \leftarrow \mathtt{input}\ !u\ P_x; Q)) \multimap \{\exists !u.!\mathsf{exec}(x \leftarrow \mathtt{input}\ !u\ P_x) \otimes \mathsf{exec}(Q_{!u})\}$$

$$(\textsc{copy})\ !\mathsf{exec}\,(x \leftarrow \mathsf{input}\,!u; P_x) \otimes \mathsf{exec}\,(x \leftarrow \mathsf{copy}\,!u; Q_x) \multimap \{\exists c'.\,\mathsf{exec}(P_{c'}) \otimes \mathsf{exec}(Q_{c'})\}$$

$$(\textsc{repl})\ \mathsf{exec}\,(\mathsf{output}\,x\,!(y.P_y)) \otimes \mathsf{exec}\,(!u \leftarrow \mathsf{input}\,x; Q_{!u}) \multimap \{\mathsf{exec}\,(\mathtt{new}\ !u.(y \leftarrow \mathsf{input}\,!u\ P_y \parallel Q_{!u}))\}$$

Figure 2.4: Process Expression SSOS Rules

We note that our assignment for the multiplicative unit entails process action, unlike that of [16] which has no associated observable action for either the multiplicative unit **1** nor for the assignment for the exponential. In their assignment, the **1**R rule simply consists of the inactive process **0**, whereas the **1**L rule has no associated prefix, meaning that the associated proof reduction has no matching process reduction. This makes it so that processes that use and offer a channel of type **1** effectively never interact and so their composition is a form of independent parallel composition akin to the mix rule of linear logic, which is not directly expressible in our interpretation without modifying the assignment for the unit (this is consistent with the judgmental analysis of linear logic of [21]). Similarly, their assignment for the exponential (input-guarded replication for !R and silent renaming between the linear and unrestricted channel for !L) makes it so that the proof reduction from a linear cut of ! to a $\mathsf{cut}^!$ has no associated process reduction.

At a technical level, this requires a more specialized treatment of **1** and ! in the proofs of type preservation and progress and their associated auxiliary lemmas, whereas our assignment for these connectives (which comes at the expense of reduced levels of parallelism) allows for a uniform treatment of all connectives in the proofs.

CHAPTER 3

BEYOND PROPOSITIONAL LINEAR LOGIC

In this chapter we aim to support the more general claim that linear logic can indeed be used as a suitable foundation for message-passing concurrent computation by extending the connection beyond the basic session typing discipline of Chapter 2. Specifically, we account for the idea of *value-dependent session types*, that is, session types that may depend on the values exchanged during communication, and parametric polymorphism (in the sense of Reynolds [64, 65]) at the level of session types. The former allows us to express richer properties within the type structure by considering *first-order linear logic*, describing not just the interactive behavior of systems but also constraints and properties of the values exchanged during communication. Moreover, our logically grounded framework enables us to express a high-level form of *proof-carrying code* in a concurrent setting, where programs may exchange proof certificates that attest for the validity of the properties asserted in types. As we will see, we achieve this by also incorporating proof irrelevance and modal affirmation into our term language.

Polymorphism is a concept that has been explored in the session-type community but usually viewed from the subtyping perspective [31] (while some systems do indeed have ideas close to parametric polymorphism, they are either not applied directly to session types [10] or they do not develop the same results we are able to, given our logical foundation). Here we explore polymorphism in the sense of System F by studying the concurrent interpretation of *second-order linear logic*.

We highlight these two particular extensions to the basic interpretation of [16], although others are possible, ranging from asynchronous process communication [25] to parallel evaluation strategies for λ-calculus terms [76]. The key point we seek to establish is that our framework is not only rich enough to explain a wide range of relevant concurrent phenomena, but also that it enables us to do so in a clean and elegant way, where the technical development is greatly simplified when compared to existing (non-logically based) approaches. Moreover, we can study these phenomena in a *compositional* way insofar as we remain within the boundaries of logic, that is, we can develop these extensions independently from each other, provided we ensure that the underlying logical foundation is sound.

3.1 First-Order Linear Logic and Value-Dependent Session Types

In the context of λ-calculus and the Curry-Howard correspondence, when we move from intuitionistic propositional logic to the first-order setting we obtain a *dependently typed* λ-calculus [40]. While the idea of dependent types can seem superficially simple (types that can depend on terms of the language), the gain in expressive power is known to be far from trivial, providing the ability to write sophisticated theorems as types, for which the proofs are the inhabitants of the type. In this regard, dependent type theories have been studied to great lengths, both for the more theoretical aspects regarding the foundations of mathematics but also for the more pragmatical aspects of correctness by typing that dependent types enable.

In the area of session typed communication, dependent types in the sense described above are virtually non-existant. While some notions of dependent session types exist, they do not carry with them the richness associated with dependent types in the sense of type theory, either because they are designed with a very specific goal in mind (such as [85], where dependencies appear as indices that parameterize session types w.r.t the number of messages exchanged or the number of principals involved in a protocol) or simply because the appropriate logical basis has eluded the research community for a long time.

We consider a form of dependent session types that is closer to the notion of dependent types in functional type theories, where session types may depend on (or be indexed by) terms of a functional type theory such as LF [40] or the Calculus of Constructions [22]. We achieve this by considering the first-order quantifiers of linear logic where the domain of quantification is itself a (functional) dependent type theory. In this sense, it is not a full dependent session type theory since we introduce a separation between the index language and the language of session types, but it already greatly enhances the expressive power of traditional session-typed languages.

3.1.1 Universal and Existential First-Order Quantification

To develop our logically driven concept of value-dependent session types, we consider the sequent calculus rules for (first-order) universal and existential quantification in linear logic, where the domain is quantification is an unspecified typed λ-calculus:

$$\frac{\Psi, x{:}\tau; \Gamma; \Delta \vdash A}{\Psi; \Gamma; \Delta \vdash \forall x{:}\tau.A}\ (\forall\mathsf{R}) \qquad \frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta, A\{M/x\} \vdash C}{\Psi; \Gamma; \Delta, \forall x{:}\tau.A \vdash C}\ (\forall\mathsf{L})$$

To justify the quantification scheme above (besides extending the language of propositions to include the quantifiers), we extend our basic judgment to $\Psi; \Gamma; \Delta \vdash A$, where Ψ is a context region that tracks the term variables introduced by quantification. We also need an additional judgment for the domain of quantification, written $\Psi \vdash M{:}\tau$, which allows us to assert that M witnesses τ for the purposes of providing a witness to quantifiers.

What should then be the concurrent interpretation of universal quantification? Let us consider the principal cut reduction:

$$\dfrac{\dfrac{\Psi, x{:}\tau; \Gamma; \Delta_1 \vdash A}{\Psi; \Gamma; \Delta_1 \vdash \forall x{:}\tau.A}\ (\forall\mathsf{R}) \qquad \dfrac{\Psi \vdash M{:}\tau \quad \Psi; \Gamma; \Delta_2, A\{M/x\} \vdash C}{\Psi; \Gamma; \Delta, \forall x{:}\tau.A \vdash C}\ (\forall\mathsf{L})}{\Gamma; \Delta_1, \Delta_2 \vdash C}\ (\textsc{cut})$$

$$\Longrightarrow \quad \dfrac{\Psi; \Gamma; \Delta_1 \vdash A\{M/x\} \quad \Psi; \Gamma; \Delta_2, A\{M/x\} \vdash C}{\Psi; \Gamma; \Delta_1, \Delta_2 \vdash C}\ (\textsc{cut})$$

We require a substitution principle that combines the judgment $\Psi \vdash M{:}\tau$ with the premise of the $(\forall\mathsf{R})$ rule to obtain the left premise of the reduced cut. From a concurrency perspective, this hints that the term assignment for the right rule should have an input flavor. Compatibly, the left rule must be an output:

$$\dfrac{\Psi, x{:}\tau; \Gamma; \Delta \vdash P :: z{:}A}{\Psi; \Gamma; \Delta \vdash x \leftarrow \mathtt{input}\ z; P :: z{:}\forall x{:}\tau.A}\ (\forall\mathsf{R}) \qquad \dfrac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta, y{:}A\{M/x\} \vdash Q :: z{:}C}{\Psi; \Gamma; \Delta, y{:}\forall x{:}\tau.A \vdash \mathtt{output}\ y\ M; Q :: z{:}C}\ (\forall\mathsf{L})$$

Thus, we identify quantification with communication of terms of a (functional) type theory. Existential quantification is dual. The right rule consists of an output, whereas the left rule is an input:

$$\dfrac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta \vdash P :: z{:}A\{M/x\}}{\Psi; \Gamma; \Delta \vdash \mathtt{output}\ y\ M; P :: z{:}\exists x{:}\tau.A}\ (\exists\mathsf{R}) \qquad \dfrac{\Psi, x{:}\tau; \Gamma; \Delta, y{:}A \vdash Q :: z{:}C}{\Psi; \Gamma; \Delta, y{:}\exists x{:}\tau.A \vdash x \leftarrow \mathtt{input}\ y; P :: z{:}C}\ (\exists\mathsf{L})$$

One crucial concept is that we do not restrict *a priori* what the particular type theory which defines the domain of quantification should be, only requiring it to adhere to basic soundness properties of type safety, and that it remain separate from the linear portion of our framework (although this type theory can itself be linear).

The interested reader may wonder whether or not a type conversion rule is needed in our theory, given that functional terms may appear in linear propositions through quantification. It turns out that this is not the case since the occurrences of functional terms are restricted to the types of quantifiers, and so we may reflect back to type conversion in the functional language as needed. Consider for instance the following session type, describing a session that outputs a natural number and then outputs its successor (assuming a dependent encoding of successors),

$$T \triangleq \exists x{:}\mathsf{nat}.\exists y{:}\mathsf{succ}(x).\mathbf{1}$$

We can produce a session of type $z{:}T$ as follows, appealing to an ambient (dependently-typed) function that given a natural number produces its sucessor:

$$\dfrac{\dfrac{s{:}\Pi n{:}\mathsf{nat}.\mathsf{succ}(n) \vdash s(1+1) : \mathsf{succ}(2) \quad \dfrac{}{s{:}\Pi n{:}\mathsf{nat}.\mathsf{succ}(n); \cdot; \cdot \vdash \mathsf{close}\ z :: z{:}\mathbf{1}}\ (\mathbf{1}\mathsf{R})}{s{:}\Pi n{:}\mathsf{nat}.\mathsf{succ}(n); \cdot; \cdot \vdash \mathsf{output}\ z\ (s(1+1)); \mathsf{close}\ z :: z{:}\exists y{:}\mathsf{succ}(2).\mathbf{1}}\ (\exists\mathsf{R})}{s{:}\Pi n{:}\mathsf{nat}.\mathsf{succ}(n); \cdot; \cdot \vdash \mathsf{output}\ z\ 2; \mathsf{output}\ z\ (s(1+1)); \mathsf{close}\ z :: z{:}T}\ (\exists\mathsf{R})$$

In the typing above we only need type conversion of the functional theory to show,

$$s{:}\Pi n{:}\mathsf{nat}.\mathsf{succ}(n) \vdash s(1+1) : \mathsf{succ}(2)$$

is a valid judgment. No other instances of type conversion are in general required.

To give an SSOS to term-passing we require the ability to evaluate functional terms. The persistent predicate !step $M\,N$ expresses that the functional term M reduces in a single step to term N without using linear resources. The persistent predicate !value N identifies that term N is a value (cannot be reduced further). The evaluation of functional terms is a usual call-by-value semantics. We thus obtain the following SSOS rules:

$$\text{(VRED) } \mathsf{exec}\,(\mathsf{output}\ c\ M; P) \otimes !\mathsf{step}\,M\,N \multimap \{\mathsf{exec}\,(\mathsf{output}\ c\ N; P)\}$$

$$\text{(VCOM) } \mathsf{exec}\,(\mathsf{output}\ c\ V; P) \otimes \mathsf{exec}\,(x \leftarrow \mathsf{input}\ c; Q_x) \otimes !\mathsf{value}\,V \multimap \{\mathsf{exec}\,(P) \otimes \mathsf{exec}\,(Q_V)\}$$

The π-calculus process assignment requires an extension of the calculus with the functional term language, whose terms are then exchanged between processes during communication:

$$\frac{\Psi, x{:}\tau; \Gamma; \Delta \Rightarrow P :: z{:}A}{\Psi; \Gamma; \Delta \Rightarrow z(x).P :: z{:}\forall x{:}\tau.A}\ (\forall\mathsf{R}) \qquad \frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta, y{:}A\{M/x\} \Rightarrow Q :: z{:}C}{\Psi; \Gamma; \Delta, y{:}\forall x{:}\tau.A \Rightarrow y\langle M\rangle.Q :: z{:}C}\ (\forall\mathsf{L})$$

$$\frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta \Rightarrow P :: z{:}A\{M/x\}}{\Psi; \Gamma; \Delta \Rightarrow y\langle M\rangle.P :: z{:}\exists x{:}\tau.A}\ (\exists\mathsf{R}) \qquad \frac{\Psi, x{:}\tau; \Gamma; \Delta, y{:}A \Rightarrow Q :: z{:}C}{\Psi; \Gamma; \Delta, y{:}\exists x{:}\tau.A \Rightarrow y(x).P :: z{:}C}\ (\exists\mathsf{L})$$

The reduction rule is the expected synchronization between term inputs and outputs:

$$x\langle M\rangle.P \mid x(y).Q \longrightarrow P \mid Q\{M/y\}$$

This simple extension to our framework enables us to specify substantially richer types than those found in traditional session type systems since we may now refer to the communicated values in the types, and moreover, if we use a dependent type theory as the term language, we can also effectively state properties of these values and exchange *proof objects* that witness these properties, establishing a high-level framework of concurrent proof-carrying code.

3.2 Example

Consider the following session type, commonly expressible in session typed languages (we write $\tau \supset A$ and $\tau \wedge A$ as shorthand for $\forall x{:}\tau.A$ and $\exists x{:}\tau.A$, where x does not occur in A):

$$\mathsf{Indexer} \triangleq \mathsf{file} \supset (\mathsf{file} \wedge \mathbf{1})$$

The type $\mathsf{Indexer}$ above is intended to specify an indexing service. Clients of this service send it a document and expect to receive back an indexed version of the document. However, the type only specifies the communication behavior of receiving and sending back a file, the indexing portion of the service (its actual functionality) is not captured at all in the type specification.

This effectively means that any service that receives and sends files adheres to the specification of $\mathsf{Indexer}$, which is clearly intended to be more precise. For instance, the process expression $\mathsf{PingPong}$ below, which simply receives and sends back the same file, satisfies the specification (i.e.,$\cdot \vdash \mathsf{PingPong} :: x{:}\mathsf{Indexer}$):

$$\mathsf{PingPong} \triangleq f \leftarrow \mathsf{input}\ x; \mathsf{output}\ x\ f; \mathsf{close}\ x$$

This is especially problematic in a distributed setting, where not all code is available for inspection and so such a general specification is clearly not sufficient.

If we employ dependent session types, we can make the specification more precise by not only specifying that the sent and received files have to be, for instance, PDF files, but we can also specify that the file sent by the indexer has to "agree" with the received one, in the sense that it is, in fact, its indexed version:

$$\mathsf{Indexer}_{dep} \triangleq \forall f{:}\mathsf{file}.\forall p{:}\mathsf{pdf}(f).\exists g{:}\mathsf{file}.\exists q_1{:}\mathsf{pdf}(g).\exists q_2{:}\mathsf{agree}(f,g).\mathbf{1}$$

The revised version of the indexer type $\mathsf{Indexer}_{dep}$ now specifies a service that will receive a file f, a proof object p certifying that f is indeed a pdf, and will then send back a file g, a proof object certifying g as a pdf file and also a proof object that certifies that g and f are in agreement. A process expression I satisfying this specification is given in Fig. 3.1 (we use a let-binding in the expression syntax for conciseness). In the example we make full use of the functional type theory in the quantification domain, where the process expression I makes use of an auxiliary function $\mathsf{genIndex}$ which is itself dependently typed. The type,

$$\mathsf{funIndexer} \triangleq \Pi f{:}\mathsf{file}.\Pi p{:}\mathsf{pdf}(f).\Sigma(g{:}\mathsf{file}, q_1{:}\mathsf{pdf}(g), q_2{:}\mathsf{agree}(f,g))$$

makes use of the dependent product type (the Π-type from Martin-Löf type theory [49]) and of a dependent triple (a curried Σ-type) to specify a function that given a file f and a proof object that encodes the fact that f is a pdf file produces a triple containing a file g, a proof object encoding the fact that g is indeed a pdf file and a proof object q_2 which encodes the fact that g is the indexed version of f.

This much more precise specification provides very strong guarantees to the users of the indexing service, which are not feasibly expressible without this form of (value) dependent session types.

3.3 Metatheoretical Properties

Following the development of Section 2.4, we extend the type preservation and global progress results of our framework to include the value-dependent type constructors.

The logical nature of our framework makes it so that the technical development is fully compositional: extending our previous results to account for the new connectives only requires us to consider the new rules and principles, enabling a clean account of the language features under consideration.

$$
\begin{array}{lcl}
\mathsf{I} & \triangleq & f \leftarrow \mathsf{input}\ x; p \leftarrow \mathsf{input}\ x; \\
 & & \mathsf{let}\ g = \mathsf{genIndex}(f,p)\ \mathsf{in} \\
 & & \mathsf{output}\ x\ (\pi_1\ (g)); \mathsf{output}\ x\ (\pi_2\ (g)); \\
 & & \mathsf{output}\ x\ (\pi_3\ (g)); \mathsf{close}\ x
\end{array}
$$

where

$$
\begin{array}{lcl}
\mathsf{funIndexer} & \triangleq & \Pi f{:}\mathsf{file}.\Pi p{:}\mathsf{pdf}(f).\Sigma(g{:}\mathsf{file}, q_1{:}\mathsf{pdf}(g), q_2{:}\mathsf{agree}(f,g)) \\
\mathsf{Indexer}_{dep} & \triangleq & \forall f{:}\mathsf{file}.\forall p{:}\mathsf{pdf}(f).\exists g{:}\mathsf{file}.\exists q_1{:}\mathsf{pdf}(g).\exists q_2{:}\mathsf{agree}(f,g).\mathbf{1}
\end{array}
$$

with

$$\mathsf{genIndex} : \mathsf{funIndexer}; \cdot; \cdot \vdash \mathsf{I} :: x{:}\mathsf{Indexer}_{dep}$$

Figure 3.1: A PDF Indexer

3.3.1 Correspondence between process expressions and π-calculus processes

Extending the correspondence between process expressions and π-calculus processes is straightforward, provided we enforce a call-by-value communication discipline on processes such that the synchronization between sending and receiving a functional term may only fire after the term has reduced to a value. We thus extend the reduction semantics of Def. 2 with the following rules:

$$\frac{M \to M'}{x\langle M\rangle.P \to x\langle M'\rangle.P} \qquad \frac{V \text{ is a value}}{x\langle V\rangle.P \mid x(y).Q \to P \mid Q\{V/x\}}$$

The translation from process expressions to processes $\rightsquigarrow$ is extended with the expected two clauses, matching the appropriate term communication actions.

$$
\begin{array}{lcl}
P & \rightsquigarrow & \hat{Q} \\
\hline
x \leftarrow \mathsf{input}\ y; P & \rightsquigarrow & y(x).\hat{P} \\
\mathsf{output}\ y\ M; P & \rightsquigarrow & y\langle M\rangle.\hat{P}
\end{array}
$$

Having extended the appropriate definitions accordingly, we can establish the analogues of the theorems of Section 2.4.1. All the proofs are straightforward extensions, taking into account the new typing rules and reduction/execution rules.

Theorem 8 (Static Correspondence - Process Expressions to π-calculus). *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *then there exists a* π*-calculus process* $\hat{Q}$ *such that* $P \rightsquigarrow \hat{Q}$ *with* $\Psi;\Gamma;\Delta \Rightarrow \hat{Q} :: z{:}A$.

Theorem 9 (Static Correspondence - π-calculus to Process Expressions). *If* $\Psi;\Gamma;\Delta \Rightarrow P :: z{:}A$ *then there exists a process expression* $\hat{Q}$ *such that* $P \rightsquigarrow^{-1} \hat{Q}$ *with* $\Psi;\Gamma;\Delta \vdash \hat{Q} :: z{:}A$.

Theorem 10 (Simulation of SSOS Executions). *Let* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$, *with* $\mathsf{exec}\ P\ z\ A \longrightarrow^{1,2} \Omega$. *We have that* $P \rightsquigarrow \hat{P}$, *such that the following holds:*

(i) Either, $\hat{P} \to Q$, *for some* Q, *with* $\Omega \rightsquigarrow\equiv Q$

(ii) Or, $\Omega \rightsquigarrow Q$, *for some* Q, *such that* $\hat{P} \equiv Q$.

Theorem 11 (Simulation of π-calculus Reduction)**.** *Let* $\Psi;\Gamma;\Delta \Rightarrow P :: z{:}A$ *with* $P \rightarrow Q$*. We have that* $P \rightsquigarrow^{-1} \hat{P}$ *such that* $\mathsf{exec}\ \hat{P}\ z\ A \longrightarrow^{+} \Omega$ *with* $\Omega \rightsquigarrow\equiv Q$*.*

3.3.2 Type Preservation

Following the development of Section 2.4.2, we first show a reduction lemma for each of the new type constructors, identifying the synchronization behavior of processes according to their observed actions, under parallel composition. To this end we must also extend the labelled transition semantics (Def. 3) of the π-calculus with the appropriate action labels and transitions for term communication (where V is a value):

$$\alpha ::= \cdots \mid \overline{x\langle V\rangle} \mid x(V)$$

$$\frac{M \rightarrow M'}{x\langle M\rangle.P \xrightarrow{\tau} x\langle M'\rangle.P}\ (\text{TEVAL}) \qquad x\langle V\rangle.P \xrightarrow{\overline{x\langle V\rangle}} P\ (\text{VOUT}) \qquad x(y).P \xrightarrow{x(V)} P\{V/y\}\ (\text{VIN})$$

The proofs for Lemmas 13 and 14 follow along the same principles of the previous reduction lemmas, but they require a lemma accounting for the substitution of (functional) terms in processes. We postulate that type-preserving substitution holds for values in the functional layer outright.

Lemma 12. *If* $\Psi \vdash V{:}\tau$ *and* $\Psi, x : \tau;\Gamma;\Delta \Rightarrow P :: z{:}C$ *then* $\Psi;\Gamma;\Delta \Rightarrow P\{V/x\} :: z{:}C$

Proof. By induction on the structure of typing for P. All cases follow directly from the i.h. or due to weakening with the exception of $\exists$R and $\forall$L, where we appeal to the value substitution postulate of the functional layer, after which the result follows by i.h. □

Lemma 13 (Reduction Lemma - $\forall$)**.** *Assume*

(a) $\Psi;\Gamma;\Delta_1 \Rightarrow P :: x{:}\forall y{:}\tau.A$ *with* $P \xrightarrow{x(V)} P'$ *and* $\Psi \vdash V{:}\tau$

(b) $\Psi;\Gamma;\Delta_2, x{:}\forall y{:}\tau.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{x\langle V\rangle}} Q'$ *and* $\Psi \vdash V{:}\tau$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. See Appendix A.2. □

Lemma 14 (Reduction Lemma - $\exists$)**.** *Assume*

(a) $\Psi;\Gamma;\Delta_1 \Rightarrow P :: x{:}\exists y{:}\tau.A$ *with* $P \xrightarrow{\overline{x\langle V\rangle}} P'$ *and* $\Psi \vdash V{:}\tau$

(b) $\Psi;\Gamma;\Delta_2, x{:}\exists y{:}\tau.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x(V)} Q'$ *and* $\Psi \vdash V{:}\tau$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. See Appendix A.2. □

Before generalizing our type preservation result to account for value dependent session types, we must first extend Lemma 40 with the additional clauses pertaining to the new types (clauses 14-17 below).

Lemma 15. *Let* $\Psi;\Gamma;\Delta \Rightarrow P :: x{:}C$.

1. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = \mathbf{1}$ *then* $\alpha = \overline{x\langle\rangle}$
2. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : \mathbf{1} \in \Delta$ *then* $\alpha = y()$.
3. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \otimes B$ *then* $\alpha = \overline{(\nu y)x\langle y\rangle}$.
4. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \otimes B \in \Delta$ *then* $\alpha = y(z)$.
5. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \multimap B$ *then* $\alpha = x(y)$.
6. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \multimap B \in \Delta$ *then* $\alpha = \overline{(\nu z)y\langle z\rangle}$.
7. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \mathbin{\&} B$ *then* $\alpha = x.\mathsf{inl}$ *or* $\alpha = x.\mathsf{inr}$.
8. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \mathbin{\&} B \in \Delta$ *then* $\alpha = \overline{y.\mathsf{inl}}$ *or* $\alpha = \overline{y.\mathsf{inr}}$.
9. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \oplus B$ *then* $\alpha = \overline{x.\mathsf{inl}}$ *or* $\alpha = \overline{x.\mathsf{inr}}$.
10. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \oplus B \in \Delta$ *then* $\alpha = y.\mathsf{inl}$ *or* $\alpha = y.\mathsf{inr}$.
11. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = {!A}$ *then* $\alpha = \overline{(\boldsymbol{\nu} u)x\langle u\rangle}$.
12. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : {!A} \in \Delta$ *then* $\alpha = y(u)$.
13. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = u$ *and* $u{:}A \in \Gamma$ *then* $\alpha = \overline{(\boldsymbol{\nu} y)u\langle y\rangle}$.
14. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = \forall y{:}\tau.A$ *then* $\alpha = x(V)$.
15. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y{:}\forall z{:}\tau.A \in \Delta$ *then* $\alpha = \overline{y\langle V\rangle}$.
16. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = \exists y{:}\tau.A$ *then* $\alpha = \overline{x\langle V\rangle}$.
17. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y{:}\exists z{:}\tau.A \in \Delta$ *then* $\alpha = y(V)$.

We may now extend our type preservation result accordingly.

Theorem 12 (Type Preservation)**.** *If* $\Psi;\Gamma;\Delta \Rightarrow P :: x{:}A$ *and* $P \rightarrow Q$ *then* $\Psi;\Gamma;\Delta \Rightarrow Q :: x{:}A$

Proof. By induction on typing. As in Theorem 6, when the last rule is an instance of cut, we appeal to the appropriate reduction lemmas for the given cut formula. In this setting we have two new sub-cases, one pertaining to a principal cut where the involved type is $\forall x{:}\tau.B$ and another where the type is $\exists x{:}\tau.B$, each following from Lemmas 13 and 14, respectively. The rest of the proof is as before. □

An important consequence of type preservation in the presence of dependent types is that the property of session fidelity also entails that all properties of exchanged data asserted in types in our framework are always ensured to hold during execution.

3.3.3 Global Progress

Establishing global progress with the two new type constructors turns out to be rather straightforward, given the formal framework setup in Section 2.4.3.

The definition of a live process is unchanged, and thus we need only update the contextual progress and global progress statements to account for the new typing context and reductions of terms in the functional type theory (for which progress is assumed implicitly).

Lemma 16 (Progress - Functional Layer)**.** *If* $\Psi \vdash M{:}\tau$ *then* $M \rightarrow M'$ *or* M *is a value.*

Lemma 17 (Contextual Progress)**.** *Let* $\Psi; \Gamma; \Delta \Rightarrow P :: z{:}C$. *If* $live(P)$ *then there is* Q *such that one of the following holds:*

(a) $P \rightarrow Q$

(b) $P \xrightarrow{\alpha} Q$ *for some* α *where* $s(\alpha) \in z, \Gamma, \Delta$

Proof. The proof is as that of Lemma 11. □

Theorem 13 (Global Progress)**.** *Let* $\cdot; \cdot; \cdot \Rightarrow P :: x{:}\mathbf{1}$. *If* $live(P)$ *then there is* Q *such that* $P \rightarrow Q$.

We have thus established both type preservation and global progress properties in the value dependent session-typed setting. One key observation is that considering new, much more expressive typing schemes does not require a total reworking of the theoretical framework, provided we remain within the realm of the logical interpretation. Rather, re-establishing the type safety results follows naturally as a consequence of the strong logical foundation.

3.4 Value-Dependent Session Types – Proof Irrelevance and Affirmation

While dependent session types provide additional guarantees on the properties of data exchanged during communication, these guarantees come at the price of explicit exchanges of proof objects that certify these properties. For instance, in the type $\mathsf{Indexer}_{dep}$ clients must not only send the file but also a proof object that certifies that it is a valid PDF file. While in the abstract this may be desirable, in practice it can be the case that some proof objects need not be communicated, either because the properties are easily decidable or because of some trust relationship established between the communicating parties. These pragmatic considerations can also fall under the scope of our framework, again by appealing to our solid logical underpinnings.

3.4.1 Proof Irrelevance

To account for the idea of potentially omitting certain proof objects, we require the type theory that makes up the quantification domain to include a *proof irrelevance* modality.

Proof irrelevance [6, 61] is a technique that allows us to selectively hide portions of a proof (and by the proofs-as-programs principle, portions of a program). The intuition is that these “irrelevant” proof objects are required to exist for the purpose of type-checking, but they must have no bearing on the computational outcome of the program. This means that typing must ensure that these hidden proofs are never required to compute something that is not itself hidden. We internalize proof irrelevance in our functional type theory by requiring a modal type constructor, $[\tau]$ (read *bracket* τ), meaning that there is a term of type τ, but the term is deemed irrelevant from a computational point of view.

Operationally, terms of type $[\tau]$ must be present during type-checking, but may consistently be erased at runtime and therefore their communication omitted, since by construction they have no computationally relevant content. For instance, a version of the indexing service that now uses proof irrelevance in order not to require the communication of proofs can be:

$$\mathsf{Indexer}_{[]} \triangleq \forall f{:}\mathsf{file}.\forall p{:}[\mathsf{pdf}(f)].\exists g{:}\mathsf{file}.\exists q_1{:}[\mathsf{pdf}(g)].\exists q_2{:}[\mathsf{agree}(f,g)].\mathbf{1}$$

A service of type $\mathsf{Indexer}_{[]}$ is still required to satisfy the constraints imposed by $\mathsf{Indexer}_{dep}$, and so all the proof objects must exist for type-checking purposes, but they need not be communicated at runtime.

We can give a precise meaning to $[\tau]$ by adding a new introduction form for terms, written $[M]$, meaning that M will not be available computationally. We also add a new class of assumptions $x{\div}\tau$, meaning that x stands for a term of type τ that is not computationally available. Following the style of [61], we define a promotion operation on contexts that transforms computationally irrelevant hypotheses into ordinary ones:

$$\begin{array}{lcl}(\cdot)^{\oplus} & \triangleq & \cdot \\ (\Psi, x{:}\tau)^{\oplus} & \triangleq & \Psi^{\oplus}, x{:}\tau \\ (\Psi, x{\div}\tau)^{\oplus} & \triangleq & \Psi^{\oplus}, x{:}\tau\end{array}$$

We can then define the introduction and elimination forms of proof irrelevant terms:

$$\frac{\Psi^{\oplus} \vdash M{:}\tau}{\Psi \vdash [M]{:}[\tau]}\ ([]\mathrm{I}) \qquad \frac{\Psi \vdash M{:}[\tau] \quad \Psi, x{\div}\tau \vdash N{:}\tau'}{\Psi \vdash \mathbf{let}\ [x] = M\ \mathbf{in}\ N{:}\tau'}\ ([]\mathrm{E})$$

And the appropriate substitution principle for the new hypothesis form:

Theorem 14 (Irrelevant Substitution). *If* $\Psi^{\oplus} \vdash M{:}\tau$ *and* $\Psi, x{\div}\tau, \Psi' \vdash N : \tau'$ *we have that* $\Psi, \Psi' \vdash N\{^{M}/_{x}\}{:}\tau'$.

Proof. By structural induction on the derivation of $\Psi, x{\div}\tau, \Psi' \vdash N : \tau'$. □

These rules guarantee that a variable of the form $x \div \tau$ can only be used in terms that are irrelevant (in the technical sense). In such terms, we are allowed to refer to *all* variables, including the irrelevant ones, since the term is not intended to be available at runtime. Terms of bracket type can still be used through the let-binding shown above, but the bound variable x is tagged with the irrelevant hypothesis form, to maintain the invariant that no relevant term can use irrelevant variables in a computational manner.

While any type that is tagged with the proof irrelevance modality denotes the type of a computationally irrelevant term, the introduction form for $[\tau]$ essentially contains an actual term of type τ, which is then still communicated. We can prevent this by defining an operation † that, given a well-typed process expression, erases all terms of bracket type and replaces them with unit elements. Let the unit type of the functional language be denoted by unit, with the single inhabitant $\langle\rangle$. Since we do not specify the complete type structure of the types of the functional type theory, we will assume, for illustration purposes, that the functional layer has function types $\tau_1 \supset \tau_2$ and pairs $\tau_1 \times \tau_2$. The key aspect is that † preserves the type structure for all types *except* for bracketed types, which are replaced consistently with unit. We thus define † on contexts, types, processes and terms according to the rules of Fig. 3.2, and we can establish the following correctness result.

Theorem 15 (Correctness of Erasure)**.** *If* $\Psi;\Gamma;\Delta \vdash P :: z{:}A$ *then* $\Psi^\dagger;\Gamma^\dagger;\Delta^\dagger \vdash P^\dagger :: z{:}A^\dagger$.

Proof. Straightforward, by induction on the typing derivation. Note that in the case for the let-binding for bracket types we rely on irrelevant substitution (Theorem 14) and on the fact that the variable $[x]$ can only occur in a bracketed term (which is itself replaced by $\langle\rangle$ in †). □

The erasure is designed to be applied conceptually *after* we have ensured that a process expression is well-typed (and therefore abides by whatever specification is defined in its type), but before the code is actually executed. Thus, the erasure is safe because we know that all properties ensured by typing still hold. After performing the erasure †, we can make use of type isomorphisms (Fig. 3.3) to completely erase the communication actions of elements of unit type, thus removing the communication overhead of the proof objects in the original process expression.

At the level of session types we have yet to develop the technical machinery necessary to make precise the notion of a type isomorphism (see Section 7.2). Informally, the isomorphisms of Fig. 3.3 capture the fact that despite the types specifying different behaviors, it is possible to coerce back-and-forth between the types without any "loss of information". It is easy to see why, informally, this is the case: consider the type $x{:}\forall y{:}\mathsf{unit}.A$, denoting a process expression which is meant to receive a value of type unit (the unit element) along x, and continue as A. We can easily construct a process expression that mediates between channel x and the external world through channel $z{:}A$, by first sending the unit element along x and then forwarding between x and z (this is only possible since unit has a single canonical inhabitant). Similarly, given a session of type $x{:}A$ we construct a mediator between x and the external world at channel $z{:}\forall y{:}\mathsf{unit}.A$ by first performing a "dummy" input along z and then forwarding between the two channels. A similar argument can be made for the existentially quantified type.

3.4.2 Affirmation

While we have developed a way of explicitly exchanging and omitting proof objects during execution, it is often the case that what is required in practice lies somewhere between the two extremes of this spectrum: we want to avoid the communication overhead of exchanging proof objects, but do not necessarily want to completely omit them outright, instead demanding some *signed certificate* for the existence of the proof, whose validity should in principle be easier to

$$
\begin{array}{lcl}
(\Delta, x{:}A)^\dagger & \triangleq & \Delta^\dagger, x{:}A^\dagger \\
(\Gamma, x{:}A)^\dagger & \triangleq & \Gamma^\dagger, x{:}A^\dagger \\
(\Psi, x{:}\tau)^\dagger & \triangleq & \Psi^\dagger, x{:}\tau^\dagger \\
(\Psi, x{\div}\tau)^\dagger & \triangleq & \Psi^\dagger
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl}
\mathbf{1}^\dagger & \triangleq & \mathbf{1} & (\forall x{:}\tau.A)^\dagger & \triangleq & \forall x{:}\tau^\dagger.A^\dagger \\
(A \multimap B)^\dagger & \triangleq & A^\dagger \multimap B^\dagger & (\exists x{:}\tau.A)^\dagger & \triangleq & \exists x{:}\tau^\dagger.A^\dagger \\
(A \otimes B)^\dagger & \triangleq & A^\dagger \otimes B^\dagger & (\tau_1 \supset \tau_2)^\dagger & \triangleq & \tau_1^\dagger \supset \tau_2^\dagger \\
(A \oplus B)^\dagger & \triangleq & A^\dagger \oplus B^\dagger & (\tau_1 \times \tau_2)^\dagger & \triangleq & \tau_1^\dagger \times \tau_2^\dagger \\
(A \mathbin{\&} B)^\dagger & \triangleq & A^\dagger \mathbin{\&} B^\dagger & [\tau]^\dagger & \triangleq & \mathsf{unit} \\
(!A)^\dagger & \triangleq & !A^\dagger & \mathsf{unit}^\dagger & \triangleq & \mathsf{unit}
\end{array}
$$

$$
\begin{array}{lcl}
(\mathsf{close}\ z)^\dagger & \triangleq & \mathsf{close}\ z \\
(\mathsf{wait}\ x; Q)^\dagger & \triangleq & \mathsf{wait}\ x; Q^\dagger \\
(\mathsf{new}\ x.(P \parallel Q))^\dagger & \triangleq & \mathsf{new}\ x.(P^\dagger \parallel Q^\dagger) \\
(\mathsf{output}\ z\ (y.P); Q)^\dagger & \triangleq & (\mathsf{output}\ z\ (y.P^\dagger); Q^\dagger) \\
(y \leftarrow \mathsf{input}\ x; R)^\dagger & \triangleq & y \leftarrow \mathsf{input}\ x; R^\dagger \\
(\mathsf{new}\ !u.(x \leftarrow \mathsf{input}\ !u\ P \parallel Q))^\dagger & \triangleq & \mathsf{new}\ !u.(x \leftarrow \mathsf{input}\ !u\ P^\dagger \parallel Q^\dagger) \\
(x.\mathsf{case}(P, Q))^\dagger & \triangleq & x.\mathsf{case}(P^\dagger, Q^\dagger) \\
(x.\mathsf{inl}; P)^\dagger & \triangleq & x.\mathsf{inl}; P^\dagger \\
(x.\mathsf{inr}; P)^\dagger & \triangleq & x.\mathsf{inr}; P^\dagger \\
(x \leftarrow \mathsf{copy}\ !u; R)^\dagger & \triangleq & x \leftarrow \mathsf{copy}\ !u; R^\dagger \\
(\mathsf{output}\ x\ !(y.P))^\dagger & \triangleq & \mathsf{output}\ x\ !(y.P^\dagger) \\
(!u \leftarrow \mathsf{input}\ x; P)^\dagger & \triangleq & !u \leftarrow \mathsf{input}\ x; P^\dagger \\
(\mathsf{output}\ x\ M; P)^\dagger & \triangleq & \mathsf{output}\ x\ M^\dagger; P^\dagger \\
(\mathsf{fwd}\ x\ y)^\dagger & \triangleq & \mathsf{fwd}\ x\ y
\end{array}
$$

$$
\begin{array}{lcl}
[M]^\dagger & \triangleq & \langle\rangle \\
(\lambda x.M)^\dagger & \triangleq & \lambda x.M^\dagger \\
\langle M, N\rangle^\dagger & \triangleq & \langle M^\dagger, N^\dagger\rangle \\
(\mathbf{let}\ [x] = M\ \mathbf{in}\ N)^\dagger & \triangleq & N^\dagger
\end{array}
$$

Figure 3.2: Type-Directed Proof Erasure.

$$
\begin{array}{lcl@{\qquad}lcl}
\forall x{:}\mathsf{unit}.A & \cong & A & \mathsf{unit} \times A & \cong & A \\
\exists x{:}\mathsf{unit}.A & \cong & A & \mathsf{unit} \supset A & \cong & A
\end{array}
$$

Figure 3.3: Type Isomorphisms for Erasure.

check. For example, when we download a large application we may be willing to trust its safety if it is digitally signed by a reputable vendor. On the other hand, if we are downloading and running a piece of Javascript code embedded in a web page, we may insist on some explicit proof that it is safe and adheres to our security policy. The key to making such trade-offs explicit in session types is a notion of *affirmation* (in the sense of [30]) of propositions and proofs by principals. Such affirmations can be realized through explicit digital signatures on proofs by principals, based on some underlying public key infrastructure.

The basic idea is to consider an additional modality in the term language that states that some principal K asserts (or affirms) the existence of a proof of some property τ, written $\Diamond_K \tau$. Terms of this type would in practice be generated by performing a digital signature on the actual proof object that witnesses τ. Formally, we have the new judgment of affirmation [30] by a principal K, written as $\Psi \vdash M :_K \tau$ (K affirms that M has type τ). The rule that defines it is

$$\frac{\Psi \vdash M : \tau}{\Psi \vdash \langle M{:}\tau\rangle_K :_K \tau}\ \mathsf{affirm}$$

The notation $\langle M{:}\tau\rangle_K$ literally refers to a certificate that M has type τ, signed by K. We internalize this judgment in the functional type theory with a new principal-indexed family of modal operators $\Diamond_K \tau$. It is defined by the following two rules:

$$\frac{\Psi \vdash M :_K \tau}{\Psi \vdash M : \Diamond_K \tau}\ \Diamond I \qquad \frac{\Psi \vdash M : \Diamond_K \tau \quad \Psi, x{:}\tau \vdash N :_K \sigma}{\Psi \vdash \mathbf{let}\ \langle x{:}\tau\rangle_K = M\ \mathbf{in}\ N :_K \sigma}\ \Diamond E$$

Such an affirmation may seem redundant: after all, the certificate contains the term which can be type-checked. However, checking a digitally signed certificate may be faster than checking the validity of a proof, so we may speed up the system if we trust K's signature. More importantly, if we have proof irrelevance, and some parts of the proof have been erased, then we have in general no way to reconstruct the original proofs. In this case we must trust the signing principal K to accept the τ as true, because we cannot be sure if K played by the rules and did indeed have a proof. Therefore, in general, the affirmation of τ by K is *weaker* than the truth of τ, for which we demand explicit evidence. Conversely, when τ is true K can always sign it and be considered as "playing by the rules".

We can now add these signed certificates to our running example:

$$\mathsf{Indexer}_\Diamond \triangleq \forall f{:}\mathsf{file}.\forall p{:}[\mathsf{pdf}(f)].\exists g{:}\mathsf{file}.\exists q_1{:}[\mathsf{pdf}(g)].\exists q_2{:}\Diamond_I[\mathsf{agree}(f,g)].\mathbf{1}$$

While we still do not require the communication of the proof objects that assert the files are actually PDFs, the type $\mathsf{Indexer}_\Diamond$ now specifies that a digitally signed (by principal I) certificate of the agreement of the sent and received files is communicated back to the client of the service. The combination of both proof irrelevance and affirmation is crucial: the type $\Diamond_I[\mathsf{agree}(f,g)]$ denotes a proof relevant object q_2 which is an affirmation (and thus digitally signed) by principal I of a *proof irrelevant* object, asserting the agreement of f and g. Thus, the object q_2 may not contain the actual proof object and is in principle substantially smaller and easier to check.

This once again showcases the strengths of having a robust logical foundation, which not only allows us to move from session types to *value dependent* session types in a relatively straightforward way, while preserving the desirable properties of session fidelity (which in the dependent case also entails the validity of the properties expressed in the dependencies) and deadlock freedom, but also enables us to incorporate known logical concepts such as proof irrelevance and affirmation in our concurrent, session-typed setting.

These extensions provide a clean account of high-level concepts such as proof-carrying code with various refined levels of trust: we may not trust the used services at all, and demand that proofs of all the properties be communicated; we may trust the services completely (or check that the properties hold ourselves) by omitting proofs at runtime; or we may appeal to some trusted third party to digitally sign objects that certify the existence of proof objects witnessing the desired properties.

3.5 Second-Order Linear Logic and Polymorphic Session Types

When considering concurrent message-passing systems, it is rather straightforward to consider polymorphism at the data level, that is, systems that admit polymorphism on the data values exchanged in communication. It is less common to consider polymorphism at the behavioral level (in the sense of communication protocols), or *behavioral genericity*. This form of genericity is particularly important since it allows systems to be generic with respect to arbitrary communication protocols, which may be known and instantiated only at runtime. For instance, critical web applications often involve dynamic reconfiguration of communication interfaces/protocols (e.g. replacing a service provider), which should be transparent to clients. To this end, these applications should be conceived as generic with respect to such interfaces/protocols. Another common scenario is that of "cloud-based" services, which acquire computing resources when demand is high and release them as they are no longer needed. These scaling policies require the services to be generic with respect to their underlying coordination protocols, which may depend on the system's architecture at a given time.

In the realm of concurrent processes, polymorphism was first studied by Turner [78], in the context of a simply-typed π-calculus. Berger et al. [10] were the first to study a π-calculus with parametric polymorphism based on universal and existential quantification over types. However, in the context of session types, polymorphism has mainly been developed as *bounded polymorphism* [31, 33], which is controlled via subtyping. This form of polymorphism is sufficient to capture data-level genericity but not behavioral genericity. True parametric polymorphism for session types in the sense of Reynolds was proposed by Wadler [81] for a language based on classical linear logic, but without the associated reasoning techniques that arise by the study of parametricity which is done in Part III.

By making use of our linear logic foundation for message-passing concurrency, we can give a rather clean account of polymorphism in the sense of behavioral genericity by considering a concurrent interpretation of *second-order* linear logic, exploring a correspondence along the lines of that between the polymorphic λ-calculus [35, 64] and second-order intuitionistic logic.

Our formulation of second-order linear logic uses a specialized context region for type variables introduced by second-order quantification (which we write Ω) and an additional judgment $\Omega \vdash$

A type stating that A is a well-formed type (or proposition) with free variables registered in Ω. Thus the main judgment is now $\Omega; \Gamma; \Delta \vdash A$. The rules that define the well-formedness judgment are straightforward and thus omitted. The rules for (second-order) universal and existential quantification are:

$$\frac{\Omega, X; \Gamma; \Delta \vdash A}{\Omega; \Gamma; \Delta \vdash \forall X.A}\ (\forall 2\mathsf{R}) \qquad \frac{\Omega \vdash B \ \mathsf{type} \quad \Omega; \Gamma; \Delta, A\{B/X\} \vdash C}{\Omega; \Gamma; \Delta, \forall X.A \vdash C}\ (\forall 2\mathsf{L})$$

$$\frac{\Omega \vdash B \ \mathsf{type} \quad \Omega; \Gamma; \Delta \vdash A\{B/X\}}{\Omega; \Gamma; \Delta \vdash \exists X.A}\ (\exists 2\mathsf{R}) \qquad \frac{\Omega, X; \Gamma; \Delta, A \vdash C}{\Omega; \Gamma; \Delta, \exists X.A \vdash C}\ (\exists 2\mathsf{L})$$

Proving a universally quantified proposition $\forall X.A$ simply requires us to prove the proposition A without assuming any specifics on the type variable, whereas using a universally quantified proposition requires us to form a valid proposition B which then warrants the use of $A\{B/X\}$. The existential is dual. We note that, since we impose only type well-formedness in quantifier witnesses, the form of quantification implemented by these rules is *impredicative* (i.e.,the type variable X in $\forall X.A$ may be instantiated with $\forall X.A$).

Much like in the concurrent interpretation of the first-order quantifiers of Section 3.1, the second-order quantifiers also have an input/output flavor. However, instead of exchanging data values, we exchange *session types*. This essentially means that the process expression assignment (and π-calculus assignment) for second-order quantification enables an expressive form of *abstract protocol communication*.

$$\frac{\Omega, X; \Gamma; \Delta \vdash P :: z{:}A}{\Omega; \Gamma; \Delta \vdash X \leftarrow \mathtt{input}\ z; P :: z{:}\forall X.A}\ (\forall 2\mathsf{R}) \qquad \overset{(\forall 2\mathsf{L})}{\frac{\Omega \vdash B \ \mathsf{type} \quad \Omega; \Gamma; \Delta, x{:}A\{B/X\} \vdash Q :: z{:}C}{\Omega; \Gamma; \Delta, x{:}\forall X.A \vdash \mathtt{output}\ x\ B; Q :: z{:}C}}$$

$$\frac{\Omega \vdash B \ \mathsf{type} \quad \Omega; \Gamma; \Delta \vdash P :: z{:}A\{B/X\}}{\Omega; \Gamma; \Delta \vdash \mathtt{output}\ z\ B; P :: z{:}\exists X.A}\ (\exists 2\mathsf{R}) \qquad \overset{(\exists 2\mathsf{L})}{\frac{\Omega, X; \Gamma; \Delta, x{:}A \vdash Q :: z{:}C}{\Omega; \Gamma; \Delta, x{:}\exists X.A \vdash X \leftarrow \mathtt{input}\ x; Q :: z{:}C}}$$

As before, the intuitive semantics of the assignment are justified by the following cut reduction:

$$\frac{\dfrac{\Omega, X; \Gamma; \Delta \vdash P :: x{:}A}{\Omega; \Gamma; \Delta \vdash X \leftarrow \mathtt{input}\ z; P :: x{:}\forall X.A}\ (\forall 2\mathsf{R}) \qquad \dfrac{\Omega \vdash B \ \mathsf{type} \quad \Omega; \Gamma; \Delta, x{:}A\{B/X\} \vdash Q :: z{:}C}{\Omega; \Gamma; \Delta, x{:}\forall X.A \vdash \mathtt{output}\ x\ B; Q :: z{:}C}\ (\forall 2\mathsf{L})}{\Omega; \Gamma; \Delta \vdash \mathsf{new}\ x.((X \leftarrow \mathtt{input}\ x; P) \parallel (\mathtt{output}\ x\ B; Q)) :: z{:}C}\ (\textsc{cut})$$

$$\Longrightarrow \quad \frac{\Omega; \Gamma; \Delta \vdash P\{B/X\} :: x{:}A\{B/X\} \quad \Omega; \Gamma; \Delta, x{:}A\{B/X\} \vdash Q :: z{:}C}{\Omega; \Gamma; \Delta \vdash \mathsf{new}\ x.(\leftarrow P\{B/X\} \parallel Q) :: z{:}C}\ (\textsc{cut})$$

Which results in the following SSOS rule:

$$(\textsc{tcom})\ \mathsf{exec}\,(\mathsf{output}\ c\ B; P) \otimes \mathsf{exec}\,(X \leftarrow \mathsf{input}\ c; Q_X) \multimap \{\mathsf{exec}\,(P) \otimes \mathsf{exec}\,(Q_B)\}$$

The π-calculus assignment mimics the process expression assignment, requiring an extension of the π-calculus with the ability to communicate session types:

$$\frac{\Omega, X; \Gamma; \Delta \Rightarrow P :: z{:}A}{\Omega; \Gamma; \Delta \Rightarrow z(X).P :: z{:}\forall X.A}\ (\forall 2\mathsf{R}) \qquad \frac{\Omega \vdash B\ \mathsf{type} \quad \Omega; \Gamma; \Delta, x{:}A\{B/X\} \Rightarrow Q :: z{:}C}{\Omega; \Gamma; \Delta, x{:}\forall X.A \Rightarrow x\langle B\rangle.Q :: z{:}C}\ (\forall 2\mathsf{L})$$

$$\frac{\Omega \vdash B\ \mathsf{type} \quad \Omega; \Gamma; \Delta \Rightarrow P :: z{:}A\{B/X\}}{\Omega; \Gamma; \Delta \Rightarrow z\langle B\rangle.P :: z{:}\exists X.A}\ (\exists 2\mathsf{R}) \qquad \frac{\Omega, X; \Gamma; \Delta, x{:}A \Rightarrow Q :: z{:}C}{\Omega; \Gamma; \Delta, x{:}\exists X.A \Rightarrow x(X).Q :: z{:}C}\ (\exists 2\mathsf{L})$$

3.5.1 Example

Consider the following session type:

$$\mathsf{CloudServer} \triangleq \forall X.!(\mathsf{api} \multimap X) \multimap !X$$

The CloudServer type represents a simple interface for a *cloud-based application server*. In our theory, this is the session type of a system which first *inputs* an arbitrary type (say GMaps); then inputs a shared service of type api $\multimap$ GMaps. Each instance of this service yields a session that when provided with the implementation of an API will provide a behavior of type GMaps; finally becoming a persistent (shared) server of type GMaps. Our application server is meant to interact with developers that, by building upon the services it offers, implement their own applications. In our framework, the dependency between the cloud server and applications may be expressed by the typing judgment

$$\cdot\,;\ x{:}\mathsf{CloudServer} \vdash \mathit{DrpBox} :: z{:}\mathsf{dbox}$$

Intuitively, the judgment above states that to offer behavior dbox on z, the file hosting service represented by process *DrpBox* relies on a linear behavior described by type CloudServer provided on x (no persistent behaviors are required). The crucial role of behavioral genericity should be clear from the following observation: to support interaction with developers such as *DrpBox*—which implement all kinds of behaviors, such as dbox above—any process realizing type CloudServer should necessarily be *generic* on such expected behaviors, which is precisely what we accomplish here through our logical foundation.

A natural question to ask is whether or not a theory of parametricity as that for the polymorphic λ-calculus exists in our setting, given our development of impredicative polymorphism in a way related to that present in the work of Reynolds [65]. We answer positively to this question in Part III, developing a theory of parametricity for polymorphic session types as well as the natural notion of equivalence that arises from parametricity (Section 6.2.2), showing it coincides with the familiar process calculus notion of (typed) *barbed congruence* in Chapter 7.

3.6 Metatheoretical Properties

As was the case for our study of first-order linear logic as a (value) dependent session-typed language, we revisit the analysis of the correspondence between process expressions and the π-calculus, type preservation and global progress, now in the presence of polymorphic sessions.

3.6.1 Correspondence between process expressions and π-calculus processes

As before, we must define a translation between our polymorphic process expressions and a polymorphic π-calculus (i.e.,a π-calculus with type input and output prefixes). The reduction semantics of this polymorphic π-calculus consist of those in Def. 2, extended with the following reduction rule (where A is a session type):

$$z\langle A\rangle.P \mid z(X).Q \to P \mid Q\{A/X\}$$

The labelled transition semantics (Def. 3) are extended accordingly, with the type output and input action labels and the respective transitions:

$$\alpha ::= \cdots \mid \overline{x\langle A\rangle} \mid x(A)$$

$$x\langle A\rangle.P \xrightarrow{\overline{x\langle A\rangle}} P \text{ (TOUT)} \quad x(Y).P \xrightarrow{x(A)} P\{A/Y\} \text{ (TIN)}$$

The translation from process expressions to processes $\rightsquigarrow$ is extended accordingly:

P	$\rightsquigarrow$	$\hat{Q}$
$X \leftarrow \mathsf{input}\ y; P$	$\rightsquigarrow$	$y(X).\hat{P}$
$\mathsf{output}\ y\ A; P$	$\rightsquigarrow$	$y\langle A\rangle.\hat{P}$

Again, since we have a strict one-to-one correspondence between the π-calculus reduction rule and the SSOS rule, revisiting the static and dynamic correspondence results is straightforward (we write Θ for process expression execution states to avoid confusion with the new typing context Ω).

Theorem 16 (Static Correspondence - Process Expressions to π-calculus)**.** *If* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$ *then there exists a π-calculus process $\hat{Q}$ such that $P \rightsquigarrow \hat{Q}$ with $\Omega; \Gamma; \Delta \Rightarrow \hat{Q} :: z{:}A$.*

Theorem 17 (Static Correspondence - π-calculus to Process Expressions)**.** *If* $\Omega; \Gamma; \Delta \Rightarrow P :: z{:}A$ *then there exists a process expression $\hat{Q}$ such that $P \rightsquigarrow^{-1} \hat{Q}$ with $\Omega; \Gamma; \Delta \vdash \hat{Q} :: z{:}A$.*

Theorem 18 (Simulation of SSOS Executions)**.** *Let* $\Omega; \Gamma; \Delta \vdash P :: z{:}A$, *with* $\mathsf{exec}\ P\ z\ A \longrightarrow^{1,2} \Theta$. *We have that $P \rightsquigarrow \hat{P}$, such that the following holds:*

(i) Either, $\hat{P} \to Q$, for some Q, with $\Theta \rightsquigarrow\equiv Q$

(ii) Or, $\Theta \rightsquigarrow Q$, for some Q, such that $\hat{P} \equiv Q$.

Theorem 19 (Simulation of π-calculus Reduction)**.** *Let* $\Omega; \Gamma; \Delta \Rightarrow P :: z{:}A$ *with* $P \to Q$. *We have that* $P \rightsquigarrow^{-1} \hat{P}$ *such that* $\mathsf{exec}\ \hat{P}\ z\ A \longrightarrow^{+} \Theta$ *with* $\Theta \rightsquigarrow\equiv Q$.

3.6.2 Type Preservation

As before, we first establish a reduction lemma for the polymorphic universal and existential types, with the caveat that the type in the action labels is well-formed in the appropriate context.

Lemma 18 (Reduction Lemma - $\forall 2$)**.** *Assume* *(a)* $\Omega;\Gamma;\Delta_1 \Rightarrow P :: x{:}\forall X.A$ *with* $P \xrightarrow{x(B)} P'$ *and* $\Omega \vdash B\ \mathsf{type}$

(b) $\Omega;\Gamma;\Delta_2, x{:}\forall X.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{x\langle B\rangle}} Q'$ *and* $\Omega \vdash B\ \mathsf{type}$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Omega;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. See Appendix A.3. □

Lemma 19 (Reduction Lemma - $\exists 2$)**.** *Assume*

(a) $\Omega;\Gamma;\Delta_1 \Rightarrow P :: x{:}\exists X.A$ *with* $P \xrightarrow{\overline{x\langle B\rangle}} P'$ *and* $\Omega \vdash B\ \mathsf{type}$

(b) $\Omega;\Gamma;\Delta_2, x{:}\exists X.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x(B)} Q'$ *and* $\Omega \vdash B\ \mathsf{type}$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Omega;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. See Appendix A.3. □

Following the lines of the reformulation of Lemma 40 to Lemma 15, we can characterize the action labels as a result of the polymorphic typing. For the sake of not being overly redundant, we omit the reformulation of this lemma, noting that offering a polymorphic universal entails a type input action and using a polymorphic universal entails a type output action (the dual holds for the polymorphic existential). We can now extend the type preservation result as before.

Theorem 20 (Type Preservation)**.** *If* $\Omega;\Gamma;\Delta \Rightarrow P :: x{:}A$ *and* $P \rightarrow Q$ *then* $\Omega;\Gamma;\Delta \Rightarrow Q :: x{:}A$

Proof. By induction on typing. As in Theorem 6, when the last rule is an instance of cut, we appeal to the appropriate reduction lemmas for the given cut formula. In this setting we have two new sub-cases, one pertaining to a principal cut where the involved type is $\forall X.B$ and another where the type is $\exists X.B$, each following from Lemmas 18 and 19, respectively, as in Theorem 12. The remainder of the proof follows as before. □

3.6.3 Global Progress

Unsurprisingly, our framework is robust enough to account for progress in our new polymorphic session-typed setting. The technical development follows the same pattern as before: we first establish a contextual progress lemma that characterizes precisely the behavior of typed, live processes, from which our global progress result follows.

Lemma 20 (Contextual Progress)**.** *Let* $\Omega;\Gamma;\Delta \Rightarrow P :: z{:}C$. *If* $live(P)$ *then there is* Q *such that one of the following holds:*

(a) $P \rightarrow Q$

(b) $P \xrightarrow{\alpha} Q$ *for some* α *where* $s(\alpha) \in z, \Gamma, \Delta$

Proof. The proof is as that of Lemma 11. □

Theorem 21 (Global Progress)**.** *Let* $\cdot; \cdot; \cdot \Rightarrow P :: x{:}\mathbf{1}$. *If* $live(P)$ *then there is* Q *such that* $P \rightarrow Q$.

3.7 Summary and Further Discussion

In this chapter we have further justified our use of linear logic as a logical foundation for message-passing concurrent computation by going beyond propositional linear logic (Chapter 2) and studying concurrent interpretations of both first-order and second-order intuitionistic linear logic as (value) dependent session types and polymorphic session types, respectively, as a testament of the flexibility of our logically grounded approach. We have also considered extensions of the dependently-typed framework with proof irrelevance and modal affirmation to account for notions of trust and (abstract) digital proof certificates, in order to further solidify the modularity of our logical foundation.

Our particular choice of first and second-order linear logic arose from an attempt to mimic the Curry-Howard correspondence between typed λ-calculi and intuitionistic logic.

For the first-order case, we have opted for a stratification between the concurrent layer (linear logic) and the type indices, thus providing a form of value dependencies which express rich properties of exchanged data at the interface level, an important extension when considering concurrent and distributed settings.

In the second-order setting, our development mimics rather closely that of the polymorphic λ-calculus, except we replace type abstraction with session-type (or protocol) *communication*. It is often the case in polymorphic λ-calculi that the explicit representation of types in the syntax is avoided as much as possible for the sake of convenience. We opt for an opposite approach given our concurrent setting: we make explicit the behavioral type abstractions by having process expressions exchange these behavioral descriptions as first-order values, which is not only reasonable but realistic when we consider scenarios of logically and physically distributed agents needing to agree on the behavioral protocols they are meant to subsequently adhere to.

Part II

Towards a Concurrent Programming Language

Our development up to this point focused on providing logically motivated accounts of relevant concurrent phenomena. Here we take a different approach and exploit its ability to express concurrent computation to develop a simple yet powerful session-based concurrent programming language based on the process expression assignment developed in the previous sections, encapsulating process expressions in a *contextual monad* embedded in a λ-calculus.

Clearly this is not the first concurrent programming language with session types. Many such languages have been proposed over the years [12, 32, 38, 43]. The key aspect that is lacking from such languages is a true logical foundation, with the exception of Wadler's GV [81]. This makes the metatheory of such languages substantially more intricate, and often the properties obtained "for free" due to logic cannot easily be replicated in their setting. The key difference between the language developed here and that of [81] is that we combine functions and concurrency through monadic encapsulation, whereas GV provides no such separation. This entails that the entirety of GV is itself linear, whereas the language developed here is not. Another significant difference between the two approaches is that the underlying type theory of GV is classical, whereas ours is intuitionistic. Monads are intuitionistic in their logical form [29], which therefore makes the intuitionistic form of linear logic a particularly good candidate for a monadic integration of functional and concurrent computation based on a Curry-Howard correspondence. We believe our natural examples demonstrate this clearly.

We note that our language is mostly concerned with the *concurrent* nature of computation, rather than the issues that arise when considering *distribution* of actual executing code. We believe it should be possible to give a precise account of both concurrency and explicit distribution of running code by exploring ideas of hybrid or modal logic in the context of our linear framework, similar to that of [54], where worlds in the sense of modal logic are considered as sites where computation may take place. One of the key aspects of our development that seems to align itself with this idea is the identification of running processes with the channel along which they offer their intended behavior, which provide a delimited running object that may in principle be located in one of these sites.

CHAPTER 4

CONTEXTUAL MONADIC ENCAPSULATION OF CONCURRENCY

The process expression assignment developed throughout this document enables us to write expressive concurrent behavior, but raises some issues when viewed as a proper programming language. In particular, it is not immediately obvious how to fully and uniformly incorporate the system into a complete functional calculus to support higher-order, message-passing concurrent computation. Moreover, its not clear how significantly our session-based (typed) communication restricts the kinds of programs we are allowed to write, or how easy it is to fully combine functional and concurrent computation while preserving the ability to reason about programs in the two paradigms.

To address this challenge we use a *contextual monad* to encapsulate open concurrent computations, which can be passed as arguments in functional computation but also *communicated* between processes in the style of *higher-order processes*, providing a uniform integration of both higher-order functions and concurrent computation. For expressiveness and practicality, we allow the construction of recursive process expressions, which is a common motif in realistic applications.

The key idea that motivates the use of our contextual monad is that we can identify an open process expression as a sort of “black box”: if we compose it with process expressions implementing the sessions required by the open process expression, it will provide a session channel along which it implements its specified behavior. If we attempt to simply integrate process expressions and functional terms outright, it is unclear how to treat session channels combined with ordinary functional variables. One potential solution is to map channels to ordinary functional variables, forcing the entire language to be linear, which is a significant departure from typical approaches. Moreover, even if linear variables are supported, it is unclear how to restrict their occurrences so they are properly localized with respect to the structure of running processes. Thus, our solution consists of isolating process expressions from functional terms in such a way that each process expression is bundled with all channels (both linear and shared) it uses and the one that it offers.

$$\begin{array}{lcll}
\tau, \sigma & ::= & \tau \rightarrow \sigma \mid \ldots \mid \forall t.\,\tau \mid \mu t.\,\tau \mid t & \text{(ordinary functional types)} \\
 & \mid & \{\overline{a_i{:}A_i} \vdash a{:}A\} & \text{process offering } A \text{ along channel } a, \\
 & & & \text{using channels } a_i \text{ offering } A_i
\end{array}$$

$$\begin{array}{lcll}
A, B, C & ::= & \tau \supset A & \text{input value of type } \tau \text{ and continue as } A \\
 & \mid & \tau \wedge A & \text{output value of type } \tau \text{ and continue as } A \\
 & \mid & A \multimap B & \text{input channel of type } A \text{ and continue as } B \\
 & \mid & A \otimes B & \text{output fresh channel of type } A \text{ and continue as } B \\
 & \mid & \mathbf{1} & \text{terminate} \\
 & \mid & \&\{\overline{l_j{:}A_j}\} & \text{offer choice between } l_j \text{ and continue as } A_j \\
 & \mid & \oplus\{\overline{l_j{:}A_j}\} & \text{provide one of the } l_j \text{ and continue as } A_j \\
 & \mid & !A & \text{provide replicable service } A \\
 & \mid & \nu X.A \mid X & \text{recursive session type}
\end{array}$$

Figure 4.1: The Syntax of Types

4.1 Typing and Language Syntax

The type structure of the language is given in Fig. 4.1, separating the types from the functional language, written τ, σ from session types, written A, B, C. The most significant type is the contextual monadic type $\{\overline{a_i{:}A_i} \vdash a{:}A\}$, which types monadic values that encapsulate session-based concurrency in the λ-calculus. The idea is that a value of type $\{\overline{a_i{:}A_i} \vdash a{:}A\}$ denotes a process expression that offers the session A along channel a, when provided with sessions $a_1{:}A_1$ through $a_n{:}A_n$ (we write $\{a{:}A\}$ when the context regions are empty). The language of session types is basically the one presented in the previous sections of this document, with the ability to send and receive values of functional type (essentially a non-dependent version of $\forall$ and $\exists$ from Section 3.1), with n-ary labelled versions of the choice and selection types and with recursive session types. One subtle point is that since monadic terms are seen as opaque functional values, they can be sent and received by process expressions, resulting in a setting that is reminiscent of higher-order processes in the sense of [68].

Our language appeals to two typing judgments, one for functional terms which we write $\Psi \Vdash M{:}\tau$, where M is a functional term and Ψ a context of functional variables; and another for process expressions $\Psi; \Gamma; \Delta \vdash P :: z{:}A$, as before. For convenience of notation, we use $\Delta = \mathsf{lin}(\overline{a_i{:}A_i})$ to denote that the context Δ consists of the linear channels $c_i{:}A_i$ in $\overline{a_i{:}A_i}$; and $\Gamma = \mathsf{shd}(\overline{a_i{:}A_i})$ to state that Γ consists of the unrestricted channels $!u_i{:}A_i$ in $\overline{a_i{:}A_i}$.

To construct a value of monadic type, we introduce a monadic constructor, typed as follows:

$$\frac{\Delta = \mathsf{lin}(a_i{:}A_i) \quad \Gamma = \mathsf{shd}(a_i{:}A_i) \quad \Psi; \Gamma; \Delta \vdash P :: c{:}A}{\Psi \Vdash c \leftarrow \{P_{c,\overline{a_i}}\} \leftarrow \overline{a_i{:}A_i} : \{\overline{a_i{:}A_i} \vdash c{:}A\}}\ \{\,\}I$$

We thus embed process expressions within the functional language. A term of monadic type consists of a process expression, specifying a channel name along which it offers its session and those it requires to be able to offer said session. Typing a monadic value requires that the underlying process expression be well-typed according to this specification.

To use a value of monadic type, we extend the process expression language with a form of contextual monadic composition (in Haskell terminology, with a monadic bind):

$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi \Vdash M : \{\overline{a_i{:}A_i} \vdash c{:}A\} \quad \Psi; \Gamma; \Delta', c{:}A \vdash Q_c :: d{:}D}{\Psi; \Gamma; \Delta, \Delta' \vdash c \leftarrow M \leftarrow \overline{a_i}; Q_c :: d{:}D}\ \{\,\}E$$

Thus, only process expressions may actually "unwrap" monadic values, which are opaque values from the perspective of the functional language. In a monadic composition, the monadic term M is provided with the channels $\overline{a_i}$ required to run the underlying process expression. When the evaluation of M produces the actual monadic value, it is composed in parallel with the process expression Q, sharing a fresh session channel along which the two processes may interact.

The SSOS rule for monadic composition is:

$$\mathsf{exec}\,(c \leftarrow M \leftarrow \overline{a_i}; Q_c) \otimes !\mathsf{eval}\, M\, (c \leftarrow \{P_{c,\overline{a_i}}\} \leftarrow \overline{a_i}) \multimap \{\exists c'.\, \mathsf{exec}\,(P_{c',\overline{a_i}}) \otimes \mathsf{exec}\,(Q_{c'})\}$$

Executing a monadic composition evaluates M to a value of the appropriate form, which must contain a process expression P. We then create a fresh channel c' and execute $P_{c',\overline{a_i}}$ itself, in parallel with $Q_{c'}$. In the underlying value of M, the channels c and $\overline{a_i}$ are all bound names, so we rename them implicitly to match the interface of M in the monadic composition.

One interesting aspect of monadic composition is that it subsumes the process composition constructs presented in the previous sections (proof-theoretically this is justified by the fact that monadic composition effectively reduces to a cut). Given this fact, in our language we omit the previous process composition constructs and enable composition only through the monad.

On the Monadic Nature of the Monad We have dubbed our construction that embedding session-based concurrency in a functional calculus a contextual *monad*. In general, a monad consists of a type constructor supporting two operations usually dubbed *return* and *bind*. The return operation allows for terms of the language to be made into monadic objects. The bind operation is a form of composition, which unwraps a monadic object and injects its underlying value into another. These operations are expected to satisfy certain natural equational laws: return is both a left and right unit for bind, with the latter also being associative.

However, from a category-theoretic perspective, monads are endomorphic constructions so it is not immediately obvious that our construct, bridging two different languages[1] does indeed form a monad. While a proper categorical analysis of our construct is beyond the scope of this work, we point the interested reader to the work of Altenkirch et al. [5] on so-called relative monads, which are monadic constructions for functors that are not endomorphic, and to the work of Benton [9] which defines a pair of functors which form an adjunction between intuitionistic and linear calculi.

Informally, we argue for the monadic-like structure and behavior of our construction (as an endomorphism) by considering the type $\{c{:}\tau \wedge \mathbf{1}\}$; a $\mathtt{return}$ function of type $\tau \rightarrow \{c{:}\tau \wedge \mathbf{1}\}$; which given an M element of type τ constructs a process expression that outputs M and terminates;

[1] It is known that simply-typed λ-calculus with $\beta\eta$-equivalence is the internal language of cartesian closed categories, whereas linear typing disciplines of λ-calculus are associated with closed symmetric monoidal categories.

and a bind function of type $\{c{:}\tau \wedge \mathbf{1}\} \rightarrow (\tau \rightarrow \{d{:}\sigma \wedge \mathbf{1}\}) \rightarrow \{d{:}\sigma \wedge \mathbf{1}\}$, which given a monadic object $P : \{c{:}\tau \wedge \mathbf{1}\}$ and a function $f : \tau \rightarrow \{d{:}\sigma \wedge \mathbf{1}\}$ produces a process expression that first executes P, inputting from it a value M of type τ, enabling the execution of f applied to M, which we then forward to d satisfying the type of bind.

It is easy to informally see that these two functions satisfy the monad laws: $\mathsf{bind}\ (\mathsf{return}\ V)\ f$ results in a process expression that is observably identical to $f\ V$, since that is precisely what bind does, modulo some internal steps (left unit); $\mathsf{bind}\ P\ \mathsf{return}$ results in a process expression that behaves identically to P, given the resulting process will run P, receive from it some value τ which is then passed to the return function, constructing a process expression that behaves as P (right unit). Associativity of bind is straightforward.

Process Expression Constructors The remainder of the constructs of the language that pertain to the concurrent layer are those of our process expression language, with a few minor conveniences. Value input and output uses the same constructs for the first-order quantifiers of Section 3.1, but without type dependencies. The choice and selection constructs are generalized to their n-ary labelled version, and so we have:

$$\frac{\Psi;\Gamma;\Delta \vdash P_1 :: c{:}A_1 \quad \ldots \quad \Psi;\Gamma;\Delta \vdash P_k :: c{:}A_k}{\Psi;\Gamma;\Delta \vdash \mathtt{case}\ c\ \mathtt{of}\ \overline{l_j \Rightarrow P_j} :: c{:}\ \&\ \{\overline{l_j : A_j}\}}\ \&\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta, c{:}A_j \vdash P :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\ \&\ \{\overline{l_j : A_j}\} \vdash c.l_j; P :: d{:}D}\ \&\mathsf{L}$$

The rules for $\oplus$ are dual, using the same syntax.

The full typing rules for process expressions and the contextual monad are given in Fig. 4.2. Given that monadic composition subsumes direct process expression composition, we distinguish between linear and shared composition through rules $\{\}E$ and $\{\}E^!$. The latter, reminiscent of a persistent cut, requires that the monadic expression only make use of shared channels and offer a shared channel $!u$, which is used in Q accordingly (via copy). The operational semantics for this form of composition will execute the underlying process expression by placing an input-guarded replication prefix, identical to the operational behavior of $\mathsf{cut}^!$ of the previous sections. The remainder of the constructs of the language follow precisely the typing schemas we have presented earlier. We highlight the rules for value communication (those pertaining to $\wedge$ and $\supset$) which mimic the rules for $\forall$ and $\exists$ of Section 3.1 but without the type dependencies.

4.2 Recursion

The reader may then wonder how can recursive session types be inhabited if we do not extend the process expression language with a recursor or the typical fold and unfold constructs. Indeed, our process expression language does not have a way to internally define recursive behavior. Instead, we enable the definition of recursive process expressions through recursion in the functional language

$$\frac{\Delta = \mathsf{lin}(a_i{:}A_i) \quad \Gamma = \mathsf{shd}(a_i{:}A_i) \quad \Psi;\Gamma;\Delta \vdash P :: c{:}A}{\Psi \Vdash c \leftarrow \{P_{c,\overline{a_i}}\} \leftarrow \overline{a_i{:}A_i} : \{\overline{a_i{:}A_i} \vdash c{:}A\}}\ \{\,\}I$$

$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi \Vdash M : \{\overline{a_i{:}A_i} \vdash c{:}A\} \quad \Psi;\Gamma;\Delta', c{:}A \vdash Q_c :: d{:}D}{\Psi;\Gamma;\Delta,\Delta' \vdash c \leftarrow M \leftarrow \overline{a_i}; Q_c :: d{:}D}\ \{\,\}E$$

$$\frac{\Gamma \subseteq \overline{!u_i{:}B_i} \quad \Psi \Vdash M : \{\overline{!u_i{:}B_i} \vdash !u{:}A\} \quad \Psi;\Gamma, u{:}A;\Delta \vdash Q_{!u} :: c{:}C}{\Psi;\Gamma;\Delta \vdash !u \leftarrow M \leftarrow \overline{!u_i}; Q_{!u} :: c{:}C}\ \{\,\}E^{!}$$

$$\frac{\Psi \Vdash M : \tau \quad \Psi;\Gamma;\Delta \vdash P :: c{:}A}{\Psi;\Gamma;\Delta \vdash \mathtt{output}\ c\ M; P :: c{:}\tau \wedge A}\ \wedge R$$

$$\frac{\Psi, x{:}\tau;\Gamma;\Delta, c{:}A \vdash Q_x :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\tau \wedge A \vdash x \leftarrow \mathtt{input}\, c; Q_x :: d{:}D}\ \wedge L$$

$$\frac{\Psi, x{:}\tau;\Gamma;\Delta \vdash Q_x :: c{:}A}{\Psi;\Gamma;\Delta \vdash x \leftarrow \mathtt{input}\, c; Q_x :: c{:}\tau \supset A}\ \supset R$$

$$\frac{\Psi \Vdash M : \tau \quad \Psi;\Delta, c{:}A \vdash P :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\tau \supset A \vdash \mathtt{output}\ c\ M; P :: d{:}D}\ \supset L$$

$$\frac{}{\Psi;\Gamma;\cdot \vdash \mathsf{close}\, c :: c{:}\mathbf{1}}\ \mathbf{1}R \qquad \frac{\Psi;\Gamma;\Delta \vdash P :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\mathbf{1} \vdash \mathtt{wait}\ c; P :: d{:}D}\ \mathbf{1}L$$

$$\frac{\Psi;\Gamma;\Delta \vdash P_d :: d{:}A \quad \Psi;\Gamma;\Delta' \vdash Q :: c{:}B}{\Psi;\Gamma;\Delta,\Delta' \vdash \mathtt{output}\ c\ (d.P_d); Q :: c{:}A \otimes B}\ \otimes\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta, d{:}A, c{:}B \vdash R_d :: e{:}E}{\Psi;\Gamma;\Delta, c{:}A \otimes B \vdash d \leftarrow \mathtt{input}\ c; R_d :: e{:}E}\ \otimes\mathsf{L}$$

$$\frac{\Psi;\Gamma;\Delta, d{:}A \vdash R_d :: c{:}B}{\Psi;\Gamma;\Delta \vdash d \leftarrow \mathtt{input}\ c; R_d :: c : A \multimap B}\ \multimap\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta \vdash P_d :: d{:}A \quad \Psi;\Gamma;\Delta', c{:}B \vdash Q :: e{:}E}{\Psi;\Gamma;\Delta,\Delta', c{:}A \multimap B \vdash \mathtt{output}\ c\ (d.P_d); Q :: e{:}E}\ \multimap\mathsf{L}$$

$$\frac{\Psi;\Gamma;\cdot \vdash P_d :: d{:}A}{\Psi;\Gamma;\cdot \vdash \mathtt{output}\ c\ !(d.P_d) :: c{:}!A}\ !\mathsf{R} \qquad \frac{\Gamma, u{:}A;\Delta \vdash Q_u :: d{:}D}{\Gamma;\Delta, c{:}!A \vdash !u \leftarrow \mathtt{input}\ c; Q_u :: d{:}D}\ !\mathsf{L}$$

$$\frac{}{\Psi;\Gamma; d{:}A \vdash \mathtt{fwd}\, c\ d :: c{:}A}\ id \qquad \frac{\Psi;\Gamma, u{:}A;\Delta, c{:}A \vdash Q_c :: d{:}D}{\Psi;\Gamma, u{:}A;\Delta \vdash c \leftarrow \mathtt{copy}\ !u; Q_c :: d{:}D}\ \mathsf{copy}$$

Figure 4.2: Typing Process Expressions and the Contextual Monad.

combined with the monad, expressed with a fixpoint construct which is typed as follows:

$$\dfrac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi, F{:}\tau \rightarrow \{\overline{a_i{:}A_i} \vdash c{:}\nu X.A\}, x{:}\tau; \Gamma; \Delta \vdash P :: c{:}A\{\nu X.A/X\}}{\Psi \Vdash (\mathsf{fix}\, F\; x \ = c \leftarrow \{P\} \leftarrow \overline{a_i{:}A_i}) : \tau \rightarrow \{\overline{a_i{:}A_i} \vdash c{:}\nu X.A\}}$$

The fixpoint construct above allows us to define a recursive process expression parameterized by a functional argument of type τ, for convenience. To type such a fixpoint, we must be able to type the underlying process expression in the appropriate context, offering the unfolded recursive type, where F stands for the recursive definition itself and x for the parameter.

At the level of process expressions, we have an equi-recursive left rule for recursive types that silently unfolds the recursive type:

$$\dfrac{\Psi; \Gamma; \Delta, d{:}A\{\nu X.A/X\} \vdash P :: c{:}C}{\Psi; \Gamma; \Delta, d{:}\nu X.A \vdash P :: c{:}C}\ \nu\mathsf{L}$$

The operational semantics of the fixpoint simply consist of the unfolding of the recursive definition to a λ-abstraction that produces a monadic value when given the appropriate functional parameter, instantiating the recursive occurrence appropriately.

$$!\mathsf{eval}\ (\mathsf{fix}\, F\; x \ = c \leftarrow \{P\} \leftarrow \overline{a_i{:}A_i})\ (\lambda x.c \leftarrow \{P\{(\mathsf{fix}\, F\; x \ = c \leftarrow \{P\} \leftarrow \overline{a_i{:}A_i})/F\}\} \leftarrow \overline{a_i{:}A_i})$$

To clarify our take on recursive process expression definitions, consider the following session type that encodes an infinite stream of integers (for convenience, we assume a type definition mechanism in the style of ML or Haskell, distinguishing between functional and session type definitions with `type` and `stype`, respectively):

```
stype intStream = nu X.int /\ X
```

In order to produce such a stream, we write a recursive function producing a process expression:

```
nats : int -> {c:intStream}
c <- nats x =
{ output c x
  c' <- nats (x+1)
  fwd c c' }
```

This an example of a (recursive) function definition that produces a process expression. We take some liberties with the syntax of these definitions for readability. In particular, we list interface channels on the left-hand side of the definition and omit the explicit fixpoint construct. We also omit the semicolon that separates actions from their process expression continuation, using a new line for readability.

Since the recursive call must be accomplished via monadic composition, it will execute a new process expression offering along a fresh channel `c'`. Both for conciseness of notation and efficiency we provide a short-hand: if a tail-call of the recursive function provides a new channel which is then forwarded to the original offering channel, we can reuse the name directly, making

$$
\begin{array}{lcll}
M, N & ::= & \lambda x{:}\tau.\, M_x \mid M\, N \mid \ldots & \text{(usual functional constructs)} \\
 & \mid & a \leftarrow \{P_{a,\overline{a_i}}\} \leftarrow a_1, \ldots, a_n & \text{process providing } a\text{, using } a_1, \ldots, a_n \\[1ex]
P, Q & ::= & a \leftarrow M \leftarrow a_1, \ldots, a_n; P_a & \text{compose process computed by } M \\
 & & & \text{in parallel with } P_a\text{, communicating} \\
 & & & \text{along fresh channel } a \\
 & \mid & x \leftarrow \texttt{input}\ c; Q_x & \text{input a value } x \text{ along channel } c \\
 & \mid & \texttt{output}\ c\ M; P & \text{output value of } M \text{ along channel } c \\[1ex]
 & \mid & d \leftarrow \texttt{input}\ c; Q_d & \text{input channel } d \text{ along channel } c \\
 & \mid & \texttt{output}\ c\ (d.P_d); Q & \text{output a fresh channel } d \text{ along } c \\[1ex]
 & \mid & \texttt{close}\ c & \text{close channel } c \text{ and terminate} \\
 & \mid & \texttt{wait}\ c; P & \text{wait for closure of } c \\[1ex]
 & \mid & \texttt{output}\ c\ !(d.P_d) & \text{output a replicable } d \text{ along } c \text{ and terminate} \\
 & \mid & !u \leftarrow \texttt{input}\ c; Q_{!u} & \text{input shared channel } u \text{ along } c \\[1ex]
 & \mid & \texttt{case}\ c\ \texttt{of}\ \overline{l_j \Rightarrow P_j} & \text{branch on selection of } l_j \text{ along } c \\
 & \mid & c.l_j; P & \text{select label } l_j \text{ along } c \\[1ex]
 & \mid & c \leftarrow \texttt{copy}\ !u; P_c & \text{spawn a copy of } !u \text{ along } c \\[1ex]
 & \mid & \texttt{fwd}\ c_1\ c_2 & \text{forward between } c_1 \text{ and } c_2
\end{array}
$$

Figure 4.3: Language Syntax.

the last line of the function above simply `c <- nats (x+1)`. Superficially, it looks as if, for example, calling `nats 0` might produce an infinite loop. However, given that communication in our language is synchronous, the output will block until a matching consumer inputs the numbers.

To showcase the flexibility of the contextual nature of the monad, we now construct a stream transducer. As an example, we write a filter that takes a stream of integers and produces a stream of integers, retaining only those satisfying a given predicate $q : \mathsf{int} \to \mathsf{bool}$. In the code below we use a case analysis construct in the expression language for conciseness of presentation (this is easily encoded via case analysis in the functional layer, at the cost of added redundancy in the process expression definition).

```
filter : (int -> bool) -> { c:intStream |- d:intStream }
d <- filter q <- c =
{ x <- input c
  case q x
    of true  => output d x
                d <- filter q <- c
     | false => d <- filter q <- c }
```

Note that the `filter` function is recursive, but is not a valid coinductive definition unless we can show that filter will be true for infinitely many elements of the stream.

To review, we summarize the syntax of our language in Fig 4.3. As we have previously mentioned, an important aspect of our monadic integration is that it enables process expressions to be communicated as values by other process expressions. We showcase this feature (among others) in Section 4.3 below.

4.3 Examples

In this section we present three examples to showcase the expressiveness and flexibility of our concurrent programming language. We will show how to define two different implementations of stacks using monadic processes and how our logical foundation enables the development of simple and arguably elegant programs. Secondly, we will describe the implementation of a binary counter as a network of communicating processes. Finally, we present a high-level implementation of an “app store”, showcasing the ability of sending, receiving and executing process expressions (in the sense of higher-order process calculi), contrasting previous work where only purely functional values could be exchanged in communication [75].

For simplicity and readability we take some syntactic freedoms in the examples below. We write `Or {...}` as concrete syntax for $\oplus\{\ldots\}$; `t => s` as concrete syntax for $t \supset s$; `t /\ s` as concrete syntax for $t \wedge s$; and `Choice {...}` as concrete syntax for $\&\{\ldots\}$. We will also assume some basic data types in the functional layer for convenience (such as unit, naturals, lists, data polymorphism, etc.).

4.3.1 Stacks

We begin by defining the type for a stack session type:

```
stype Stack t =  Choice {push: t => Stack t
                         pop: Or {none: unit /\ Stack t
                                  some:t /\ Stack t}
                         dealloc: 1}
```

The `Stack` recursive type denotes a channel along which a process offers the choice between three operations: `push`, which will then expect the input of an element that will be the new top of the stack; `pop`, which outputs either the top element or unit (if the stack is empty); and `dealloc`, which fully deallocates the stack and thus has type **1**.

We present two distinct implementations of type `Stack`: `stack1` makes use of functional lists to maintain the stack; the second implementation `stack2`, more interestingly, makes use of a process version of a list to implement the stack as a network of communicating processes.

```
stack1 : list t -> {c:Stack t}
c <- stack1 nil =                             | c <- stack1 (v::l) =
{ case c of                                     { case c of
  push      => v <- input c                       push      => v' <- input c
                c <- stack1 (v::nil)                            c  <- stack1 (v'::v::l)
  pop       => c.none                             pop       => c.some
                output c ()                                     output c v
```

```
            c <- stack1 nil                           c  <- stack1 l
  dealloc => close c }                      dealloc => close c }
```

The code above consists of a function, taking a list and producing a monadic object indexed by the given list. We define the function by branching on the structure of the list, as usual. We can, for instance, create an empty stack by calling `stack1` with the empty list.

To produce our second implementation of a stack, we first define a concurrent implementation of a list. We start with the specification of the session types.

A process expression of type `List` can either behave as the empty list (i.e. offer no behavior) or as a list with a head and a tail, modelled by the output of the head element of the list, followed by the output of a *fresh* channel consisting of the handle to the tail list process. For convenience and generality, the session type employs *data* polymorphism in the elements maintained in the list.

```
stype List t = Or {nil: 1, cons: t => (List t * 1)}
```

We can now define two functions, `Nil` and `Cons`: the first produces a process expression that corresponds to the empty list and the second, given a value `v` of type `t` will produce a process expression that expects to interact with a channel `l` denoting a list of `t`'s, such that it will implement, along channel `c`, a new list with `v` as its head.

```
Nil : unit -> {c:List t}       Cons : t -> {l:List t |- c:List t}
c <- Nil () =                  c <- Cons v <- l =
{ c.nil                        { c.cons
  close c                        output c v
}                                output c (l'. fwd l l')
                                 close c }
```

Note that the `Cons` function, after sending the `cons` label along channel `c` and outputting `v`, it will output a fresh channel `l'` that is meant to represent the tail of the list, which is actually present along channel `l`. Thus, the function will also spawn a process expression that will forward between `l` and `l'`.

Having defined a concurrent implementation of a list, we may now use it to define our second implementation of a concurrent stack. We begin by defining a function `deallocList` (Fig. 4.4), whose purpose is to fully terminate a process network implementing a list. This entails recursively consuming the list session and terminating it.

We define our second stack implementation by making use of the `Cons`, `Nil` and `deallocList` functions. Thus, function `stack2` produces a monadic stack process with a process network implementing the list itself (Fig. 4.5). The monadic process expression specified in Fig. 4.5 begins by offering the three stack operations: `push`, `pop` and `dealloc`. The first inputs along the stack channel the element that is to be pushed onto the stack and calls on the `Cons` function to produce a monadic expression that appends to the list session l the new element, binding the resulting list process to l' and making a recursive call. The `pop` case needs to branch on whether or not the list

```
deallocList : unit -> {l:List t |- c:1}
  c <- deallocList () <- l =
  {  case l of
       nil  => wait l
               close c
       cons => v  <- input l
               l′ <- input l
               wait l
               c  <- deallocList () <- l′ }
```

Figure 4.4: List Deallocation.

```
stack2 : unit -> {l:List t |- c:Stack t }
 c <- stack2 () <- l =
 { case c of
   push    => v  <- input c
              l′ <- Cons v <- l
              c  <- stack2 <- l′
   pop     => case l of
              nil  => wait l
                      c.none
                      output c ()
                      l′ <- Nil ()
                      c  <- stack2 () <- l′
              cons => v  <- input l
                      l′ <- input l
                      wait l
                      c.some
                      output c v
                      c  <- stack2 () <- l′
  dealloc => c <- deallocList () <- l }
```

Figure 4.5: A Concurrent Stack Implementation.

session *l* encodes the empty list. If such is the case (`nil`), it waits for the termination of *l*, signals that the stack is empty and calls upon the `Nil` function to reconstruct an empty list for the recursive call; if not (`cons`), it inputs the element from the list session and the continuation list l'. It then outputs the received element and proceeds recursively. Finally, `deallocList` calls out to the list deallocation function.

4.3.2 A Binary Counter

As in our stack example above, we begin with the interface type:

```
stype Counter = Choice {inc: Counter
                        val: nat /\ Counter
                        halt: 1}
```

```
bit : nat -> {d:Counter |- c:Counter}
  c <- bit b <- d =
  {  case c of
       inc  => case b of
                 0 => c <- bit 1 <- d
                 1 => d.inc
                      c <- bit 0 <- d
       val  => d.val
               n <- input d
               output c (2*n+b)
               c <- bit b <- d
       halt => d.halt
               wait d
               close c }

epsilon : unit -> {c:Counter}
 c <- epsilon () =
 { case c of
    inc  => d <- epsilon ()
            c <- bit 1 <- d
    val  => output c 0
            c <- epsilon ()
    halt => close c }
```

Figure 4.6: A Bit Counter Network.

The `Counter` session type provides three operations: an `inc` operation, which increments its internal state; a `val` operation, which just outputs the counter's current value and a `halt` operation which terminates the counter.

One way to implement such a counter is through a network of communicating processes, each storing a single bit of the bit string that encodes the value of the counter. We do this by defining two mutually recursive functions `epsilon` and `bit`. The former encodes the empty bit string, abiding to the counter interface. Notably, in the `inc` branch, a new bit process is spawned with value 1. To do this, we make a recursive call to the epsilon function, bound to channel d, and then simply call the `bit` function with argument 1, also providing it with the channel d.

The `bit` function (Fig. 4.6) encodes an actual bit element of the bit string. It takes a number as an argument which is the 1 or 0 value of the bit and constructs a process expression that provides the counter interface along channel c by having access to the process encoding the previous bit in the string along channel d.

A bit b outputs the counter value by polling the previous bit for its counter value n and then outputting $2n + b$. This invariant ensures an adequate binary encoding of the counter. Termination triggers the cascade termination of all bits by sending a termination message to the previous bit, waiting on the channel and then terminating. The increment case simply recurses with value 1 if the bit is 0; otherwise it sends an increment message to the previous bit to encode the carry and recurses with value 0.

4.3.3 An App Store

Consider an App Store service that sells applications to its customers. These applications are not necessarily functional programs, in that they may communicate with the outside world.

In our example, the store offers a variety of different applications, themselves concurrent programs that rely on a remote API. Each store instance sends its client the application of choice as a monadic object and then terminates. The client can then run the application (in this scenario, all applications rely on a remote `API`, but the `weather` application also relies on a `GPS` module). The session types for the app. store and the weather app. are as follows:

```
stype Weather = Or{sun:Weather , rain:Weather , cloudy:Weather}
stype AppStore = !Choice{weather: payInfo =>
                               {d:API,g:GPS |- c:Weather} /\ 1,
                         travel: payInfo =>
                               {d:API |- c:Travel} /\ 1,
                         game: payInfo =>
                               {d:API |- c:Game} /\ 1}
```

The App Store service is represented as a replicated session since each Client-Store interaction is a separate, independent session with a finite behavior, even though the store itself is a persistent service that can be accessed by an arbitrary number of clients. On the other hand, the weather application offers a recursive session that can *continuously* inform the client of the current weather conditions.

Below we show the code for a client that downloads and runs the `weather` application, by appealing to some library function to enable its GPS module and then plugging all the components together (the GPS module interface along channel `g`, the API on channel `d` and the application itself, bound to `w`) and executing the received application through monadic composition.

```
ActivateGPS : unit -> {g:GPS}
WeatherClient : unit -> {!u:AppStore,d:API |- c:Weather}
c <- WeatherClient () <- !u,d = { s <- copy !u
                                  s.weather
                                  w <- input s
                                  wait s
                                  g <- ActivateGPS ()
                                  c <- w <- d,g }
```

This simple example shows how cleanly we can integrate *communication* and *execution* of open, process expressions in our language. It is relatively easy to extend this example to more complex communication interfaces.

$$\begin{array}{c} \Psi;\Gamma;\Delta \Rightarrow P :: c{:}A \\ \vdots \\ \dfrac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi \Vdash M : \{\overline{a_i{:}A_i} \vdash c{:}A\} \quad \Psi;\Gamma;\Delta',c{:}A \Rightarrow Q :: z{:}C}{\Psi;\Gamma;\Delta,\Delta' \Rightarrow \mathsf{spawn}(M;\overline{a_i};Q) :: z{:}C} \\ \Longrightarrow \\ \dfrac{\Psi;\Gamma;\Delta \Rightarrow P :: c{:}A \quad \Psi;\Gamma;\Delta',c{:}A \Rightarrow Q :: z{:}C}{\Psi;\Gamma;\Delta,\Delta' \Rightarrow (\nu c)(P \mid Q) :: z{:}C}\ \mathsf{cut} \end{array}$$

Figure 4.7: Monadic Composition as Cut

4.4 Connection to Higher-Order π-calculus

As the example of Section 4.3.3 shows, our language can easily encode process passing. Thus, the connection of our language with a variant of the π-calculus can only be achieved via a π-calculus with higher-order process features. The higher-order features of the calculus rely on the ability to communicate and execute processes as monadic values, using a dedicated construct which we dub spawn, similar to run in [53] or $\lfloor \star \rfloor$ in the higher-order calculi of [68].

The spawn π-calculus primitive allows for the execution of monadic values and is written as $\mathsf{spawn}(M;a_1,\dots,a_n;a.P)$, where $a_1,\dots,a_n$ denote the ambient channels required for the execution of the process expression underlying M, and a the channel that will be generated for the interactions with P, allowing for the execution of higher-order code. The operational semantics of spawn consists of evaluating M to a monadic value of the form $a \leftarrow \{Q\} \leftarrow d_1,\dots,d_n$ and then composing Q in parallel with P, sharing a fresh name a, where d_1 through d_n in Q are instantiated with the channels a_1 through a_n. The typing rule is virtually identical to that of monadic composition. The typing rule for the spawn construct is:

$$\dfrac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi \Vdash M : \{\overline{a_i{:}A_i} \vdash a{:}A\} \quad \Psi;\Gamma;\Delta',a{:}A \Rightarrow Q :: z{:}C}{\Psi;\Gamma;\Delta,\Delta' \Rightarrow \mathsf{spawn}(M;\overline{a_i};a.Q) :: z{:}C}\ (\textsc{spawn})$$

Operationally, the reduction rule for spawn matches precisely the SSOS rule for monadic composition, considering a big-step semantics for λ-terms since that is the strategy encoded in SSOS for convenience. We recall that λ-terms are evaluated using a call-by-value strategy and that monadic objects are values of the language.

$$\mathsf{spawn}(c \leftarrow \{P\} \leftarrow \overline{b};\overline{a};c.Q) \longrightarrow (\nu c)(P\{\overline{a}/\overline{b}\} \mid Q)$$

Using the cut rule we can justify the typing rule for spawn, shown pictorially in Fig. 4.7. Executing $\mathsf{spawn}(M;\overline{a_i};a.Q)$ results in the parallel composition of Q with the process underlying M, sharing a local name a. This composition under a name restriction matches precisely with the cut rule.

We thus develop a mapping from the process expressions of our language to higher-order processes (in the sense described above) with term passing (as in the π-calculus of Section 3.3.1).

The translation from process expressions to processes $\rightsquigarrow$ is extended with the following clause for monadic composition:

$$\begin{array}{lcl} P & \rightsquigarrow & \hat{Q} \\ \hline a \leftarrow M \leftarrow a_1, \ldots, a_n; P_a & \rightsquigarrow & \mathsf{spawn}(M; a_1, \ldots, a_n; a.\hat{P}_a) \end{array}$$

As before, our strict one-to-one correspondence of between π-calculus reduction rule and the SSOS rule greatly simplifies the connection between the two languages.

Theorem 22 (Static Correspondence - Process Expressions to π-calculus)**.** *If* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$ *then there exists a π-calculus process $\hat{Q}$ such that* $P \rightsquigarrow \hat{Q}$ *with* $\Psi; \Gamma; \Delta \Rightarrow \hat{Q} :: z{:}A$.

Theorem 23 (Static Correspondence - π-calculus to Process Expressions)**.** *If* $\Psi; \Gamma; \Delta \Rightarrow P :: z{:}A$ *then there exists a process expression $\hat{Q}$ such that* $P \rightsquigarrow^{-1} \hat{Q}$ *with* $\Psi; \Gamma; \Delta \vdash \hat{Q} :: z{:}A$.

The simulation results for execution of process expressions and π-calculus reduction follow naturally.

Theorem 24 (Simulation of SSOS Executions)**.** *Let* $\Psi; \Gamma; \Delta \vdash P :: z{:}A$*, with* $\mathsf{exec}\ P\ z\ A \longrightarrow^{1,2} \Theta$. *We have that* $P \rightsquigarrow \hat{P}$*, such that the following holds:*

(i) Either, $\hat{P} \rightarrow Q$*, for some* Q*, with* $\Omega \rightsquigarrow \equiv Q$

(ii) Or, $\Omega \rightsquigarrow Q$*, for some* Q*, such that* $\hat{P} \equiv Q$.

Theorem 25 (Simulation of π-calculus Reduction)**.** *Let* $\Psi; \Gamma; \Delta \Rightarrow P :: z{:}A$ *with* $P \rightarrow Q$. *We have that* $P \rightsquigarrow^{-1} \hat{P}$ *such that* $\mathsf{exec}\ \hat{P}\ z\ A \longrightarrow^{+} \Omega$ *with* $\Omega \rightsquigarrow \equiv Q$.

4.5 Metatheoretical Properties

In this section we study the standard metatheoretical properties of type preservation and global progress for our language.

We note that from the perspective of the functional language, an encapsulated process expression is a value and is not executed. Instead, functional programs can be used to construct concurrent programs which can be executed at the top-level, or with a special built-in construct such as *run*, which would have type

```
run : {c:1} -> unit
```

Not accidentally, this is analogous to Haskell's I/O monad [60].

4.5.1 Type Preservation

In the metatheoretical developments of previous sections we have established type preservation and progress for the π-calculus mapping of our logical interpretation. We can develop similar arguments for our language, adapting the reduction lemmas of Section 3.3.2 for $\forall$ and $\exists$ to our non-dependent versions ($\supset$ and $\wedge$, respectively):

Lemma 21 (Reduction Lemma - $\supset$)**.** *Assume*

(a) $\Psi;\Gamma;\Delta_1 \Rightarrow P :: x{:}\tau \supset A$ *with* $P \xrightarrow{x(V)} P'$ *and* $\Psi \Vdash V{:}\tau$

(b) $\Psi;\Gamma;\Delta_2, x{:}\tau \supset A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{x\langle V\rangle}} Q'$ *and* $\Psi \Vdash V{:}\tau$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. The proof follow exactly the schema of the proof of Lemma 13, except there is no type dependency. □

Lemma 22 (Reduction Lemma - $\wedge$)**.** *Assume*

(a) $\Psi;\Gamma;\Delta_1 \Rightarrow P :: x{:}\tau \wedge A$ *with* $P \xrightarrow{\overline{x\langle V\rangle}} P'$ *and* $\Psi \vdash V{:}\tau$

(b) $\Psi;\Gamma;\Delta_2, x{:}\tau \wedge A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x(V)} Q'$ *and* $\Psi \vdash V{:}\tau$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. The proof follow exactly the schema of the proof of Lemma 14, except there is no type dependency. □

Before establishing type preservation, we must establish that process reductions generated by the spawn construct preserve typing.

Lemma 23. *Assume*

(a) $\Psi;\Gamma;\Delta \Rightarrow \mathsf{spawn}(V;a_1,\ldots,a_n;a.P) :: z{:}C$, *where V is a value.*

(b) $\mathsf{spawn}(V;a_1,\ldots,a_n;a.P) \rightarrow Q$

Then:

(c) $\Psi;\Gamma;\Delta \Rightarrow Q :: z{:}C$

Proof. From (a) and inversion we know that the only possible reduction is:

$$\mathsf{spawn}(c \leftarrow \{R\} \leftarrow \overline{b};a_1,\ldots,a_n;a.P) \rightarrow (\nu c)(R\{\overline{a}/\overline{b}\} \mid P)$$

for some R. We also know that $\Delta = \Delta_1,\Delta_2$ such that $\Psi;\Gamma;\Delta_1 \Rightarrow R\{\overline{a}/\overline{b}\} :: c{:}A$, for some A and that $\Psi;\Gamma;\Delta_2, c{:}A \Rightarrow P :: z{:}C$. We conclude by the cut rule. □

We can thus show the following type preservation theorem.

Theorem 26 (Type Preservation)**.**

(a) If $\Psi \Vdash M{:}\tau$ *and* $M \rightarrow M'$ *then* $\Psi \Vdash M'{:}\tau$

(b) If $\Psi;\Gamma;\Delta \Rightarrow P :: z{:}C$ and $P \rightarrow P'$ then $\Psi;\Gamma;\Delta \Rightarrow P' :: z{:}C$

Proof. By simultaneous induction on typing. The cases for (a) are standard in the literature, with the exception of the case for the fixpoint construct which appeals to the i.h. (b).

The cases for process expressions, that is, those pertaining to (b), follow similarly to the proof of type preservation for the extension with value dependent session types (Theorem 12) with two new cases for spawn, one where the reduction is generated by evaluating the functional term M (which follows by i.h. (a)) and another where the functional term is already a value and the reduction transforms spawn into a parallel composition, which follows by Lemma 23. □

4.5.2 Global Progress

To establish a global progress result for the language we must extend the definition of a live process to account for the spawn construct.

Definition 7 (Live Process)**.** *A process P is considered* live*, written $live(P)$, if and only if the following holds:*

$$\begin{array}{lll} live(P) & \triangleq & P \equiv (\nu \overline{n}(\pi.Q \mid R) \textit{ for some } \pi.Q, R, \overline{n} \\ or & & P \equiv (\nu \overline{n})(\mathsf{spawn}(M;\overline{x};S) \mid R) \textit{ for some } M, \overline{x}, S, R, \overline{n} \end{array}$$

where $\pi.Q$ is a non-replicated guarded process with $\pi \neq x\langle\rangle$ for any x.

In essence, the spawn construct always denotes a potential for further action and thus must be accounted as a form of live process.

Global progress is established in a similar way to that of Section 3.3.3, but where progress in both functional and concurrent layers are intertwined.

Lemma 24 (Contextual Progress)**.**

(i) Let $\Psi;\Gamma;\Delta \Rightarrow P :: z{:}C$. If $live(P)$ then there is Q such that one of the following holds:

(a) $P \rightarrow Q$

(b) $P \xrightarrow{\alpha} Q$ for some α where $s(\alpha) \in z, \Gamma, \Delta$

(ii) Let $\Psi \Vdash M{:}\tau$ then either M is a value or $M \rightarrow M'$

Proof. By simultaneous induction on typing. □

Our global progress property follows from Lemma 24 as before, holding even in the presence of higher-order process passing and recursion in the sense described above, a previously open problem.

Theorem 27 (Global Progress)**.** *Let $\cdot;\cdot;\cdot \Rightarrow P :: x{:}\mathbf{1}$. If $live(P)$ then there is Q such that $P \rightarrow Q$.*

CHAPTER 5

RECONCILIATION WITH LOGIC

The language developed in the first sections of Chapter 4 departs from the full connection with linear logic through the inclusion of potentially divergent behavior (i.e. *non-terminating* internal, unobservable computations) in programs, which is inherently inconsistent with logical soundness. A practical consequence of this disconnect is that plugging together subsystems, e.g. as a result of dynamic channel passing or of linking downloaded (higher-order) code to local services as in the example of Section 4.3.3, may result in a system actually unable to offer its intended services due to divergent behavior, even if the component subsystems are well-typed and divergence-free, that is, the language no longer has the property of *compositional non-divergence*.

To recover the connections with logic and restore compositional non-divergence, while maintaining a reasonable degree of expressiveness, we restrict general recursive session types to their logically sound version of coinductive session types by eliminating general recursion and replacing it with *corecursion* (i.e. recursion that ensures *productivity* of the coinductive definition). We achieve this by putting in place certain syntactic constraints on the recursive definitions.

The nature of the syntactic constraints we impose on general recursion is in its essence similar to those used in dependently-typed programming languages such as Agda [55] or the language of the Coq proof assistant [74], however, the concurrent nature of process expressions introduces some interesting considerations. For instance, in the following:

```
P : int -> {c:intStream}
c <- P n = { output c n
             c′ <- P (n+1)
             x  <- input c′
             fwd c c′ }
```

we have the definition of an integer stream that given an integer n will output it along channel c. Afterwards, a recursive call is made that is bound to a local channel c′. We then input from c′ and conclude by forwarding between the two channels. It turns out that this seemingly simple recursive definition is *not* productive. A process interacting with `P`, after receiving the first integer would trigger an infinite sequence of internal actions in `P`. This argues that even seemingly harmless recursive definitions can introduce divergence in non-obvious ways.

To fully address the problem, we must first consider the meaning of productivity in our concurrent setting. The intuitive idea of productivity is that there must always be a finite sequence of internal actions between each *observable* action. In our setting, the observable actions are those on the session channel that is being *offered*, while internal actions are those generated through interactions with ambient sessions. Given our definition of productivity, a natural restriction is to require an action on the channel being offered before performing the recursive call (i.e. the recursive call must be *guarded* by a visible action). Thus, the definition,

```
c <- S n = { output c n
             c <- S (n+1)}
```

satisfies the guardedness condition and is in fact a valid coinductive definition of an increasing stream of integers. This is identical to the guardedness conditions imposed in Coq [34]. However, guardedness alone is not sufficient to ensure productivity. Consider our initial definition:

```
c <- P n = { output c n
             c′ <- P (n+1)
             x <- input c′
             fwd c c′}
```

While guardedness is satisfied, the definition is *not* productive. A process interacting with `P` will be able to observe the first output action, but no other actions are visible since they are consumed within `P` by the input action that follows the recursive call. In essence, `P` destroys its own productivity by internally interacting with the recursive call.

The problematic definition above leads us to consider restrictions on what may happen *after* a recursive call, which turn out to be somewhat subtle. For instance, we may impose that recursion be either terminal (i.e. the continuation must be a forwarder) or that after it an action on the observable channel must be performed, before interactions with the recursive call are allowed. While such a restriction indeed excludes the definition of `P`, it is insufficient:

```
c <- P′ n = { output c n
              c′ <- P′ (n+1)
              output c n
              x <- input c′
              y <- input c′
              fwd c c′}
```

The definition above diverges after the two outputs, essentially for the same reason `P` diverges. One could explore even more intricate restrictions, for instance imposing that the continuation be itself a process expression satisfying guardedness and moreover offering an action before interacting with its ambient context. Not only would this be a rather *ad-hoc* attempt, it is also not sufficient to ensure productivity.

Thus, our syntactic restrictions that ensure productivity are the combination of both *guardedness* and a condition we dub *co-regular recursion*, that is, we disallow interactions with recursive calls within corecursive definitions. To this end, we impose that process expressions that are composed with corecursive calls (via the monadic composition construct) may not communicate with the corecursive call, although they may themselves perform other actions.

To further crystallise our argument, we illustrate the execution behavior of the ill-behaved process expression P above, when composed with a process expression that consumes its initial input:

$$\begin{gathered}\mathsf{exec}(P(0)) \otimes \mathsf{exec}(x \leftarrow \mathsf{input}\ c; Q) \longrightarrow \\ \mathsf{exec}(c' \leftarrow P(1); x \leftarrow \mathsf{input}\ c'; \mathsf{fwd}\ c\ c') \otimes \mathsf{exec}(Q) \longrightarrow \\ \exists c'.\ \mathsf{exec}(P(1)_{c'}) \otimes \mathsf{exec}(x \leftarrow \mathsf{input}\ c'; \mathsf{fwd}\ c\ c') \otimes \mathsf{exec}(Q) \longrightarrow \ldots\end{gathered}$$

As we described, we eventually reach a configuration where the output produced by recursive calls is internally consumed by a corresponding input, resulting in an infinitely repeating pattern of internal behavior that never actually reaches the executing process expression Q.

5.0.3 Guardedness and Co-regular Recursion by Typing

It turns out that we may impose both guardedness and co-regular recursion via a rather simple restriction on the typing rule for (co)recursive definitions. Recall our initial typing rule for recursive definitions:

$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi, F{:}\tau \to \{\overline{a_i{:}A_i} \vdash c{:}\nu X.A\}, x{:}\tau; \Gamma; \Delta \vdash P :: c{:}A\{\nu X.A/X\}}{\Psi \Vdash (\mathsf{fix}\ F\ x\ = c \leftarrow \{P\} \leftarrow \overline{a_i{:}A_i}) : \tau \to \{\overline{a_i{:}A_i} \vdash c{:}\nu X.A\}}$$

In the rule above, as we have argued previously, nothing requires P to produce any observable action specified in the session type A nor do we restrict the way in which P may use the recursive occurrence F.

To ensure guardedness, we first introduce a standard strict positivity restriction on recursive types $\nu X.A$. This restriction does not by itself ensure guardedness, since even if $\nu X.A$ is strictly positive the body of the corecursive definition may simply appeal to the recursive call and produce no action. We thus impose an additional constraint by modifying the typing rule for corecursive definitions as follows:

$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi, F{:}\tau \to \{\overline{a_i{:}A_i} \vdash c{:}X\}, x{:}\tau; \Gamma; \Delta \vdash P :: c{:}A(X)}{\Psi \Vdash (\mathsf{fix}\ F\ x\ = c \leftarrow \{P\} \leftarrow \overline{a_i{:}A_i}) : \tau \to \{\overline{a_i{:}A_i} \vdash c{:}\nu X.A\}}\ (\textsc{corec})$$

Notice the subtle differences between the two rules: we now maintain the type offered by P as the open session type $A(X)$ and the type for F no longer mentions the offered type $c{:}\nu X.A$ but rather only $c{:}X$. Intuitively, what this accomplishes is that P must necessarily produce a single "unrolling" of the behavior A, and at any point while typing P all offers of the recursive type variable X must be satisfied via monadic composition with the recursive occurrence F (since the only way to offer X is via F). Moreover, consider what happens when we compose a process expression Q with the recursion variable F, such as:

$$c \leftarrow F\ x \leftarrow a_1, \ldots, a_n; Q$$

The process expression Q above is able to use a channel c typed with the specification of $F\ x$, which is $c{:}X$. Thus, Q cannot in fact interact with the recursive call in a destructive way such as those argued above. In fact, all that Q may do with this channel is forward it (although Q itself may produce many different behaviors by using other channels).

We thus make the informal case that combining the new typing rule for corecursive definitions with the strict positivity requirement on recursive session types ensures that non-divergent processes are excluded by typing: corecursive process expression definitions must produce some observable action as specified by the strictly positive recursive session type before appealing to the recursive call, since the type of the recursive call does not specify *any* behavior whatsoever. This also forbids destructive interactions with the recursive call since, from the perspective of a process expression that is composed with the recursive occurrence, it does not have access to any behavior that it may use. It is relatively easy to see how all the problematic recursive definitions given above are excluded by this new typing scheme. For instance, the definition,

```
c <- P n = { output c n
             c' <- P (n+1)
             x <- input c'
             fwd c c'}
```

where P could conceivably by typed with $c{:}\nu X.\mathsf{nat} \wedge X$, will fail to type since the input action `x <- input c'` is not well-typed since the channel c' cannot be used for inputs (it is typed with X). The definition of `P'` fails to type for similar reasons.

While we have informally argued that well-typed programs in our language never produce internally divergent behavior, in order to formally obtain this result we must develop reasoning techniques that go beyond those used to establish type preservation and progress. More precisely, we develop a theory of linear logical relations that enables us to show non-divergence of well-typed programs, as well as setting up a framework to reason about program (and process expression) equivalence. We give an account of this development in Part III, Section 6.1.2.

However, one case that has yet to be made is what is the cost of these restrictions in terms of expressiveness. What kind of programs are we now able to write, in the presence of this restricted form of recursion? While we are obviously restricting the space of valid programs, we argue that the expressiveness of the resulting programming language is not overly constrained via an extended example, consisting of an alternative implementation of the bit counter example of Section 4.3.2 that adheres to the necessary restrictions to ensure non-divergence.

5.0.4 Coregular Recursion and Expressiveness - Revisiting the Binary Counter

Our goal is to implement a coinductive counter protocol offering three operations: we can poll the counter for its current integer value; we can increment the counter value and we can terminate the counter. The protocol is encoded via the following session type,

```
Counter = Choice(val:int /\ Counter, inc:Counter, halt:1}
```

Our implementation of `Counter` consists of a network of communicating processes, each process node encoding a single bit of the binary representation of the counter value in little endian form (we can think of the leftmost process in the network as representing the least significative bit). These network nodes communicate with their two adjacent bit representations, whereas the top level coordinator for the counter communicates with the most significant bit process (and with the client), spawning new bits as needed. The protocol for the nodes is thus given by the following session type:

```
CImpl = Choice{val:int => int /\ CImpl,
               inc:Or{carry:CImpl, done:CImpl}, halt:1}
```

The protocol is as follows: to compute the integer value of the bit network, each node expects to input the integer value of the node to its right (i.e. the more significant bit), update this value with its own contribution and propagate it down the bit network. When the last bit is reached, the final value is sent back through the network to the client. To increment the value of the counter, given the little endian representation, we must propagate the increment command down the network to the least significant bit, which will flip its value and either send a `carry` or a `done` message up the chain, flipping the bits as needed (a 1 bit receiving `carry` will flip and propagate `carry`, whereas a 0 bit will flip and send `done`, signalling no more bits need to be flipped). If the top level coordinator receives a `carry` message we must generate a new bit node. The type and code for the node process is given below. Notice how `Node` is a valid coinductive definition: recursive calls are guarded by an action on `n` and coregular.

```
Node : int -> {x:CImpl |- n:CImpl}
n <- Node b <- x = { case n of
                      val  => x.val
                              m <- input n
                              output x (2*m+b)
                              m <- input x
                              output n m
                              n <- Node b <- x
                      inc  => x.inc
                              case x of
                               carry => if (b=1) then
                                         n.carry
                                         n <- Node 0 <- x
                                        else
                                         n.done
                                         n <- Node 1 <- x
                               done => n <- Node b <- x
                      halt => x.halt
                              wait x
                              close n}
```

The coordinator process interfaces between the clients and the bit network, generating new bit nodes as needed. Again, we note that `Coord` meets our criteria of a valid coinductive definition.

```
Coord : unit -> {x:CImpl |- z:Counter}
z <- Coord () <- x = { case z of
                        val  => x.val
                                output x 0
                                n <- input x
                                output z n
                                z <- Coord () <- x
                        inc  => x.inc
                                case x of
                                 carry => n <- Node 1 <- x
                                          z <- Coord () <- n
                                 done  => z <- Coord () <- x
                        halt => x.halt
                                wait x
                                close z}
```

The only remaining component is the representation of the empty bit string `epsilon`, which will be a closed process expression of type `x:Impl`, implement the "base cases" of the protocol. For polling the value, it just ping pongs the received value. For incrementing, it just signals the `carry` message. We can then put the `Counter` system together by composing `epsilon` and `Coord`.

```
epsilon:unit -> {x:CImpl}
x <- epsilon () = { case x of
                     val  => n <- input x
                             output x n
                             x <- epsilon ()
                     inc  => x.carry
                             x <- epsilon ()
                     halt => close x}
z <- Counter = { x <- epsilon ()
                 z <- Coord () <- x}
```

This example exhibits a more general pattern, where we have an interface protocol (in this case `Counter`) that is then implemented by a lower level protocol through some network of processes

(`CImpl`), similar to abstract data types and implementation types, but here from the perspective of a concurrent communication protocol.

5.1 Further Discussion

In the previous two chapters we have developed a programming language based on our logical interpretation of linear logic as concurrent process expressions by encapsulating linearity (and concurrency) inside a contextual monad in a functional programming language akin to a λ-calculus, for which we have shown a type safety result in the form of type preservation and global progress for both the functional and concurrent layers.

The development of the language itself posed a series of design challenges, some of which we now discuss in further detail.

Contextuality of the Monad As we have shown throughout this chapter, the contextual nature of the monad provides us with a way to directly represent *open* process expressions in our language.

The advantages of having the monad be contextual, instead of simply allowing us to form closed process expressions, is easily seen in our examples of Section 4.3, where not only the monadic types provide a much more accurate description of the process expression interface (since they specify both the dependencies and the offered behavior, as made clear in the App Store example of Section 4.3.3), but also for a cleaner construction of modular code. For instance, in the bit counter of Section 4.3.2 we can cleanly separate the definition of the "empty" bit counter `epsilon` from the definition of a `bit` node in the network, connected to its left and right neighbours in the chain. This is only made possible precisely due to the contextual nature of the monad, since otherwise we would only be able to specify closed process expressions.

We note that it is not even clear how we could represent `bit` from Section 4.3.2 only using closed process expressions. While we could conceivably pass the neighbouring bit as a functional monadic argument to `bit`, to then be explicitly composed in the body of the definition via monadic composition, and effected upon by the recursive definition, we would then have to reconstruct this effected neighbouring bit as a closed monadic object to pass as an argument to the recursive call, which if possible would be a rather unnatural programming pattern.

Even in the very simple case of our stream filter, a similar problem arises if we were to make the definition as a closed process expression: we would attempt to write a function `filter q P`, where `q` is the filter predicate and `P` the process expression that stands for the stream that we wish to filter. When we filter an element from `P` and wish to make the recursive call, how can we refer to the new state of `P` as a closed monadic expression that may be passed as an argument to the recursive call? Again, it is not obvious how this may be done without opting for a contextual version of the monad, where we refer to open process expressions directly.

Recursion and the Monad While general recursion is a natural feature in a more realistic programming language, readers may find it somewhat odd that we choose to incorporate recursive process definition via a functional recursor threaded through the contextual monad. The main justification for this particular design choice is that when considering a functional programming language, recursive definitions are usually taken as a given. While we could have simply added a recursor to the process expression language outright (in fact, we do just that in Chapter 6 since it makes formally reasoning about coinductive types and corecursive definitions technically simpler), we would be in a situation where we could write recursive process definitions in two distinct, yet arguably equivalent, ways: through functional-level recursion, threaded through the monad; and through process-level recursion. We believe this design choice would add a substantial deal of redundancy to the language. Alternatively we could restrict functional-recursion to completely avoid the monad, but in the presence of higher-order functions this seems to be a cumbersome "solution".

Thus, we opted to exclude recursion as a primitive process construct and simply include it as an explicit functional-level recursor that threads through the contextual monad as a compromise between these issues.

Synchronous Communication Another crucial design challenge is the nature of communication in the language. Given that our logical interpretation follows a pattern of synchronous communication, it is only natural to also adopt synchronous communication in our programming language (noting the work of [25] where an asynchronous interpretation of linear logic is developed). This design choice also makes reasoning about recursive definitions simpler: since communication actions are blocking, recursive calls will typically block until a matching action is presented (with the caveats discussed in Chapter 5).

In principle, it should be possible to develop an asynchronous version of our concurrent programming language following a similar design, although additional care is needed in the interactions with recursion.

Increasing Parallelism As we have previously discussed, it is possible to give an interpretation of the multiplicative unit **1** that increases the parallelism of process expressions by making the assignment silent, thus allowing for independent process expression composition. Alternatively, we could safely make the session closing and waiting actions asynchronous since they are the last actions on the given session. We mostly opted for a design closer to our fully synchronous logical interpretation of the previous chapters for the sake of uniformity, although it is conceivable that the increased parallelism generated by this alternative assignment should be present as an option in a real implementation of our language.

Distributed Computation We have mostly dealt with design issues pertaining to the integration of *concurrent* and functional computation and have not really addressed design issues that appear when we consider distributed computation, that is, the possibility of code executing at a remote site. Although our language has a natural degree of modularity (via the functional layer) that is amenable to a distributed implementation, there are fundamental design challenges that need to be studied in a distributed setting, namely the potential for remote code to not adhere to its static specification (since in principle we do not have access to remote source code). To this end, dynamic monitoring techniques that interpose between the trusted local code (which we can ensure type-checks) and the external running code would need to be developed, in order to ensure the correct behavior of the distributed system. Moreover, the actual access to remote objects is not primitive in the language we have presented, and methods for acquiring a remote session in a way that preserves typing would need to be considered.

Part III

Reasoning Techniques

In Part I of this document, we have mostly been concerned with connecting logic and concurrent computation by identifying concurrent phenomena that can be justified using our linear logical foundation, or dually, logical aspects that can be given a concurrent interpretation.

As we have argued from the start, a true logical foundation must also be able to provide us with techniques for *reasoning* about concurrent computation, not just be able to give them a logically consistent semantics. This is of paramount importance since reasoning about concurrent computation is notoriously hard. For instance, even establishing a seemingly simple property such as progress in traditional session types [26] already requires sophisticated technical machinery, whereas in this logical interpretation we obtain this property, roughly, "for free" [18, 58, 75]. More sophisticated and important properties such as termination [83] and confluence pose even harder challenges (no such results even exist for traditional session type systems). One of the existing main issues is the lack of a uniform proof technique that is robust enough to handle all these properties of interest in the face of a variety of different formalisms and languages of session types.

Beyond the actual formal machinery required to prove such properties, if we consider the techniques commonly used to reason about concurrent behavior we typically rely on behavioral equivalence techniques such as bisimulations [70, 84]. In a typed setting, it turns out to be somewhat challenging to develop bisimulations that are consistent with the canonical notion of behavioral (or observational) equivalence of barbed congruence. This is even more so the case in the higher-order setting [66, 72]. Moreover, these bisimulations are very specifically tuned to the particular language being considered and while common patterns arise, there is no unified framework that guides the development of these bisimulations.

We attempt to address these issues by developing a theory of *linear logical relations*. We apply the ideas of logical relations to our concurrent interpretation of linear logic, developing a rich framework which enables reasoning about concurrent programs in a uniform way, allowing us to establish sophisticated properties such as termination and confluence in a very natural and uniform way. The logical foundation also allows us to extend the linear logical relations beyond simple session types, also accounting for polymorphism and coinductive session types. Moreover, beyond serving as a general proof technique for session-typed programs, our linear logical relations, when extended to the binary case, naturally arise as typed bisimulations which we can show to coincide with barbed congruence in the polymorphic and simply-typed settings. These results establish linear logical relations as a general reasoning technique for concurrent session-typed computation, providing the remaining missing feature that supports our claim of linear logic as a foundation of message passing computation.

The main goal of this Part is to establish the generality and applicability of our theory of linear logical relations as an effective tool for establishing sophisticated properties and reasoning about concurrent programs. We seek to accomplish this by developing the concept of linear logical relations for the session-typed π-calculus that coincides with our concurrent interpretation of linear logic. We introduce the unary (or predicate) and binary (or relational) formulations of the technique for the polymorphic and coinductive settings. For the development of the binary version of the logical relations, we also introduce the canonical notion of (typed) observational equivalence for π-calculus of barbed congruence. Moreover, we present several applications of the logical relations technique, showing how they can be used to prove termination, reason about session type isomorphisms and observational equivalence in our typed settings.

CHAPTER 6

LINEAR LOGICAL RELATIONS

We develop our theory of linear logical relations by building on the well-known reducibility candidates technique of Girard [37]. The general outline of the development consists of defining a notion of *reducibility candidate* at a given type, which is a predicate on well-typed terms satisfying some crucial closure conditions. Given this definition, we can then define a logical predicate on well-typed terms by induction on the type (more precisely, first on the size of the typing contexts and then on the right-hand-side typing).

We highlight that our linear logical relations are defined on (typed) π-calculus processes. The reasoning behind this is twofold: first, it more directly enables us to explore the connections of our logical relations to usual process calculi techniques of bisimulations and barbed congruence, providing a more familiar setting in which to explore these ideas; secondly, it allows us to more deeply explore the π-calculus term assignment and take advantage of certain features that turn out to be technically convenient such as labelled transitions. The reason why labelled transitions are particularly convenient is because they allow us to reason about the observable behavior of a process as it interacts along its distinguished session channel, allowing for a systematic treatment of closed processes (while such a transition system could be defined for our SSOS, it already exists for the π-calculus assignment and so there's no reason to avoid it). To maintain this as a self-contained chapter, we reiterate the early labelled transition system for π-calculus processes in Fig. 6.1.

One interesting aspect of our development is how cleanly it matches the kind of logical relations arguments typically defined for λ-calculi. In particular, the conditions imposed by our definition of reducibility candidate are fundamentally the same.

6.1 Unary Logical Relations

We now present the unary logical relation that enables us to show termination by defining, as expected, a logical predicate on typed processes. We develop the logical relation for both the impredicative polymorphic (Section 3.5) and coinductive settings (a restricted version of the

$$\overset{\text{(out)}}{\overline{x}\langle y\rangle.P \xrightarrow{\overline{x\langle y\rangle}} P} \quad \overset{\text{(in)}}{x(y).P \xrightarrow{x(z)} P\{z/y\}} \quad \overset{\text{(outT)}}{\overline{x}\langle A\rangle.P \xrightarrow{\overline{x\langle A\rangle}} P} \quad \overset{\text{(inT)}}{x(Y).P \xrightarrow{x(B)} P\{B/Y\}}$$

$$\overset{\text{(id)}}{(\nu x)([x\leftrightarrow y] \mid P) \xrightarrow{\tau} P\{y/x\}} \quad \overset{\text{(par)}}{\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}} \quad \overset{\text{(com)}}{\frac{P \xrightarrow{\overline{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}} \quad \overset{\text{(res)}}{\frac{P \xrightarrow{\alpha} Q}{(\nu y)P \xrightarrow{\alpha} (\nu y)Q}}$$

$$\overset{\text{(rep)}}{!x(y).P \xrightarrow{x(z)} P\{z/y\} \mid !x(y).P} \quad \overset{\text{(open)}}{\frac{P \xrightarrow{\overline{x\langle y\rangle}} Q}{(\nu y)P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} Q}} \quad \overset{\text{(close)}}{\frac{P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')}}$$

$$\overset{\text{(lout)}}{x.\mathtt{inl};P \xrightarrow{\overline{x.\mathsf{inl}}} P} \quad \overset{\text{(rout)}}{x.\mathtt{inr};P \xrightarrow{\overline{x.\mathsf{inr}}} P} \quad \overset{\text{(lin)}}{x.\mathtt{case}(P,Q) \xrightarrow{x.\mathsf{inl}} P} \quad \overset{\text{(rin)}}{x.\mathtt{case}(P,Q) \xrightarrow{x.\mathsf{inr}} Q}$$

Figure 6.1: π-calculus Labeled Transition System.

calculus underlying Chapter 4), showcasing the main technical aspects of the development – a logical relation equipped with some form of mappings that treat type variables as appropriate for the particular setting under consideration.

While we have also developed a logical relation for a variant of the basic logical interpretation of Chapter 2 (see [58]), its original formulation was not general enough to allow for extensions to the richer settings analyzed here, insofar as we did not employ the reducibility candidates technique which is crucial in the development for the general case. We refrain from presenting the logical relation (in both unary and binary formulations) for the basic propositional setting since it is subsumed by the development for the polymorphic setting. The same results apply to the propositional formulation as a special case.

Before addressing the details of the logical relation, we introduce some preliminary definitions, notation and the overall structure of our formal development in the remainder of this Chapter. We state that a process P terminates (written $P \Downarrow$) if there is no infinite reduction sequence starting with P. We write $P \not\to$ to denote that no reductions starting from P are possible. We write $\Longrightarrow$ (resp. $\overset{\alpha}{\Longrightarrow}$) for the reflexive transitive closure of reduction (resp. labelled transitions) on π-calculus processes. We often use the terminology "the meaning of type A" or "the interpretation of type A", by which we mean the collection of processes that inhabit the logical relation at type A.

The logical predicates use the following extension to structural congruence (Def. 1) with the so-called *sharpened replication axioms* [68].

Definition 8. *We write $\equiv_!$ for the least congruence relation on processes which results from extending structural congruence $\equiv$ with the following axioms:*

1. $(\nu u)(!u(z).P \mid (\nu y)(Q \mid R)) \equiv_! (\nu y)((\nu u)(!u(z).P \mid Q) \mid (\nu u)(!u(z).P \mid R))$
2. $(\nu u)(!u(y).P \mid (\nu v)(!v(z).Q \mid R)) \equiv_! (\nu v)((!v(z).(\nu u)(!u(y).P \mid Q)) \mid (\nu u)(!u(y).P \mid R))$
3. $(\nu u)(!u(y).Q \mid P) \equiv_! P \quad \text{if } u \notin fn(P)$

The relation $\equiv_!$ allows us to properly "split" processes: axioms (1) and (2) represent the distribution of shared servers among processes, while (3) formalizes the garbage collection of shared servers which can no longer be invoked by any process. It is worth noticing that $\equiv_!$ expresses sound behavioral equivalences in our typed setting (specific instantiations are discussed in Section 7.3, pertaining to proof conversions involving $\mathsf{cut}^!$), insofar as (1) expresses the fact that in a closed system, a replicated process with two clients is indistinguishable from two copies of the replicated process, one assigned to each client; (2) expresses the fact in a closed system, usages of different replicated processes can be distributed, including when such usages are themselves done by replicated processes; (3) expresses the fact that in a setting where u is no longer used, no observable outcome can come from using such a session and so we may discard it.

We now state some auxiliary properties regarding extended structural congruence, reduction and labelled transitions which are useful for our development.

Proposition 2. *Let P and Q be well-typed processes:*

1. *If $P \to P'$ and $P \equiv_! Q$ then there is Q' such that $Q \to Q'$ and $P' \equiv_! Q'$.*

2. *If $P \xrightarrow{\alpha} P'$ and $P \equiv_! Q$ then there is Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \equiv_! Q'$.*

Proposition 3. *If $P \Downarrow$ and $P \equiv_! Q$ then $Q \Downarrow$.*

We now define the notion of a *reducibility candidate* at a given type: this is a predicate on well-typed processes which satisfies some crucial closure conditions. As in Girard's proof, the idea is that one of the particular candidates is the "true" logical predicate. Below and henceforth, $\cdot \Rightarrow P :: z{:}A$ stands for a process P which is well-typed under the empty typing environment.

Definition 9 (Reducibility Candidate)**.** *Given a type A and a name z, a* reducibility candidate *at $z{:}A$, written $\mathsf{R}[z{:}A]$, is a predicate on all processes P such that $\cdot \vdash P :: z{:}A$ and satisfy the following:*

(1) If $P \in \mathsf{R}[z{:}A]$ then $P \Downarrow$.

(2) If $P \in \mathsf{R}[z{:}A]$ and $P \Longrightarrow P'$ then $P' \in \mathsf{R}[z{:}A]$.

(3) If for all P_i such that $P \Longrightarrow P_i$ we have $P_i \in \mathsf{R}[z{:}A]$ then $P \in \mathsf{R}[z{:}A]$.

As in the functional case, the properties required for our reducibility candidates are termination (1), closure under reduction (2), and closure under backward reduction (or a form of head expansion) (3).

Given the definition of reducibility candidate we then define our logical predicate as an inductive definition. Intuitively, the logical predicate captures the behavior of processes as induced by typing. This way, e.g., the meaning of type $z{:}\forall X.A$ is that it specifies a process that after inputing an (arbitrary) type B, produces a process of type $z{:}A\{B/X\}$.

The overall structure of the logical relation can be broken down into two steps: the first step, which we dub as the *inductive case*, considers typed processes with a non-empty left-hand side typing context and proceeds through well-typed composition with closed processes in the logical predicate (thus making the left-hand side contexts smaller); the second step, which we dub as the *base case*, is defined inductively on the right-hand side type, considering processes with an empty left-hand typing context and characterizing the observable process behavior that is specified by the offered session type. Again, we explore the insight of identifying a process with the channel along which it offers its behavior. In the following two sections we develop the logical predicate for the polymorphic and coinductive settings.

6.1.1 Logical Predicate for Polymorphic Session Types

As we consider impredicative polymorphism, the main technical issue is that $A\{B/X\}$ may be larger than $\forall X.A$, for any reasonable measure of size, thus disallowing more straightforward inductive definitions of the base case of the logical predicate. We solve this technical issue in the same way as in the strong normalization proof for System F, by considering *open* types and parameterizing the logical predicate with mappings from type variables to types and from type variables to reducibility candidates at the appropriate types. This idea of a parameterized predicate is quite useful in the general case, not only for the polymorphism setting but also, as we will see, for the coinductive setting.

It is instructive to compare the key differences between our development and the notion of logical relation for functional languages with impredicative polymorphism, such as System F. In that context, types are assigned to *terms* and thus one maintains a mapping from type variables to reducibility candidates at the appropriate types. In our setting, since types are assigned to *channel names*, we need the ability to refer to reducibility candidates at a given type at channel names which are *yet to be determined*. Therefore, when we quantify over types and reducibility candidates at that type, intuitively, we need to "delay" the choice of the actual name along which the candidate must offer the session type, since we cannot determine the identity of this name at the point where the quantification is made. A reducibility candidate at type A which is "delayed" in this sense is denoted as $\mathsf{R}[-{:}A]$, where '$-$' stands for a name that is to be instantiated. It is easy to see why this technical device is necessary by considering the polymorphic identity type,

$$\mathsf{Id} \triangleq z{:}\forall X.X \multimap X$$

If we wish to inductively characterize the behavior of a process offering such a type, we seek to quantify over all types B that may instantiate X and all reducibility candidates at the type B, however, we cannot fix the channel name along which processes inhabiting this candidate predicate offer their behavior as z (the only "known" channel name at this point) since a process offering type Id will input a type (say B), input a fresh session channel (say x) that offers the behavior $x{:}B$, and then proceed as $z{:}B$. Therefore, to appropriately characterize this behavior at the open type $z{:}X \multimap X$ using our logical predicate, we must be prepared to identify the interpretation of the type variable X with candidates at channel names that are not necessarily the name being offered.

For the reader concerned with foundational issues that may lurk in the background of this slightly non-standard presentation, note that we are still in the same realm (outside of $\mathbf{PA_2}$) as in Girard's original proof. Quoting [37]:

"We seek to use "all possible" axioms of comprehension, or at least a large class of them. [...] to interpret the universal quantification scheme $t\ U$, we have to substitute in the definition of reducibility candidate for t, not an arbitrary candidate, but the one we get by induction on the construction of U. So we must be able to define a set of terms of type U by a *formula*, and this uses the comprehension scheme in an essential way."

In our setting, intuitively, we characterize a set of terms of type U by a slightly different, yet just as reasonable formula, making use of comprehension in a similar way.

For this particular setting of polymorphism, the logical predicate is parameterized by two mappings, ω and η. Given a context of type variables Ω, we write $\omega : \Omega$ to denote that ω is an assignment of closed types to variables in Ω. We write $\omega[X \mapsto A]$ to denote the extension of ω with a new mapping of X to A. We use a similar notation for extensions of η. We write $\hat{\omega}(P)$ (resp. $\hat{\omega}(A)$) to denote the simultaneous application of the mapping ω to free type-variables in P (resp. in A). We write $\eta : \omega$ to denote that η is an assignment of "delayed" reducibility candidates, to type variables in Ω (at the types specified in ω).

The logical predicate itself is defined as a sequent-indexed family of process predicates: a set of processes $\mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \Rightarrow T]$ satisfying some conditions is assigned to any sequent of the form $\Omega; \Gamma; \Delta \Rightarrow T$, provided both $\omega{:}\Omega$ and $\eta{:}\omega$. As mentioned above, the predicate is defined inductively on the structure of the sequents: the base case considers sequents with an empty left-hand side typing (abbreviated $\mathcal{T}^{\omega}_{\eta}[T]$), whereas the inductive case considers arbitrary typing contexts and relies on typed principles for process composition.

Definition 10 (Logical Predicate - Inductive Case)**.** *For any sequent $\Omega; \Gamma; \Delta \Rightarrow T$ with a non-empty left hand side environment, we define $\mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \Rightarrow T]$ (with $\omega : \Omega$ and $\eta : \omega$) as the set of processes inductively defined as follows:*

$$P \in \mathcal{T}^{\omega}_{\eta}[\Gamma; y{:}A, \Delta \Rightarrow T] \text{ iff } \forall R \in \mathcal{T}^{\omega}_{\eta}[y{:}A].(\nu y)(\hat{\omega}(R) \mid \hat{\omega}(P)) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \Rightarrow T]$$
$$P \in \mathcal{T}^{\omega}_{\eta}[u{:}A, \Gamma; \Delta \Rightarrow T] \text{ iff } \forall R \in \mathcal{T}^{\omega}_{\eta}[y{:}A].(\nu u)(!u(y).\hat{\omega}(R) \mid \hat{\omega}(P)) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \Rightarrow T]$$

Definition 10 above enables us to focus on closed processes by inductively "closing out" open processes according to the proper composition forms. While the definition above is specific to the polymorphic case due to the ω and η mappings, it exhibits a general pattern in our framework where we take an open well-typed process and "close it" by composition with closed processes in the predicate at the appropriate type, allowing us to focus on the behavior of closed processes, which takes place on the channel along which they offer a session.

The definition of the predicate for closed processes captures the behavior of a well-typed process, as specified by the session it is offering. We do this by appealing to the appropriate labelled transitions of the π-calculus. For instance, to define the logical predicate for processes offering a session of type $z{:}A \multimap B$ we specify that such a process must be able to input a channel y along z such that the resulting process P', when composed with a process Q that offers A along y (in the predicate), is in the predicate at the type $z{:}B$, which is precisely the meaning of a session of type $A \multimap B$. Similarly, the predicate at type $z{:}\mathbf{1}$ specifies that the process must be able to signal the session closure (by producing the observable $\overline{z\langle\rangle}$) and transition to a process that is structurally equivalent (under the extended notion of structural equivalence which "garbage collects" replicated processes that are no longer used) to the inactive process.

To specify the predicate at type $z{:}A \otimes B$ we identify the observable output of a fresh name y along z, after which we must be able to decompose the resulting process (using the extended structural congruence $\equiv_!$) into two parts: one which will be in the predicate at type $y{:}A$ and another in the predicate at $z{:}B$.

The interpretation of the exponential type $z{:}!A$ identifies the fact that processes must be able to output a fresh (unrestricted) name u, after which a replicated input is offered on this channel, whose continuation must be a process in the logical predicate at type A.

The interpretations of the choice and branching types $z{:}A \,\&\, B$ and $z{:}A \oplus B$ characterize the alternative nature of these sessions. Processes must be able to exhibit the appropriate selection actions, resulting in processes in the predicate at the selected type.

Type variables are accounted for in the predicate by appealing to the appropriate mappings. In the case for polymorphism this simply means looking up the η mapping for the particular type variable, instantiating the channel name accordingly. Thus, we define the predicate at type $z{:}\forall X.A$ by quantifying over all types and over all candidates at the given type, such that a process offering $z{:}\forall X.A$ must be able to input any type B and have its continuation be in the predicate at $z{:}A$, with the mappings extended accordingly. The case for the existential is dual. This quantification over all types and candidates matches precisely the logical relations argument for impredicative polymorphism in the functional setting. The formal definition of the base case for the logical predicate is given in Definition 11.

Definition 11 (Logical Predicate - Base Case)**.** *For any session type A and channel z, the logical predicate $\mathcal{T}^{\omega}_{\eta}[z{:}A]$ is inductively defined by the set of all processes P such that $\cdot \Rightarrow \hat{\omega}(P) :: z{:}\hat{\omega}(A)$ and satisfy the conditions in Figure 6.2.*

As usual in developments of logical relation arguments, the main burden of proof lies in showing that all well-typed terms (in this case, processes) are in the logical predicate at the appropriate type. To show this we require a series of auxiliary results which we now detail.

In proofs, it will be useful to have "logical representatives" of the dependencies specified in the left-hand side typing. We write $\Omega \triangleright \Gamma$ (resp. $\Omega \triangleright \Delta$) to denote that the types occurring in Γ (resp. Δ) are well-formed wrt the type variables declared in Ω (resp. Δ).

Definition 12. *Let $\Gamma = u_1{:}B_1, \ldots, u_k{:}B_k$, and $\Delta = x_1{:}A_1, \ldots, x_n{:}A_n$ be a non-linear and a linear typing environment, respectively, such that $\Omega \triangleright \Delta$ and $\Omega \triangleright \Gamma$, for some Ω. Letting $I = \{1, \ldots, k\}$ and $J = \{1, \ldots, n\}$, $\omega : \Omega$, $\eta : \omega$, we define the sets of processes $\mathcal{C}^{\omega,\eta}_{\Gamma}$ and $\mathcal{C}^{\omega,\eta}_{\Delta}$ as:*

$$\mathcal{C}^{\omega,\eta}_{\Gamma} \stackrel{def}{=} \{ \prod_{i \in I} !u_i(y_i).\hat{\omega}(R_i) \ \mid R_i \in \mathcal{T}^{\omega}_{\eta}[y_i{:}B_i] \} \qquad \mathcal{C}^{\omega,\eta}_{\Delta} \stackrel{def}{=} \{ \prod_{j \in J} \hat{\omega}(Q_j) \ \mid Q_j \in \mathcal{T}^{\omega}_{\eta}[x_j{:}A_j] \}$$

Proposition 4. *Let Γ and Δ be a non-linear and a linear typing environment, respectively, such that $\Omega \triangleright \Gamma$ and $\Omega \triangleright \Delta$, for some Ω. Assuming $\omega : \Omega$ and $\eta : \omega$, then for all $Q \in \mathcal{C}^{\omega,\eta}_{\Gamma}$ and for all $R \in \mathcal{C}^{\omega,\eta}_{\Delta}$, we have $Q \Downarrow$ and $R \Downarrow$. Moreover, $Q \not\longrightarrow$.*

The following lemma formalizes the purpose of "logical representatives": it allows us to move from predicates for sequents with non empty left-hand side typings to predicates with an empty

$$
\begin{array}{lll}
P \in \mathcal{T}^{\omega}_{\eta}[z{:}X] & \text{iff} & P \in \eta(X)(z) \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}\mathbf{1}] & \text{iff} & \forall P'.(P \overset{z\langle\rangle}{\Longrightarrow} P' \wedge P' \not\longrightarrow) \Rightarrow P' \equiv_{!} \mathbf{0} \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}A \multimap B] & \text{iff} & \forall P'y.(P \overset{z(y)}{\Longrightarrow} P') \Rightarrow \forall Q \in \mathcal{T}^{\omega}_{\eta}[y{:}A].(\nu y)(P' \mid Q) \in \mathcal{T}^{\omega}_{\eta}[z{:}B] \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}A \otimes B] & \text{iff} & \forall P'y.(P \overset{\overline{(\nu y)z\langle y\rangle}}{\Longrightarrow} P') \Rightarrow \\
 & & \quad \exists P_1, P_2.(P' \equiv_{!} P_1 \mid P_2 \wedge P_1 \in \mathcal{T}^{\omega}_{\eta}[y{:}A] \wedge P_2 \in \mathcal{T}^{\omega}_{\eta}[z{:}B]) \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}!A] & \text{iff} & \forall P'.(P \overset{\overline{(\nu u)z\langle u\rangle}}{\Longrightarrow} P') \Rightarrow \exists P_1.(P' \equiv_{!} !u(y).P_1 \wedge P_1 \in \mathcal{T}^{\omega}_{\eta}[y{:}A]) \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}A \,\&\, B] & \text{iff} & (\forall P'.(P \overset{z.\mathsf{inl}}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega}_{\eta}[z{:}A]) \wedge \\
 & & (\forall P'.(P \overset{z.\mathsf{inr}}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega}_{\eta}[z{:}B]) \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}A \oplus B] & \text{iff} & (\forall P'.(P \overset{\overline{z.\mathsf{inl}}}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega}_{\eta}[z{:}A]) \wedge \\
 & & (\forall P'.(P \overset{\overline{z.\mathsf{inr}}}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega}_{\eta}[z{:}B]) \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A] & \text{iff} & (\forall B, P', \mathsf{R}[-{:}B].\ (B\ \mathsf{type} \wedge P \overset{z(B)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X \mapsto B]}_{\eta[X \mapsto \mathsf{R}[-:B]]}[z{:}A]) \\
P \in \mathcal{T}^{\omega}_{\eta}[z{:}\exists X.A] & \text{iff} & (\exists B, \mathsf{R}[-{:}B].(B\ \mathsf{type} \wedge P \overset{\overline{z\langle B\rangle}}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X \mapsto B]}_{\eta[X \mapsto \mathsf{R}[-:B]]}[z{:}A])
\end{array}
$$

Figure 6.2: Logical predicate (base case).

left-hand side typing, provided processes have been appropriately closed. We write $\widetilde{x}$ to denote a sequence of distinct names $x_1, \ldots, x_n$.

Lemma 25. *Let P be a process such that $\Omega; \Gamma; \Delta \Rightarrow P :: T$, with $\Gamma = u_1{:}B_1, \ldots, u_k{:}B_k$ and $\Delta = x_1{:}A_1, \ldots, x_n{:}A_n$, with $\Omega \triangleright \Delta$, $\Omega \triangleright \Gamma$, $\omega : \Omega$, and $\eta : \omega$.*
We have: $P \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \Rightarrow T]$ iff $\forall Q \in \mathcal{C}^{\omega,\eta}_{\Gamma}, \forall R \in \mathcal{C}^{\omega,\eta}_{\Delta}, (\nu \widetilde{u}, \widetilde{x})(P \mid Q \mid R) \in \mathcal{T}^{\omega}_{\eta}[T]$.

Proof. Straightforward from Def. 10 and 12. □

The key first step in our proof is to show that our logical predicate satisfies the closure conditions that define our notion of a reducibility candidate at a given type.

Theorem 28 (The Logical Predicate is a Reducibility Candidate). *If $\Omega \vdash A$ type, $\omega : \Omega$, and $\eta : \omega$ then $\mathcal{T}^{\omega}_{\eta}[z{:}A]$ is a reducibility candidate at $z{:}\hat{\omega}(A)$.*

Proof. By induction on the structure of A; see Appendix A.4.1 for details. □

Given that our logical predicate manipulates open types, giving them meaning by instantiating the type variables according to the mappings ω and η, we show a compositionality result that relates the predicate at an open type A with a free type variable X (with the appropriate mappings) to the predicate at the instantiated version of A.

Theorem 29 (Compositionality). *For any well-formed type B, the following holds:*

$$P \in \mathcal{T}^{\omega[X \mapsto \hat{\omega}(B)]}_{\eta[X \mapsto \mathcal{T}^{\omega}_{\eta}[-:B]]}[x{:}A] \ \text{ iff } \ P \in \mathcal{T}^{\omega}_{\eta}[x{:}A\{B/X\}]$$

Proof. By induction on the structure of A.

Case: $A = X$

$P \in \mathcal{T}^{\omega[X\mapsto\hat{\omega}(B)]}_{\eta[X\mapsto\mathcal{T}^{\omega}_{\eta}[-:B]]}[x{:}X]$ assumption

$P \in \eta[X \mapsto \mathcal{T}^{\omega}_{\eta}[z{:}B]](X)(x)$ by Def. 11

$P \in \mathcal{T}^{\omega}_{\eta}[x{:}B]$ by Def. of η

$P \in \mathcal{T}^{\omega}_{\eta}[x{:}B]$ assumption

T.S: $P \in \mathcal{T}^{\omega[X\mapsto\hat{\omega}(B)]}_{\eta[X\mapsto\mathcal{T}^{\omega}_{\eta}[-:B]]}[x{:}X]$

S.T.S: $P \in \mathcal{T}^{\omega}_{\eta}[x{:}B]$ follows by assumption

Case: $A = \forall Y.A_1$

$P \in \mathcal{T}^{\omega[X\mapsto\hat{\omega}(B)]}_{\eta[X\mapsto\mathcal{T}^{\omega}_{\eta}[-:B]]}[x{:}\forall X.A_1]$ by assumption

$\forall C, P', \mathsf{R}[- : C].\ (C\ \mathsf{type} \wedge P \overset{x(C)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X\mapsto\hat{\omega}(B), Y\mapsto C]}_{\eta[X\mapsto\mathcal{T}^{\omega}_{\eta}[-:B], Y\mapsto\mathsf{R}[-:C]]}[x{:}A_1]$ by Def. 11

Choose P' arbitrarily, such that $P \overset{x(C)}{\Longrightarrow} P'$, for any C type and $\mathsf{R}[-{:}C]$:

$P' \in \mathcal{T}^{\omega[Y\mapsto C]}_{\eta[Y\mapsto\mathsf{R}[-:C]]}[x{:}A_1\{B/X\}]$ by i.h.

$P \in \mathcal{T}^{\omega}_{\eta}[x{:}\forall X.A_1\{B/X\}]$ by Def. 11, satisfying $\Rightarrow$

$P \in \mathcal{T}^{\omega}_{\eta}[x{:}\forall Y.A_1\{B/X\}]$ by assumption

$\forall C, P', \mathsf{R}[- : C].\ (C\ \mathsf{type} \wedge P \overset{x(C)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[Y\mapsto C]}_{\eta[Y\mapsto\mathsf{R}[-:C]]}[x{:}A_1\{B/X\}]$ by Def. 11

Choose P' arbitrarily, such that $P \overset{x(C)}{\Longrightarrow} P'$, for any C type and $\mathsf{R}[-{:}C]$:

$P' \in \mathcal{T}^{\omega[X\mapsto\hat{\omega}(B), Y\mapsto C]}_{\eta[X\mapsto\mathcal{T}^{\omega}_{\eta}[-:B], Y\mapsto\mathsf{R}[-:C]]}[x{:}A_1]$ by i.h

$P \in \mathcal{T}^{\omega[X\mapsto\hat{\omega}(B)]}_{\eta[X\mapsto\mathcal{T}^{\omega}_{\eta}[-:B]]}[x{:}\forall X.A_1]$ by Def. 11, satisfying $\Leftarrow$

All other cases are similar to the above, following by a straightforward induction.

□

The following lemma captures the fact that, for any well-formed type A, we may consistently move from the predicate at $z{:}A$ to the predicate at $x{:}A$ by renaming z to x at the process level (provided x is not free in the process structure).

Lemma 26 (Renaming)**.** *Let A be a well-formed type. If $P \in \mathcal{T}^{\omega}_{\eta}[z{:}A]$ and $x \notin fn(P)$ then $P\{x/z\} \in \mathcal{T}^{\omega}_{\eta}[x{:}A]$.*

Proof. Immediate from the definition of the logical predicate. □

Finally, we show that we may discard mappings to type variables when they are not used in a type.

Lemma 27 (Strengthening)**.** *If $P \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto\mathsf{R}[-:B]]}[x{:}A]$ and $X \notin fv(A)$ then $P \in \mathcal{T}^{\omega}_{\eta}[x : A]$*

Proof. Straightforward by induction on A. □

Having established all the necessary auxiliary results we can now show the so-called fundamental theorem of logical relations, stating that any well-typed process is in the logical predicate at the appropriate typing.

Theorem 30 (Fundamental Theorem)**.** *If* $\Omega; \Gamma; \Delta \Rightarrow P :: T$ *then, for all* $\omega : \Omega$ *and* $\eta : \omega$*, we have that* $\hat{\omega}(P) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \Rightarrow T]$.

Proof. By induction on typing. We show here the cases that pertain to polymorphism and to forwarding. In each case, we appeal to Lemma 25 and show that every $M = (\nu \widetilde{u}, \widetilde{x})(P \mid G \mid D)$ with $G \in \mathcal{C}^{\omega,\eta}_{\Gamma}$ and $D \in \mathcal{C}^{\omega,\eta}_{\Delta}$ is in $\mathcal{T}^{\omega}_{\eta}[T]$.

Case (T$\forall$R2): $\Omega; \Gamma; \Delta \Rightarrow z(X).P :: z{:}\forall X.A$

$\Omega, X; \Gamma; \Delta \Rightarrow P :: z{:}A$ by inversion

$\hat{\omega}(P)\{^{B}/X\} \in \mathcal{T}^{\omega'}_{\eta'}[\Gamma; \Delta \Rightarrow z{:}A]$ by i.h.,

for some B, $\omega' = \omega[X \mapsto B]$ and $\eta' = \eta[X \mapsto R[- : B]]$

Pick any $G \in \mathcal{C}^{\omega',\eta'}_{\Gamma}$, $D \in \mathcal{C}^{\omega',\eta'}_{\Delta}$:

$G \Downarrow, G \not\longrightarrow, D \Downarrow$ by Prop. 4

$(\nu \widetilde{u}, \widetilde{x})(\hat{\omega}(P)\{^{B}/X\} \mid G \mid D) \in \mathcal{T}^{\omega'}_{\eta'}[z{:}A]$ (a) by Lemma 25

S.T.S: $M = (\nu \widetilde{u}, \widetilde{x})(\hat{\omega}(z(X).P) \mid G \mid D) \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A]$

$M \overset{z(B)}{\Longrightarrow} (\nu \widetilde{u}, \widetilde{x})(\hat{\omega}(P)\{^{B}/X\} \mid G \mid D') = M_1$

$M_1 \in \mathcal{T}^{\omega'}_{\eta'}[z{:}A]$ by (a) and forward closure

$M \in \mathcal{T}^{\omega'}_{\eta'}[z{:}\forall X.A]$ by Def. 11

$M \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A]$ by Lemma 27

Case (T$\forall$L2): $\Omega; \Gamma; \Delta, x : \forall X.A \Rightarrow x\langle B\rangle.P :: T$

$\Omega \vdash B\ \mathsf{type}$ by inversion

$\Omega; \Gamma; \Delta, x : A\{^{B}/X\} \Rightarrow P :: T$ by inversion

$\hat{\omega}(P) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta, x : A\{^{B}/X\} \Rightarrow T]$ by i.h.

Pick any $G \in \mathcal{C}^{\omega,\eta}_{\Gamma}$, $D \in \mathcal{C}^{\omega,\eta}_{\Delta}$:

$G \Downarrow, G \not\longrightarrow, D \Downarrow$ by Prop. 4

$(\nu \widetilde{u}, \widetilde{x})(\hat{\omega}(P) \mid G \mid D \mid \mathcal{T}^{\omega}_{\eta}[x : A\{^{B}/X\}]) \in \mathcal{T}^{\omega}_{\eta}[T]$ (a) by Lemma 25

S.T.S: $(\nu \widetilde{u}, \widetilde{x}, x)(\hat{\omega}(x\langle B\rangle.P) \mid G \mid D \mid \mathcal{T}^{\omega}_{\eta}[x : \forall X.A]) \in \mathcal{T}^{\omega}_{\eta}[T]$

Pick $R \in \mathcal{T}^{\omega}_{\eta}[x{:}\forall X.A]$

S.T.S: $M = (\nu \widetilde{u}, \widetilde{x}, x)(x\langle \hat{\omega}(B)\rangle.\hat{\omega}(P)) \mid G \mid D \mid R) \in \mathcal{T}^{\omega}_{\eta}[T]$

$R \overset{x(\hat{\omega}(B))}{\Longrightarrow} R'$ by Theorem 28

$\forall \mathsf{R}[- : \hat{\omega}(B)].R' \in \mathcal{T}^{\omega[X \mapsto \hat{\omega}(B)]}_{\eta[X \mapsto \mathsf{R}[- : \hat{\omega}(B)]]}[x{:}A]$ by Def. 11

$R' \in \mathcal{T}^{\omega[X \mapsto \hat{\omega}(B)]}_{\eta[X \mapsto \mathcal{T}^{\omega}_{\eta}[- : B]]}[x{:}A]$ by Theorem 28

$R' \in \mathcal{T}^{\omega}_{\eta}[x{:}A\{^{B}/X\}]$ by Theorem 29

$M \Longrightarrow (\nu\widetilde{u}, \widetilde{x}, x)(\hat{\omega}(P)) \mid G \mid D' \mid R') = M_1$

$M_1 \in \mathcal{T}^{\omega}_{\eta}[T]$ by (a) and forward closure

$M \in \mathcal{T}^{\omega}_{\eta}[T]$ by backward closure

Case (T∃R2): $\Omega; \Gamma; \Delta \Rightarrow z\langle B\rangle.P :: z{:}\exists X.A$

$\Omega; \Gamma; \Delta \Rightarrow P :: z{:}A\{^{B}/X\}$ by inversion

$\Omega \vdash B\ \mathsf{type}$ by inversion

$\hat{\omega}(P) \in \mathcal{T}^{\omega}_{\eta}[\Gamma; \Delta \Rightarrow z{:}A\{^{B}/X\}]$ by i.h.

Pick any $G \in \mathcal{C}^{\omega,\eta}_{\Gamma}$, $D \in \mathcal{C}^{\omega,\eta}_{\Delta}$:

$G \Downarrow, G \not\longrightarrow, D \Downarrow$ by Prop. 4

$M_1 = (\nu\widetilde{u}, \widetilde{x})(\hat{\omega}(P) \mid G \mid D) \in \mathcal{T}^{\omega}_{\eta}[z{:}A\{^{B}/X\}]$ (a) by Lemma 25

S.T.S: $(\nu\widetilde{u}, \widetilde{x})(\hat{\omega}(z\langle B\rangle.P) \mid G \mid D) \in \mathcal{T}^{\omega}_{\eta}[z{:}\exists X.A]$

S.T.S: $M = (\nu\widetilde{u}, \widetilde{x})(z\langle\hat{\omega}(B)\rangle.\hat{\omega}(P) \mid G \mid D) \in \mathcal{T}^{\omega}_{\eta}[z{:}\exists X.A]$

S.T.S: $\exists \mathsf{C}.\mathsf{R}[-{:}\mathsf{C}].(\mathsf{C}\ \mathsf{type} \wedge M \overset{z\langle C\rangle}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X\mapsto C]}_{\eta[X\mapsto\mathcal{R}[-:C]}[z{:}A]$

S.T.S: $\exists \mathsf{C}.\mathsf{R}[-:\mathsf{C}].(\mathsf{C}\ \mathsf{type} \wedge M \overset{z\langle C\rangle}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega}_{\eta}[z{:}A\{^{\mathsf{C}}/X\}]$ (b) by Theorem 29

Pick $\mathsf{C} = \hat{\omega}(B)$ and $\mathsf{R}[-:\mathsf{C}]$ as $\mathcal{T}^{\omega}_{\eta}[-:\hat{\omega}(B)]$

$M \overset{z(\hat{\omega}(B))}{\Longrightarrow} (\nu\widetilde{u}, \widetilde{x})(\hat{\omega}(P) \mid G \mid D')$

$M_1 \Longrightarrow (\nu\widetilde{u}, \widetilde{x})(\hat{\omega}(P) \mid G \mid D')$

$(\nu\widetilde{u}, \widetilde{x})(\hat{\omega}(P) \mid G \mid D') \in \mathcal{T}^{\omega}_{\eta}[z{:}A\{^{B}/X\}]$ by forward closure, satisfying (b)

Case (T∃L2): $\Omega; \Gamma; \Delta, x : \exists X.A \Rightarrow x(X).P :: T$

$\Omega, X; \Gamma; \Delta, x{:}A \Rightarrow P :: T$ by inversion

$\hat{\omega}(P)\{^{B}/X\} \in \mathcal{T}^{\omega'}_{\eta'}[\Gamma; \Delta, x{:}A \Rightarrow T]$ by i.h.,

for some B, $\omega' = \omega[X \mapsto B]$ and $\eta' = \eta[X \mapsto R[-{:}B]]$

Pick any $G \in \mathcal{C}^{\omega,\eta}_{\Gamma}$, $D \in \mathcal{C}^{\omega,\eta}_{\Delta}$:

$G \Downarrow, G \not\longrightarrow, D \Downarrow$ by Prop. 4

$(\nu\widetilde{u}, \widetilde{x}, x)(\hat{\omega}(P)\{^{B}/X\} \mid G \mid D \mid \mathcal{T}^{\omega'}_{\eta'}[x{:}A]) \in \mathcal{T}^{\omega'}_{\eta'}[T]$ (a) by Lemma 25

Pick $R' \in \mathcal{T}^{\omega'}_{\eta'}[x{:}A]$:

$(\nu\widetilde{u}, \widetilde{x}, x)(\hat{\omega}(P)\{^{B}/X\} \mid G \mid D \mid R') \in \mathcal{T}^{\omega'}_{\eta'}[T]$ (b) by (a)

S.T.S: $(\nu\widetilde{u}, \widetilde{x}, x)(x(X).\hat{\omega}(P) \mid G \mid D \mid \mathcal{T}^{\omega}_{\eta}[x{:}\exists X.A]) \in \mathcal{T}^{\omega}_{\eta}[T]$

Pick $R = x\langle B\rangle.R'$

$R \overset{x\langle B\rangle}{\Longrightarrow} R'$ and $R' \in \mathcal{T}^{\omega'}_{\eta'}[x{:}A]$

$R \in \mathcal{T}^{\omega}_{\eta}[x{:}\exists X.A]$ by Def. 11

S.T.S: $M = (\nu\widetilde{u}, \widetilde{x}, x)(x(X).\hat{\omega}(P) \mid G \mid D \mid R) \in \mathcal{T}^{\omega}_{\eta}[T]$

$M \Longrightarrow (\nu\widetilde{u}, \widetilde{x}, x)(\hat{\omega}(P)\{^{B}/X\} \mid G \mid D \mid R') \in \mathcal{T}^{\omega'}_{\eta'}[T]$ (c) by (b)

$M \in \mathcal{T}^{\omega'}_{\eta'}[T]$ by backward closure and (c)

$M \in \mathcal{T}^{\omega}_{\eta}[T]$ by Lemma 27

Case (Tid): $\Omega; \Gamma; x{:}A \Rightarrow [x \leftrightarrow z] :: z{:}A$

$$\begin{array}{ll}
\text{Pick any } G \in \mathcal{C}^{\omega,\eta}_{\Gamma}: & \\
G \Downarrow, G \not\longrightarrow & \text{by Prop. 4} \\
D \in \mathcal{T}^{\omega}_{\eta}[x{:}A] & \text{(a)} \\
\text{S.T.S: } M = (\nu \widetilde{u}, x)([x \leftrightarrow z] \mid G \mid D) \in \mathcal{T}^{\omega}_{\eta}[z{:}A] & \\
M \longrightarrow (\nu \widetilde{u})(G\{z/x\} \mid D\{z/x\}) \equiv_{!} D\{z/x\} = M' & \text{(b) since } x \notin fn(G) \\
M' \in \mathcal{T}^{\omega}_{\eta}[z{:}A] & \text{(c) by (a) and Lemma 26} \\
M \in \mathcal{T}^{\omega}_{\eta}[z{:}A] & \text{(d) by (b), (c), and backward closure} \\
[x \leftrightarrow z] \in \mathcal{T}^{\omega}_{\eta}[\Gamma; x : A \Rightarrow z : A] & \text{by (d) and Lemma 25}
\end{array}$$

□

It follows immediately from the fundamental theorem that well-typed processes are terminating.

Theorem 31 (Termination)**.** *If* $\Omega; \Gamma; \Delta \Rightarrow P :: T$ *then* $\hat{\omega}(P) \Downarrow$*, for every* $\omega : \Omega$*.*

6.1.2 Logical Predicate for Coinductive Session Types

One of the key advantages of the logical relations framework that we have presented in the previous section is its uniformity in accounting for several different formulations of session types. We make this explicit by giving an account of a logical predicate for coinductive session types in a language similar to the higher-order process calculus of Chapter 4 with corecursive definitions. To distinguish between the predicate for polymorphic session types and coinductive session types, here we use the notation $\mathcal{L}^{\omega}[x{:}A]$.

Unlike the language of Chapter 4 which includes recursive definitions as a feature of the functional layer of the language, we instead opt for a π-calculus with a native recursor. This simplifies the technical challenges of having to thread recursive process definitions through the functional language via the monad and enables us to focus more precisely on the concurrent layer of the language. As we have already discussed, recursion is orthogonal to the specifics of the monadic embedding, which is directly related to the higher-order features of the process calculus.

Thus, the process calculus we consider in this section contains a combination of the term-passing features of the π-calculus used in Section 3.1, the higher-order features of the calculus of Section 4.4 and a form of recursion.

6.1.2.1 A Higher-Order Process Calculus with Corecursion

To maintain the presentation as self-contained as possible, we summarize here the syntax and typing rules for this π-calculus, with an emphasis on the rules for typing corecursive definitions. The syntax of processes is given below and follows that of previous formulations of π-calculi in this document:

$$\begin{array}{lll} M,N & ::= & \ldots \quad \text{(basic data constructors)} \\ P,Q & ::= & x\langle M\rangle.P \mid x(y).P \mid x\langle y\rangle.P \mid (\nu y)P \mid !x(y).P \mid P \mid Q \mid \mathsf{spawn}(M;\overline{a_i};a.Q) \\ & \mid & x.\mathsf{case}(\overline{l_j \Rightarrow P_j}) \mid x.l_i;P \mid (\mathsf{corec}\ X(\overline{y}).P)\ \overline{c} \mid X(\overline{c}) \mid [x \leftrightarrow y] \mid \mathbf{0} \end{array}$$

We highlight our parametrized *corecursion* operator $(\mathsf{corec}\ X(\overline{y}).P)\ \overline{c}$, enabling corecursive process definitions (the variables $\overline{y}$ are bound in P). The parameters are used to instantiate channels (and values) in recursive calls accordingly.

The operational semantics (both reduction and labelled transitions) for (channel and data) communication, branching and replication is as in previous sections. We highlight only the labelled transition rule for corecursion, which is given a silent (unfolding) transition semantics and the equivalent reduction semantics:

$$(\mathsf{corec}\ X(\overline{y}).P)\ \overline{c} \xrightarrow{\tau} P\{\overline{c}/\overline{y}\}\{(\mathsf{corec}\ X(\overline{y}).P)/X\}$$

The typing rules for our π-calculus processes are given in Fig. 6.3, summarizing the process typing rules that we have seen throughout this document in Sections 3.1 (non-dependent versions of $\forall$ and $\exists$) and 4.4 and containing the typing rules for corecursive process definitions, which introduce a small modification to the typing judgment, written as $\Psi;\Gamma;\Delta \Rightarrow_\eta P :: z{:}A$. The judgment denotes that process P offers the session behavior A along channel z, when composed with the (linear) session behaviors specified in Δ, with the (unrestricted) session behaviors specified in Γ and where η is a mapping from (corecursive) type variables to the typing contexts under which the definition is be well-formed (in order to preserve linearity).

To make clearer the use of the mapping η, let us now consider the right rule for coinductive sessions, which types (parameterized) corecursive process definitions:

$$\frac{\Psi;\Gamma;\Delta \Rightarrow_{\eta'} P :: c{:}A \quad \eta' = \eta[X(\overline{y}) \mapsto \Psi;\Gamma;\Delta \Rightarrow c{:}\nu Y.A]}{\Psi;\Gamma;\Delta \Rightarrow_\eta (\mathsf{corec}\ X(\overline{y}).P\{\overline{y}/\overline{z}\})\ \overline{z} :: c{:}\nu Y.A}\ (\nu\mathsf{R})$$

In the rule above, process P may use the recursion variable X and refer to the parameter list $\overline{y}$, which is instantiated with the list of (distinct) names $\overline{z}$ which may occur in Ψ,Δ,Γ,c or functional values (for which we assume an implicit sorting, for simplicity). Moreover, we keep track of the contexts Ψ, Γ and Δ in which the corecursive definition is made, the channel name along which the coinductive behavior is offered and the type associated with the corecursive behavior, by extending the mapping η with a binding for X with the appropriate information.

This is necessary because, intuitively, each occurrence of the corecursion variable stands for P itself (modulo the parameter instantiations) and therefore we must check that the necessary ambient session behaviors are available for P to execute in a type correct way, respecting linearity. P itself simply offers along channel c the session behavior A (which is an open type). To type the corecursion variable we use the following rule:

$$\frac{\eta(X(\overline{y})) = \Psi;\Gamma;\Delta \Rightarrow d{:}\nu Y.A \quad \rho = \{\overline{z}/\overline{y}\}}{\rho(\Psi);\rho(\Gamma);\rho(\Delta) \Rightarrow_\eta X(\overline{z}) :: \rho(d){:}Y}\ (\textsc{var})$$

We type a process corecursion variable X by looking up in η the binding for X, which references the typing environments Ψ, Γ and Δ under which the corecursive definition is well-defined, the coinductive type associated with the corecursive behavior and the channel name d along which the behavior is offered. The corecursion variable X is typed with the type variable Y if the parameter instantiation is able to satisfy the typing signature (by renaming available linear and unrestricted resources and term variables and matching them with the specification of Ψ, Γ and Δ). We also allow for the offered session channel to be a parameter of the corecursion. Finally, the left rule for coinductive session types silently unfolds the type:

$$\frac{\Psi;\Gamma;\Delta, c{:}A\{\nu X.A/X\} \Rightarrow_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\nu X.A \Rightarrow_\eta Q :: d{:}D}\ (\nu\mathsf{L})$$

The three rules discussed above play the role of the alternative typing rules for recursion presented in Chapter 5, where the mapping η and the rule for corecursion variables mimic the contextual aspect of the monad. To illustrate our corecursor, we can encode an ascending stream of naturals, starting at 0, as:

$$\mathsf{nats} \triangleq (\mathsf{corec}\ X(n,c).c\langle n\rangle.X(n+1,c))\ 0\ c$$

with the following typing derivation:

$$\dfrac{\dfrac{\cdot \Vdash 0 : \mathsf{nat} \qquad \dfrac{\eta(X(n,c)) = \cdot;\cdot;\cdot \Rightarrow c{:}\nu Y.\mathsf{nat} \wedge Y}{\cdot;\cdot;\cdot \Rightarrow_\eta X(1,c) :: c{:}Y}\ (\mathrm{VAR})}{\cdot;\cdot;\cdot \Rightarrow_\eta c\langle 0\rangle.X(1,c) :: c{:}\mathsf{nat} \wedge Y}\ \wedge\mathsf{R} \qquad \eta = X(n,c) \mapsto (\cdot;\cdot;\cdot \Rightarrow c{:}\nu Y.\mathsf{nat} \wedge Y)}{\cdot;\cdot;\cdot \Rightarrow (\mathsf{corec}\ X(n,c).c\langle n\rangle.X(n+1,c))\ 0\ c :: c{:}\nu Y.\mathsf{nat} \wedge Y}$$

6.1.2.2 The Logical Predicate

Since our calculus also includes λ-terms, we must define the usual notion of candidate and logical predicate for these functional terms, including for terms of monadic type. This makes the definition of the logical predicate mutually inductive, given that we need to simultaneously define the predicate for processes and λ-terms (we appeal to the functional predicate in the cases of value communication and to the process predicate in the case for the contextual monadic type). For λ-terms, we specify only the cases for the monadic type.

The particulars of the logical predicate for the coinductive setting essentially consist of the way in which we handle type variables and, naturally, the definition of the logical predicate for coinductive session types. Unlike in the polymorphic setting, where a type variable could be instantiated with *any* type, a type variable in a coinductive session type always stands for the coinductive session type itself. Thus, we no longer need to track which particular type we are instantiating a type variable with and only need to extend the predicate with a single mapping ω, which maps type variables to reducibility candidates (at the appropriate type), encoding the unfolding of the coinductive type in the logical predicate itself.

The definition of candidate for λ-terms is the expected one, essentially consisting of a functional analogue of Definition 9 (which is unchanged for processes), requiring well-typedness, closure

($\wedge$R)
$$\frac{\Psi \Vdash M{:}\tau \quad \Psi;\Gamma;\Delta \Rightarrow_\eta P :: c{:}A}{\Psi;\Gamma;\Delta \Rightarrow_\eta c\langle M\rangle.P :: c{:}\tau \wedge A}$$

($\wedge$L)
$$\frac{\Psi, x{:}\tau;\Gamma;\Delta, c{:}A \Rightarrow_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\tau \wedge A \Rightarrow_\eta c(x).Q :: d{:}D}$$

($\supset$R)
$$\frac{\Psi, x{:}\tau;\Gamma;\Delta \Rightarrow_\eta P :: c{:}A}{\Psi;\Gamma;\Delta \Rightarrow_\eta c(x).P :: c{:}\tau \supset A}$$

($\supset$L)
$$\frac{\Psi \Vdash M{:}\tau \quad \Psi;\Gamma;\Delta, c{:}A \Rightarrow_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\tau \supset A \Rightarrow_\eta c\langle M\rangle.Q :: d{:}D}$$

($\mathbf{1}$L)
$$\frac{\Psi;\Gamma;\Delta \Rightarrow_\eta P :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\mathbf{1} \Rightarrow_\eta c().P :: d{:}D}$$

($\otimes$R)
$$\frac{\Psi;\Gamma;\Delta_1 \Rightarrow_\eta P_1 :: x{:}A \quad \Psi;\Gamma;\Delta_2 \Rightarrow_\eta P_2 :: c{:}B}{\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow_\eta (\boldsymbol{\nu} x)c\langle x\rangle.(P_1 \mid P_2) :: c{:}A \otimes B}$$

($\otimes$L)
$$\frac{\Psi;\Gamma;\Delta, x{:}A, c{:}B \Rightarrow_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}A \otimes B \Rightarrow_\eta c(x).Q :: d{:}D}$$

($\multimap$R)
$$\frac{\Psi;\Gamma;\Delta, x{:}A \Rightarrow_\eta P :: c{:}B}{\Psi;\Gamma;\Delta \Rightarrow_\eta c(x).P :: c{:}A \multimap B}$$

($\multimap$L)
$$\frac{\Psi;\Gamma;\Delta_1 \Rightarrow_\eta Q_1 :: x{:}A \quad \Psi;\Gamma;\Delta_2, c{:}B \Rightarrow_\eta Q_2 :: d{:}D}{\Psi;\Gamma;\Delta_1,\Delta_2, c{:}A \multimap B \Rightarrow_\eta (\boldsymbol{\nu} x)c\langle x\rangle.(Q_1 \mid Q_2) :: d{:}D}$$

($\mathbf{1}$R)
$$\frac{}{\Psi;\Gamma;\cdot \Rightarrow_\eta c\langle\rangle.\mathbf{0} :: c{:}\mathbf{1}}$$

(!R)
$$\frac{\Psi;\Gamma;\cdot \Rightarrow_\eta P :: x{:}A}{\Psi;\Gamma;\cdot \Rightarrow_\eta (\boldsymbol{\nu} u)c\langle u\rangle.!u(x).P :: c{:}!A}$$

(!L)
$$\frac{\Psi;\Gamma, u{:}A;\Delta \Rightarrow_\eta P :: d{:}D}{\Psi;\Gamma;\Delta, c{:}!A \Rightarrow_\eta c(u).P :: d{:}D}$$

(COPY)
$$\frac{\Psi;\Gamma, u{:}A;\Delta, x{:}A \Rightarrow_\eta P :: d{:}D}{\Psi;\Gamma, u{:}A;\Delta \Rightarrow_\eta (\boldsymbol{\nu} x)u\langle x\rangle.P :: d{:}D}$$

($\oplus$R)
$$\frac{\Psi;\Gamma;\Delta \Rightarrow_\eta P :: c{:}A_i}{\Psi;\Gamma;\Delta \Rightarrow_\eta c.l_i;P :: c{:}\oplus\{\overline{l_j{:}A_j}\}}$$

($\oplus$L)
$$\frac{\Psi;\Gamma;\Delta, c{:}A_1 \Rightarrow_\eta Q_1 :: d{:}D \ \ldots \ \Psi;\Gamma;\Delta, c{:}A_n \Rightarrow_\eta Q_n :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\oplus\{\overline{l_j{:}A_j}\} \Rightarrow_\eta c.\mathsf{case}(\overline{l_j \Rightarrow Q_j}) :: d{:}D}$$

(&R)
$$\frac{\Psi;\Gamma;\Delta \Rightarrow_\eta P_1 :: c{:}A_1 \ \ldots \ \Psi;\Gamma;\Delta \Rightarrow_\eta P_n :: c{:}A_n}{\Psi;\Gamma;\Delta \Rightarrow_\eta c.\mathsf{case}(\overline{l_j \Rightarrow P_j}) :: c{:}\&\{\overline{l_j{:}A_j}\}}$$

(&L)
$$\frac{\Psi;\Gamma;\Delta, c{:}A_i \Rightarrow_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\&\{\overline{l_j{:}A_j}\} \Rightarrow_\eta c.l_i;Q :: d{:}D}$$

(νL)
$$\frac{\Psi;\Gamma;\Delta, c{:}A\{\nu X.A/X\} \Rightarrow_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta, c{:}\nu X.A \Rightarrow_\eta Q :: d{:}D}$$

(VAR)
$$\frac{\eta(X(\overline{y})) = \Psi;\Gamma;\Delta \Rightarrow d{:}\nu Y.A \quad \rho = \{\overline{z}/\overline{y}\}}{\rho(\Psi);\rho(\Gamma);\rho(\Delta) \Rightarrow_\eta X(\overline{z}) :: \rho(d){:}Y}$$

(νR)
$$\frac{\Psi;\Gamma;\Delta \Rightarrow_{\eta'} P :: c{:}A \quad \eta' = \eta[X(\overline{y}) \mapsto \Psi;\Gamma;\Delta \Rightarrow c{:}\nu Y.A]}{\Psi;\Gamma;\Delta \Rightarrow_\eta (\mathsf{corec}\ X(\overline{y}).P\{\overline{y}/\overline{z}\})\ \overline{z} :: c{:}\nu Y.A}$$

(ID)
$$\frac{}{\Psi;\Gamma;d{:}A \Rightarrow_\eta [d \leftrightarrow c] :: c{:}A}$$

(CUT)
$$\frac{\Psi;\Gamma;\Delta_1 \Rightarrow_\eta P :: x{:}A \quad \Psi;\Gamma;\Delta_2, x{:}A \Rightarrow_\eta Q :: d{:}D}{\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow_\eta (\boldsymbol{\nu} x)(P \mid Q) :: d{:}D}$$

(CUT$^!$)
$$\frac{\Psi;\Gamma;\cdot \Rightarrow_\eta P :: x{:}A \quad \Psi;\Gamma, u{:}A;\Delta \Rightarrow Q :: d{:}D}{\Psi;\Gamma;\Delta \Rightarrow_\eta (\boldsymbol{\nu} u)(!u(x).P \mid Q) :: d{:}D}$$

(SPAWN)
$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi \Vdash M : \{\overline{a_i{:}A_i} \vdash a{:}A\} \quad \Psi;\Gamma;\Delta', a{:}A \Rightarrow_\eta Q :: z{:}C}{\Psi;\Gamma;\Delta,\Delta' \Rightarrow_\eta \mathsf{spawn}(M;\overline{a_i};a.Q) :: z{:}C}$$

Figure 6.3: Typing Rules for Higher-Order Processes

$$\begin{array}{ll}
\mathcal{L}^{\omega}[z{:}\nu X.A] & \triangleq \bigcup\{\Psi \in \mathbb{R}[-{:}\nu X.A] \mid \Psi \subseteq \mathcal{L}^{\omega[X\mapsto\Psi]}[z{:}A]\} \\
\mathcal{L}^{\omega}[z{:}X] & \triangleq \omega(X)(z) \\
\mathcal{L}^{\omega}[z{:}\mathbf{1}] & \triangleq \{P \mid \forall P'.(P \xRightarrow{z\langle\rangle} P' \wedge P' \not\rightarrow) \Rightarrow P' \equiv_{!} \mathbf{0}\} \\
\mathcal{L}^{\omega}[z{:}A \multimap B] & \triangleq \{P \mid \forall P' y.(P \xRightarrow{z(y)} P') \Rightarrow \forall Q \in \mathcal{L}^{\omega}[y{:}A].(\nu y)(P' \mid Q) \in \mathcal{L}^{\omega}[z{:}B]\} \\
\mathcal{L}^{\omega}[z{:}A \otimes B] & \triangleq \{P \mid \forall P' y.(P \xRightarrow{\overline{(\nu y)z\langle y\rangle}} P') \Rightarrow \\
 & \quad \exists P_1, P_2.(P' \equiv_{!} P_1 \mid P_2 \wedge P_1 \in \mathcal{L}^{\omega}[y{:}A] \wedge P_2 \in \mathcal{L}^{\omega}[z{:}B])\} \\
\mathcal{L}^{\omega}[z{:}!A] & \triangleq \{P \mid \forall P'.(P \xRightarrow{\overline{(\nu u)z\langle u\rangle}} P') \Rightarrow \exists P_1.(P' \equiv_{!} !z(y).P_1 \wedge P_1 \in \mathcal{L}^{\omega}[y{:}A])\} \\
\mathcal{L}^{\omega}[z{:}\,\&\,\{\overline{l_i \Rightarrow A_i}\}] & \triangleq \{P \mid \bigwedge_i (\forall P'.(P \xRightarrow{z.l_i} P') \Rightarrow P' \in \mathcal{L}^{\omega}[z{:}A_i])\} \\
\mathcal{L}^{\omega}[z{:}\oplus\{\overline{l_i \Rightarrow A_i}\}] & \triangleq \{P \mid \bigwedge_i (\forall P'.(P \xRightarrow{\overline{z.l_i}} P') \Rightarrow P' \in \mathcal{L}^{\omega}[z{:}A_i])\} \\
\mathcal{L}^{\omega}[z{:}\tau \wedge A] & \triangleq \{P \mid \forall P'.(P \xRightarrow{z\langle M\rangle} P') \Rightarrow (M \in \mathcal{L}[\tau] \wedge P' \in \mathcal{L}^{\omega}[z{:}A])\} \\
\mathcal{L}^{\omega}[z{:}\tau \supset A] & \triangleq \{P \mid \forall P', M.(M \in \mathcal{L}[\tau] \wedge P \xRightarrow{z(M)} P') \Rightarrow P' \in \mathcal{L}^{\omega}[z{:}A]\}
\end{array}$$

Figure 6.4: Logical Predicate for Coinductive Session Types - Closed Processes

under reduction and head expansion. The logical predicate, as before, is defined inductively on types. The only presented clause in the functional case is that for the contextual monadic type, which requires terms to reduce to a monadic value that is in the (process-level) predicate at the appropriate type:

$$\mathcal{L}^{\omega}[\{\overline{a{:}A_i} \Rightarrow c{:}A\}] \;\triangleq\; \{M \mid M \Longrightarrow c \leftarrow \{P\} \leftarrow \overline{a} \text{ and } P \in \mathcal{L}^{\omega}[\mathsf{shd}(\overline{a{:}A_i}); \mathsf{lin}(\overline{a{:}A_i}) \Rightarrow_{\emptyset} c{:}A]\}$$

For processes, the significant cases are those that pertain to type variables and coinductive session types in the inductive base case. We refer to $\mathbb{R}[-{:}A]$ below as the collection of all sets of reducibility candidates at (closed) type A. We state that the mapping ω is compatible with the mapping η if for each binding in η of the form $X \mapsto \Psi; \Gamma; \Delta \Rightarrow c{:}\nu Y.A$, ω maps Y to $\mathcal{L}[c{:}\nu Y.A]$.

Definition 13 (Logical Predicate - Open Processes)**.** *Given $\Psi; \Gamma; \Delta \Rightarrow_{\eta} T$ with a non-empty left hand side environment, we define $\mathcal{L}^{\omega}[\Psi; \Gamma; \Delta \Rightarrow T]$, where ω is a mapping from type variables to reducibility candidates compatible with η, as the set of processes inductively defined as:*

$$\begin{array}{lll}
P \in \mathcal{L}^{\omega}[\Psi, x{:}\tau; \Gamma; \Delta \Rightarrow_{\eta} T] & \textit{iff} & \forall M \in \mathcal{L}[\tau].P\{M/x\} \in \mathcal{L}^{\omega}[\Psi; \Gamma; \Delta \Rightarrow_{\eta} T] \\
P \in \mathcal{L}^{\omega}[\Gamma; \Delta, y{:}A \Rightarrow_{\eta} T] & \textit{iff} & \forall R \in \mathcal{L}^{\omega}[y{:}A].(\nu y)(R \mid P) \in \mathcal{L}^{\omega}[\Gamma; \Delta \Rightarrow_{\eta} T] \\
P \in \mathcal{L}^{\omega}[\Gamma, u{:}A; \Delta \Rightarrow_{\eta} T] & \textit{iff} & \forall R \in \mathcal{L}^{\omega}[y{:}A].(\nu u)(!u(y).R \mid P) \in \mathcal{L}^{\omega}[\Gamma; \Delta \Rightarrow_{\eta} T]
\end{array}$$

The definition of the logical interpretation for open processes inductively composes an open process with the appropriate witnesses in the logical interpretation at the types specified in the three contexts, following the development of the previous section.

Definition 14 (Logical Predicate - Closed Processes)**.** *For any type $T = z{:}A$ we define $\mathcal{L}^{\omega}[T]$ as the set of all processes P such that $P \Downarrow$ and $\cdot; \cdot; \cdot \Rightarrow_{\eta} P :: T$ satisfying the rules of Fig. 6.4.*

We define the logical interpretation of a coinductive session type $\nu X.A$ as the union of all reducibility candidates Ψ of the appropriate coinductive type that are in the predicate for (open) type A, when X is mapped to Ψ itself. The technical definition is somewhat intricate, but the key observation is that for *open types*, we may view our logical predicate as a mapping between sets of reducibility candidates, of which the interpretation for coinductive session types turns out to be a greatest fixpoint.

In order to show the fundamental theorem of logical relations for coinductive session types, we must first establish the fixpoint property mentioned above. More precisely, we define an operator $\phi_A(s)$ from and to reducibility candidates at the open type A via the logical predicate at the type.

Before establishing the fixpoint property, we show that our logical predicate is a reducibility candidate (as before).

Theorem 32 (Logical Predicate is a Reducibility Candidate)**.** *The following statements hold:*

- $\mathcal{L}[\tau] \subseteq \mathcal{R}[\tau]$.
- $\mathcal{L}^{\omega}[z{:}A] \subseteq \mathcal{R}[z{:}A]$

Proof. The proof proceeds by simultaneous induction on τ and A due to the dependencies between the two language layers. Note that since all terms in $\mathcal{L}[\tau]$ are terminating and well-typed by definition, we only need to establish points (3) and (4). The proof is identical to that of Theorem 28, so we show only the new cases.

Case: $\tau = \{\overline{a{:}A_i} \vdash c{:}A\}$

Let $M \in \mathcal{L}^{\omega}[\{\overline{a{:}A_i} \vdash c{:}A\}]$, if $M \longrightarrow M'$, by definition we know that since $M \Longrightarrow a \leftarrow \{P\} \leftarrow \overline{a_i}$ with $P \in \mathcal{L}[\mathsf{shd}(\overline{a{:}A_i}); \mathsf{lin}(\overline{a{:}A_i}) \vdash c{:}A]$, $M' \Longrightarrow a \leftarrow \{P\} \leftarrow \overline{a_i}$ and so (3) is satisfied.

Let $M \in \mathcal{L}^{\omega}[\{\overline{a{:}A_i} \vdash c{:}A\}]$ and $M_0 \longrightarrow M$. Since $M_0 \longrightarrow M \Longrightarrow a \leftarrow \{P\} \leftarrow \overline{a_i}$, with $P \in \mathcal{L}[\mathsf{shd}(\overline{a{:}A_i}); \mathsf{lin}(\overline{a{:}A_i}) \vdash c{:}A]$, we have that $M_0 \in \mathcal{L}^{\omega}[\{\overline{a{:}A_i} \vdash c{:}A\}]$, satisfying (4).

Case: $A = \nu X.A'$

$P \in \mathcal{L}^{\omega}[z{:}\nu X.A']$ is defined as the union of all sets ψ of reducibility candidates at type A satisfying $\psi \subseteq \mathcal{L}^{\omega[X \mapsto \psi]}[z{:}A']$, since these by definition satisfy the required conditions, their union must also satisfy the conditions, and we are done.

□

Definition 15. *Let $\nu X.A$ be a strictly positive type. We define the operator $\phi_A : \mathbb{R}[- : A] \to \mathbb{R}[- : A]$ as:*

$$\phi_A(s) \triangleq \mathcal{L}^{\omega[X \mapsto s]}[z{:}A]$$

To show that the interpretation for coinductive session types is a greatest fixpoint of this operator, we must first establish a monotonicity property.

Lemma 28 (Monotonicity of ϕ_A)**.** *The operator ϕ_A is monotonic, that is, $s \subseteq s'$ implies $\phi_A(s) \subseteq \phi_A(s')$, for any s and s' in $\mathbb{R}[-{:}A]$.*

Proof. If X is not free in X then $\phi_A(s) = \phi_A(s')$ and so the condition holds. If X is free in A, we proceed by induction on A (we write ω' for ω extended with a mapping for X with s or s').

Case: $A = X$

$\mathcal{L}^{\omega'}[z{:}X] = \omega'(X)$ by definition
$\phi_A(s) = s$ and $\phi_A(s') = s'$ and so $\phi_A(s) \subseteq \phi_A(s')$ by assumption

Case: $A = \nu Y.A'$

$\mathcal{L}^{\omega'}[z{:}\nu Y.A'] = \bigcup\{\phi \in \mathbb{R}[-{:}\nu Y.A'] \mid \phi \subseteq \mathcal{L}^{\omega'[Y\mapsto\phi]}[z{:}A']\}$ by definition
$\mathcal{L}^{\omega[X\mapsto s, Y\mapsto\phi]}[z{:}A'] \subseteq \mathcal{L}^{\omega[X\mapsto s', Y\mapsto\phi]}[z{:}A']$ by i.h.
To show: $\mathcal{L}^{\omega[X\mapsto s]}[z{:}\nu Y.A'] \subseteq \mathcal{L}^{\omega[X\mapsto s']}[z{:}\nu Y.A']$
S.T.S: $\bigcup\{\phi \in \mathbb{R} \mid \phi \subseteq \mathcal{L}^{\omega[X\mapsto s, Y\mapsto\phi]}[z{:}A']\} \subseteq \bigcup\{\phi \in \mathbb{R} \mid \phi \subseteq \mathcal{L}^{\omega[X\mapsto s', Y\mapsto\phi]}[z{:}A']\}$
Let $S = \bigcup\{\phi \in \mathbb{R} \mid \phi \subseteq \mathcal{L}^{\omega[X\mapsto s, Y\mapsto\phi]}[z{:}A']\}$
S is the largest $\mathcal{R}[-{:}z{:}\nu Y.A']$ contained in $\mathcal{L}^{\omega[X\mapsto s, Y\mapsto\phi]}[z{:}A']$ by union
Let $S' = \bigcup\{\phi \in \mathbb{R} \mid \phi \subseteq \mathcal{L}^{\omega[X\mapsto s', Y\mapsto\phi]}[z{:}A']\}$
S' is the largest $\mathcal{R}[-{:}z{:}\nu Y.A']$ contained in $\mathcal{L}^{\omega[X\mapsto s', Y\mapsto\phi]}[z{:}A']$ by union
$S' \subseteq \mathcal{L}^{\omega[X\mapsto s', Y\mapsto\phi]}[z{:}A']$ by the reasoning above
$S \subseteq \mathcal{L}^{\omega[X\mapsto s, Y\mapsto\phi]}[z{:}A']$ by the reasoning above
$S \subseteq \mathcal{L}^{\omega[X\mapsto s', Y\mapsto\phi]}[z{:}A']$ using the i.h. and transitivity
Since S' is the largest such set, $S \subseteq S'$

Remaining cases follow straightforwardly from the i.h.

□

Theorem 33 (Greatest Fixpoint). *$\mathcal{L}^{\omega}[z{:}\nu X.A]$ is a greatest fixpoint of ϕ_A.*

Proof.

Let $G = \mathcal{L}^{\omega}[\nu X.A]$
To show:$\phi_A(G) = G$
First, we show $G \subseteq \phi_A(G)$:
If $P \in G$ then $P \in \Psi$ for some $\Psi \in \mathbb{R}[-{:}A]$ such that $\Psi \subseteq \mathcal{L}^{\omega[X\mapsto\Psi]}[z{:}A]$ by def.
$\Psi \subseteq \mathcal{L}^{\omega[X\mapsto\Psi]}[z{:}A]$ by definition
$\Psi \subseteq G$ G is the union of all such Ψ
$\Psi \subseteq \mathcal{L}^{\omega[X\mapsto G]}[z{:}A] = \phi_A(G)$ by monotonicity and transitivity
$P \in \phi_A(G)$ by the reasoning above
$G \subseteq \phi_A(G)$ since $P \in G$ implies $P \in \phi_A(G)$
$\phi_A(G) = \mathcal{L}^{\omega[X\mapsto G]}[z{:}A] \subseteq \mathcal{L}^{\omega[X\mapsto\phi_A(G)]}[z{:}A]$ by monotonicity
$\phi_A(G) \subseteq \mathcal{L}^{\omega}[z{:}\mu X.A] = G$ since G is the union of all Ψ s.t. $\Psi \subseteq \mathcal{L}^{\omega[X\mapsto\Psi]}[z{:}A]$
If $\psi \in \mathbb{R}[-{:}\nu X.A]$ verifies $\psi = \mathcal{L}^{\omega[X\mapsto\psi]}[A]$ then $\psi \subseteq G$, so G is a *greatest* fixpoint.

□

Having characterized the interpretation of coinductive session types, we now characterize the compositional nature of substitution of type variables in types with our interpretation of open types. For this we require the following theorem, reminiscent of Theorem 29 for the polymorphic setting.

Theorem 34. *Let $\nu X.A$ be a well formed type:*

$$P \in \mathcal{L}^{\omega}[z{:}A\{\nu X.A/X\}] \textit{ iff } P \in \mathcal{L}^{\omega[X \mapsto \mathcal{L}^{\omega}[-:\nu X.A]]}[z{:}A]$$

Proof. By induction on the structure A.

Case: $A = X$

$$\begin{array}{lr}
P \in \mathcal{L}^{\omega}[z{:}X\{\nu X.A/X\}] & \text{assumption} \\
P \in \mathcal{L}^{\omega}[z{:}\nu X.A] & \text{by substitution} \\
\mathcal{L}^{\omega[X \mapsto \mathcal{L}^{\omega}[-:\nu X.A]]}[z{:}X] = \mathcal{L}^{\omega}[z{:}\nu X.A] & \text{by definition}
\end{array}$$

$$\begin{array}{lr}
P \in \mathcal{L}^{\omega[X \mapsto \mathcal{L}^{\omega}[-:\mu X.A]]}[z{:}X] & \text{assumption} \\
P \in \mathcal{L}^{\omega}[z{:}\nu X.A] & \text{by definition}
\end{array}$$

Case: $A = \nu Y.A'$

$$\begin{array}{lr}
P \in \mathcal{L}^{\omega}[z{:}\nu Y.A'\{\nu X.A/X\}] & \text{assumption} \\
\text{To show: } P \in \mathcal{L}^{\omega[X \mapsto \mathcal{L}^{\omega}[-:\nu X.A]]}[z{:}\nu Y.A'] & \\
P \in \cup\{\psi \in \mathbb{R} \mid \psi \subseteq \mathcal{L}^{\omega[Y \mapsto \psi]}[z{:}A'\{\nu X.A/X\}]\} & \text{by definition} \\
P \in \mathcal{L}^{\omega[X \mapsto \mathcal{L}^{\omega}[-:\nu X.A], Y \mapsto \mathcal{L}^{\omega}[-:\nu Y.A']]}[z{:}A'] & \text{by i.h.} \\
P \in \mathcal{L}^{\omega[X \mapsto \mathcal{L}^{\omega}[-:\nu X.A]]}[z{:}\nu Y.A'] & \text{by the fixpoint property}
\end{array}$$

Other cases are identical to the one above, or just follow directly from definitions and the i.h.

□

We make use of the technique of employing process representatives of contexts, as done in the development of the previous section (Def. 12, Prop. 4 and Lemma 25), but accounting for the terms that may occur in processes.

Definition 16. *Let $\Gamma = u_1{:}B_1, \ldots, u_k{:}B_k$ and $\Delta = x_1{:}A_1, \ldots, x_n{:}A_n$ be a non-linear and linear typing environment respectively, $\Psi = x_1{:}\tau_1, \ldots, x_l{:}\tau_l$ a term typing environment and ψ any substitution compatible with Ψ, mapping any variable $x_i \in \Psi$ to an arbitrary term in the logical predicate $\mathcal{L}[\tau_i]$. Let ω be a mapping in the sense above. Letting $I = \{1, \ldots, k\}$ and $J = \{1, \ldots, n\}$, we define the sets of processes $\mathcal{C}^{\omega}_{\Gamma,\Psi}$ and $\mathcal{C}^{\omega}_{\Delta,\Psi}$ as:*

$$\mathcal{C}^{\omega}_{\Gamma,\Psi} \stackrel{def}{=} \{ \prod_{i \in I} !u_i(y_i).\hat{\psi}(R_i) \mid R_i \in \mathcal{L}^{\omega}[y_i{:}B_i]\} \qquad \mathcal{C}^{\omega}_{\Delta,\Psi} \stackrel{def}{=} \{ \prod_{j \in J} \hat{\psi}(Q_j) \mid Q_j \in \mathcal{L}^{\omega}[x_j{:}A_j]\}$$

Proposition 5. *Let Γ and Δ be a non-linear and a linear typing environment, respectively. For all $Q \in \mathcal{C}^{\omega}_{\Gamma,\Psi}$ and for all $R \in \mathcal{C}^{\omega}_{\Delta,\Psi}$ we have $Q \Downarrow$ and $R \Downarrow$. Moreover, $Q \not\longrightarrow$*

Theorem 35. *Let $\Psi;\Gamma;\Delta \Rightarrow_{\eta} P :: T$, with $G = u_1{:}B_1,\dots,u_k{:}B_k$ and $\Delta = x_1{:}A_1,\dots,x_n{:}A_n$. We have: $\hat{\psi}(P) \in \mathcal{L}^{\omega}[\Gamma;\Delta \Rightarrow T]$ iff $\forall Q \in \mathcal{C}^{\omega}_{\Gamma,\Psi}, \forall R \in \mathcal{C}^{\omega}_{\Delta,\Psi}, (\nu\overline{u},\overline{x})(P \mid Q \mid R) \in \mathcal{L}^{\omega}[T]$.*

Proof. Straightforward from the definition of the inductive case of the logical predicate and Def. 6.1.2.2. □

The following renaming lemma is fundamental for our proof of the fundamental theorem, specifically in the coinductive case.

Lemma 29 (Renaming)**.** *Let A be a well-formed type. If $P \in \mathcal{L}^{\omega}[z{:}A]$ and $x \notin fn(P)$ then $P\{x/z\} \in \mathcal{L}^{\omega}[x{:}A]$.*

Proof. Immediate from the definition of the logical predicate. □

We can now show the fundamental theorem of logical relations for our language of coinductive session types. Before going into the details of the proof, we outline its fundamental points. The main burden of proof, beyond the cases that are fundamentally identical to the proof of the Theorem 30, are the cases pertaining to the νR and νL rules. The case for the νL rule makes crucial use of the fixpoint and unrolling properties, whereas the case for νR requires us to proceed by coinduction: we produce a set of processes $\mathcal{C}_P$, containing our recursive definition P, show that $\mathcal{C}_P$ is a reducibility candidate at $\nu X.A$ and that $\mathcal{C}_P \subseteq \mathcal{L}^{\omega[X \mapsto \mathcal{C}_P]}[z{:}A]$. This is a sufficient condition since we know that $\mathcal{L}[c{:}\nu X.A]$ is the largest such set. Showing that $\mathcal{C}_P$ satisfies this condition relies crucially on coregular recursion and guardedness.

Theorem 36 (Fundamental Theorem)**.** *If $\Psi \Vdash M{:}\tau$ and $\Psi;\Gamma;\Delta \Rightarrow_{\eta} P :: z{:}A$ then for any mapping ψ s.t. $x{:}\tau_0 \in \Psi$ iff $\psi(x) \subseteq \mathcal{L}[\tau_0]$, for any mapping ω consistent with η and for any mapping ρ s.t. $X \mapsto \Psi_0;\Gamma_0;\Delta_0 \Rightarrow c_0{:}\nu Z.B \in \eta$ iff $\rho(X) \subseteq \mathcal{L}^{\omega}[\Psi_0;\Gamma_0;\Delta_0 \Rightarrow c_0{:}\nu Z.B]$ we have that $\hat{\psi}(M) \subseteq \mathcal{L}[\tau]$ and $\hat{\psi}(\hat{\rho}(P)) \subseteq \mathcal{L}^{\omega}[\Psi;\Gamma;\Delta \Rightarrow z{:}A]$.*

Proof. The proof proceeds by mutual induction on the given typing derivation. We show only the new cases pertaining to coinductive types, variables and the spawn construct.

Case:

$$\frac{\Psi;\Gamma;\Delta, x{:}A\{\nu X.A/X\} \Rightarrow_{\eta} P :: T}{\Psi;\Gamma;\Delta, x{:}\nu X.A \Rightarrow_{\eta} P :: T}\ \nu L$$

$\hat{\psi}(\hat{\rho}(P)) \in \mathcal{L}^{\omega}[\Gamma;\Delta, x{:}\nu X.A \Rightarrow T]$ by i.h.

Pick any $G \in \mathcal{C}^{\Psi}_{\Gamma}$, $D \in \mathcal{C}^{\Psi}_{\Delta}$:

$(\nu\overline{u},\overline{x})(\hat{\psi}(\hat{\rho}(P)) \mid G \mid D \mid \mathcal{L}^{\omega}[x{:}A\{\nu X.A/X\}]) \in \mathcal{L}^{\omega}[T]$ by Theorem 35.

S.T.S: $(\nu\overline{u},\overline{x})(\hat{\psi}(\hat{\rho}(P)) \mid G \mid D \mid \mathcal{L}^{\omega}[x{:}\nu X.A]) \in \mathcal{L}^{\omega}[T]$

Pick $R \in \mathcal{L}^{\omega}[x{:}\nu X.A]$:

S.T.S: $(\nu\overline{u},\overline{x})(\hat{\psi}(\hat{\rho}(P)) \mid G \mid D \mid R) \in \mathcal{L}^{\omega}[T]$

$R \in \mathcal{L}^{\omega[X\mapsto\mathcal{L}^{\omega}[x{:}\nu X.A]]}[x{:}A]$ by fixpoint theorem

$R \in \mathcal{L}^{\omega}[x{:}A\{\nu X.A/X\}]$ by Thrm. 34

$(\nu\overline{u},\overline{x})(\hat{\psi}(\hat{\rho}(P)) \mid G \mid D \mid R) \in \mathcal{L}^{\omega}[T]$ by the statement above and the i.h.

Case:

(SPAWN)

$$\frac{\Delta = \mathsf{lin}(\overline{a_i{:}A_i}) \quad \mathsf{shd}(\overline{a_i{:}A_i}) \subseteq \Gamma \quad \Psi \Vdash M : \{\overline{a_i{:}A_i} \Rightarrow c{:}A\} \quad \Psi;\Gamma;\Delta',c{:}A \Rightarrow P :: d{:}C}{\Psi;\Gamma;\Delta,\Delta' \Rightarrow \mathsf{spawn}(M;\overline{a_i};c.P) :: z{:}C}$$

Pick any $G \in \mathcal{C}^{\Psi}_{\Gamma}$, $D' \in \mathcal{C}^{\Psi}_{\Delta'}$, $D \in \mathcal{C}^{\Psi}_{\Delta}$:

S.T.S.: $(\nu\overline{u},\overline{y},\overline{x})(G \mid D \mid D' \mid \mathsf{spawn}(\hat{\psi}(M);c.\hat{\psi}(\hat{\rho}(P)))) \in \mathcal{L}[d{:}C]$ by Theorem 35

$\hat{\psi}(\hat{\rho}(P)) \in \mathcal{L}[\Gamma;\Delta',c{:}A \vdash d{:}C]$ by i.h.

Pick any $Q \in \mathcal{L}[c{:}A]$:

$(\nu\overline{u},\overline{x},c)(\hat{\psi}(\hat{\rho}(P)) \mid G \mid D' \mid Q) \in \mathcal{L}[d{:}C]$ by Theorem 35.

$\hat{\psi}(M) \in \mathcal{L}[\{\Gamma;\Delta \Rightarrow c{:}A\}]$ by i.h.

$\hat{\psi}(M) \Longrightarrow c \leftarrow \{R\} \leftarrow \overline{d}$ with $R \in \mathcal{L}[\Gamma;\Delta \vdash c{:}A]$ by definition

$(\nu\overline{u},\overline{y})(G \mid D \mid R) \in \mathcal{L}[c{:}A]$ by Theorem 35.

$(\nu\overline{u},\overline{x},c)(\hat{\psi}(\hat{\rho}(P)) \mid G \mid D' \mid D \mid R) \in \mathcal{L}[d{:}C]$ by the statement above.

$(\nu\overline{u},\overline{y},\overline{x})(G \mid D \mid D' \mid \mathsf{spawn}(\hat{\psi}(M);c.\hat{\psi}(\hat{\rho}(P)))) \Longrightarrow (\nu\overline{u},\overline{x},c)(\hat{\psi}(\hat{\rho}(P)) \mid G \mid D' \mid D \mid R)$

by the operational semantics

$(\nu\overline{u},\overline{y},\overline{x})(G \mid D \mid D' \mid \mathsf{spawn}(\hat{\psi}(M);c.\hat{\psi}(\hat{\rho}(P)))) \in \mathcal{L}[d{:}C]$

by closure under ⇒

Case:

(VAR)

$$\frac{\eta(X(\overline{y})) = \Psi;\Gamma;\Delta \Rightarrow d{:}\nu Y.A \quad \rho' = \{\overline{z}/\overline{y}\}}{\rho'(\Psi);\rho'(\Gamma);\rho'(\Delta) \Rightarrow_{\eta} X(\overline{z}) :: \rho'(d){:}Y}$$

Follows directly from ω being compatible with η and the definition of ρ.

Case:

$(\nu\mathsf{R})$

$$\frac{\Psi;\Gamma;\Delta \Rightarrow_{\eta'} P :: c{:}A \quad \eta' = \eta[X(\overline{y}) \mapsto \Psi;\Gamma;\Delta \Rightarrow c{:}\nu Y.A]}{\Psi;\Gamma;\Delta \Rightarrow_{\eta} (\mathsf{corec}\ X(\overline{y}).P\{\overline{y}/\overline{z}\})\ \overline{z} :: c{:}\nu Y.A}$$

Let $Q = (\mathsf{corec}\ X(\overline{y}).P\{\overline{y}/\overline{z}\})\ \overline{z}$.

Pick any $G \in \mathcal{C}^{\Psi}_{\Gamma}$, $D \in \mathcal{C}^{\Psi}_{\Delta}$:

S.T.S: $(\boldsymbol{\nu}\overline{u},\overline{x})(\hat{\psi}(\hat{\rho}(Q)) \mid G \mid D) \in \mathcal{L}^{\omega}[c{:}\nu Y.A]$ by Theorem 35.

$\hat{\psi}(\hat{\rho'}(P)) \in \mathcal{L}^{\omega'}[\Psi;\Gamma;\Delta \Rightarrow c{:}A]$ for some ω' compatible with η', ρ' compatible with η'

by i.h.

$(\boldsymbol{\nu}\overline{u},\overline{x})(\hat{\psi}(\rho'(P)) \mid G \mid D) \in \mathcal{L}^{\omega'}[c{:}A]$ by Theorem 35.

We proceed by coinduction:
We define a set $\mathcal{C}(-) \in \mathbb{R}[-{:}\nu Y.A]$ such that $\mathcal{C}(c) \subseteq \mathcal{L}^{\omega[Y \mapsto \mathcal{C}]}[c{:}A]$ and show:
$(\nu \overline{u}, \overline{x})(\hat{\psi}(\hat{\rho}(Q)) \mid G \mid D) \in \mathcal{C}(c)$, concluding the proof.
For any compatible ψ,
Let $C_1 = \forall R \in \mathcal{C}^{\Psi}_{\Gamma}, S \in \mathcal{C}^{\Psi}_{\Delta}.U \equiv (\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q)))$
Let $C_2 = (U \in \mathcal{R}[-{:}\nu Y.A] \ \wedge \ U \Rightarrow (\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q))))$
Let $C_3 = ((\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q))) \Rightarrow U \wedge U \in \mathcal{R}[-{:}\nu Y.A])$
Let $\mathcal{C} = \{U \mid C_1 \ \vee \ C_2 \ \vee \ C_3\}$
S.T.S.: $\mathcal{C}(-) \in \mathbb{R}[-{:}\nu Y.A]$ and $\mathcal{C}(c) \subseteq \mathcal{L}^{\omega[Y \mapsto \mathcal{C}]}[c{:}A]$
Showing $\mathcal{C} \in \mathbb{R}[-{:}\nu Y.A]$:
(1) If $U \in \mathcal{C}$ then $\Rightarrow_{\eta}$ $U :: c{:}\nu Y.A$, for some c.
Either $U \in \mathcal{R}[-{:}\nu X.A]$, or $U \equiv (\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q)))$
(2) If $Q \in \mathcal{C}$ then $Q \Downarrow$
$\forall R \in \mathcal{C}^{\Psi}_{\Gamma}, S \in \mathcal{C}^{\Psi}_{\Delta}.(\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q))) \Downarrow$
we know that the shape of P ensures a guarding action before the recursion. Moreover, $\hat{\psi}(\hat{\rho}'(P)) \in \mathcal{L}^{\omega'}[\Psi; \Gamma; \Delta \Rightarrow c{:}A]$ by i.h., thus up to the recursive unfolding the process is in $\mathcal{L}$
Any other $U \in \mathcal{C}$ is in $\mathcal{R}[-{:}\nu Y.A]$ outright and so $Q \Downarrow$
(3) If $U \in \mathcal{C}$ and $U \Rightarrow U'$ then $U' \in \mathcal{C}$
The only case that does not follow directly from C_2 and C_3 is the recursive unfolding of $(\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q)))$
$(\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q))) \rightarrow (\nu \overline{a}(R \mid S \mid \hat{\psi}(\hat{\rho}(P\{Q/X(\overline{y})\}))))$ by red. semantics
$(\nu \overline{a}(R \mid S \mid \hat{\psi}(\hat{\rho}(P\{Q/X(\overline{y})\})))) \Downarrow$
$\hat{\psi}(\hat{\rho}'(P)) \in \mathcal{L}^{\omega'}[\Psi; \Gamma; \Delta \Rightarrow c{:}A]$, moreover, typing enforces guardedness in P, so a blocking action on c must take place before the distinction between $\hat{\psi}(\hat{\rho}'(P))$ and the recursive unfolding.
(4) If for all U_i such that $U \Rightarrow U_i$ we have $U_i \in \mathcal{C}$ then $U \in \mathcal{C}$
trivial by C_2 and C_3
Showing $\mathcal{C} \subseteq \mathcal{L}^{\omega[X \mapsto \mathcal{C}]}[c{:}A]$:
S.T.S.: $(\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}(Q))) \in \mathcal{L}^{\omega[Y \mapsto \mathcal{C}]}[c{:}A]$ by def. of $\mathcal{C}$
$(\nu \overline{a})(R \mid S \mid \hat{\psi}(\hat{\rho}'(P))) \in \mathcal{L}^{\omega'}[c{:}A]$ by Theorem 35, and the i.h.
Thus the crucial points are those in which the two processes above differ, which are the occurrences of the recursion variable. By typing we know that the recursion variable X can only typed by Y, and that in P there must be a guarding action on c before the recursive behavior is available on a channel, since $\nu Y.A$ is non-trivial and strictly positive.

Proceeding by induction on A, the interesting case is when $A = Y$, where in the unfolding of $\hat{\psi}(\hat{\rho}(Q))$ we reach the point where X is substituted by $\hat{\psi}(\hat{\rho}(Q))$, and we must show that the process is in $\mathcal{L}^{\omega[Y \mapsto \mathcal{C}]}[x{:}Y] = \mathcal{C}(x)$ for some x, which follows by definition of $\mathcal{C}$ and Lemma 29.

□

As before, it follows as a corollary from the fundamental theorem that well-typed processes do not diverge, even in the presence of coinductive session types. As we have discussed in Chapter 5, the presence of general recursive definitions and the potential for divergence introduces substantial challenges when reasoning about a complex system, since in the general case termination is not a compositional property: given two non-divergent processes, their composition and subsequent interactions can easily result in a system that diverges and produces little or no useful (observable) behavior. This is even more apparent in the presence of higher-order code mobility, where the highly dynamic nature of systems can easily introduce non-divergence in seemingly harmless process definitions.

Our termination result shows that the restrictions to the type system introduced above, when moving from general recursion to corecursion, are sufficient to ensure that well-typed processes enjoy a strong form of safety, since any well-typed process is statically guaranteed to be able to fulfil its specified protocol: no divergence can be introduced, nor can any deadlocks occur. Moreover, in our typed setting these properties are *compositional*, ensuring that any (well-typed) process composition produces a deadlock-free, non-divergent program.

These results showcase the power of our linear logical relations framework, where by developing a relatively similar logical predicate for polymorphic and coinductive session types and establishing that well-typed processes inhabit the predicate naturally entails a property of interest (in this case, termination) that is not straightforward to establish using more traditional proof techniques [83]. We note that while our emphasis here was on termination, the techniques developed in the sections above can be adapted to establish other properties expressible as predicates such as confluence [59].

In Chapter 7 we take this idea further by developing different applications of linear logical relations and the properties they entail. Before doing so, we introduce the natural next step in our framework of linear logical relations: the *relational* or binary version of the development, where instead of developing a proof technique to establish unary properties of typed processes, we seek to establish relational properties of processes (such as observational equivalence) and thus develop a lifting of our unary predicate to its arguably natural binary version.

6.2 Binary Logical Relations

As we have argued at the beginning of this part of the dissertation, one of the advantages of the framework of linear logical relations is that not only can it be used as a proof technique to show rather non-trivial properties such as termination but it also enables us to reason about program (or process) equivalence by considering the *binary* case of the logical relation, that is, lifting from a predicate on processes to a binary relation.

As we show in Section 6.2.2, we may use our binary logical relation for the polymorphic setting to develop a form of relational parametricity in the sense of Reynolds' abstraction theorem [65], a concept which was previously unexplored in session typed settings. Moreover, the equivalence relation that arises from the move to the binary setting turns out to coincide with the canonical notion of (typed) observational equivalence on π-calculus processes, commonly known as barbed congruence.

In Section 6.2.3 we introduce the variant of our binary logical relation for the coinductive setting of Section 6.1.2 without higher-order features and show the relational equivalent of the fundamental theorem of logical relations for this language (i.e. well-typed processes are related to themselves via the logical relation).

Before going into the details of our development, we first introduce the canonical notion of observational equivalence in process calculi known as *barbed congruence*, naturally extended to our typed setting.

6.2.1 Typed Barbed Congruence

A fundamental topic of study in process calculi is the notion of observational equivalence: when do two processes behave virtually "the same way"? To answer this question we need to determine what it means for process behavior to be the *same*.

Typically, for two processes to be deemed observationally equivalent they must be able to produce the same set of actions such that they are indistinguishable from each other in any admissible process context.

The canonical notion of contextual equivalence in concurrency theory is that of *barbed congruence*. Barbed congruence consists of a binary relation on processes that relies on the notion of a *barb*. A barb is the most basic observable on the behavior of processes, consisting simply of taking an action on a channel. For instance, the process $x\langle M\rangle.P$ has an (output) barb on x. Typically we distinguish between different kinds of barbs for convenience (e.g. output, input and selection).

Given the notion of a barb, there is a natural definition of *barbed equivalence*: two processes are barbed equivalent if they have the same barbs. Since we are interested in the so-called weak variants of equivalence (those that ignore differences at the level of internal actions), we can define *weak barbed equivalence* as an extension of barbed equivalence that is closed under reduction, that is, if P is weak barbed equivalent to Q then not only do they have the same barbs, but also every reduction of P is matched by zero or more reduction in Q (and vice-versa). Finally, we want our definition of observational equivalence to capture the intuition that two processes deemed observationally equivalent cannot be distinguished from one another. This is made precise by defining *barbed congruence* (written $\cong$) as the closure of weak barbed equivalence under any process context, that is, given a process with a "hole" which can be instantiated (i.e. a context), two processes are deemed barbed congruent if they are weak barbed equivalent when placed in *any* context. In a typed setting, we restrict our attention to processes of the same type, and to well-formed (according to typing) process contexts.

Formally, barbed congruence is defined as the largest equivalence relation on processes that is (i) closed under internal actions; (ii) preserves *barbs*; and is (iii) *contextual*. We now make these three *desiderata* formally precise, extending barbed congruence to our typed-setting through a notion of type-respecting relations. Below, we use $\Rightarrow$ to range over sequents of the form $\Gamma; \Delta \Rightarrow T$. The definitions below pertain to the polymorphic setting. We shall adapt them later for the coinductive case.

Definition 17 (Type-respecting relations)**.** *A (binary)* type-respecting relation *over processes, written $\{\mathcal{R}_{\Rightarrow}\}_{\Rightarrow}$, is defined as a family of relations over processes indexed by $\Rightarrow$. We often write $\mathcal{R}$ to refer to the whole family. Also, we write $\Omega;\Gamma;\Delta \Rightarrow P \,\mathcal{R}\, Q :: T$ to mean that*

(i) $\Omega;\Gamma;\Delta \Rightarrow P :: T$ *and* $\Omega;\Gamma;\Delta \Rightarrow Q :: T$,

(ii) $(P,Q) \in \mathcal{R}_{\Omega;\Gamma;\Delta \Rightarrow T}$.

We also need to define what constitutes a *type-respecting* equivalence relation. In what follows, we assume type-respecting relations and omit the adjective “type-respecting”.

Definition 18. *A relation $\mathcal{R}$ is said to be*

- Reflexive, *if $\Omega;\Gamma;\Delta \Rightarrow P :: T$ implies $\Omega;\Gamma;\Delta \Rightarrow P \,\mathcal{R}\, P :: T$;*
- Symmetric, *if $\Omega;\Gamma;\Delta \Rightarrow P \,\mathcal{R}\, Q :: T$ implies $\Omega;\Gamma;\Delta \Rightarrow Q \,\mathcal{R}\, P :: T$;*
- Transitive, *$\Omega;\Gamma;\Delta \Rightarrow P \,\mathcal{R}\, P' :: T$ and $\Omega;\Gamma;\Delta \Rightarrow P' \,\mathcal{R}\, Q :: T$ imply $\Omega;\Gamma;\Delta \Rightarrow P \,\mathcal{R}\, Q :: T$.*

Moreover, $\mathcal{R}$ is said to be an equivalence *if it is reflexive, symmetric, and transitive.*

We can now define the three *desiderata* for barbed congruence: τ-closed, barb preserving and contextuality.

Definition 19 (τ-closed)**.** *A relation $\mathcal{R}$ is* τ-closed *if $\Omega;\Gamma;\Delta \Rightarrow P\mathcal{R}Q :: T$ and $P \longrightarrow P'$ imply there exists a Q' such that $Q \Longrightarrow Q'$ and $\Omega;\Gamma;\Delta \Rightarrow P'\mathcal{R}Q' :: T$.*

The following definition of observability predicates, or *barbs*, extends the classical presentation with the observables for type input and output:

Definition 20 (Barbs)**.** *Given a name x, let $O_x = \{\overline{x}, x, \overline{x.\mathsf{inl}}, \overline{x.\mathsf{inr}}, x.\mathsf{inl}, x.\mathsf{inr}\}$ be the set of* basic observables *under x. Given a well-typed process P, we write:*

- $\mathsf{barb}(P,\overline{x})$, *if* $P \xrightarrow{\overline{(\nu y)x\langle y\rangle}} P'$;
- $\mathsf{barb}(P,\overline{x})$, *if* $P \xrightarrow{\overline{x\langle\rangle}} P'$
- $\mathsf{barb}(P,\overline{x})$, *if* $P \xrightarrow{\overline{x\langle A\rangle}} P'$, *for some* A, P';
- $\mathsf{barb}(P,x)$, *if* $P \xrightarrow{x(A)} P'$, *for some* A, P';
- $\mathsf{barb}(P,x)$, *if* $P \xrightarrow{x(y)} P'$, *for some* y, P';
- $\mathsf{barb}(P,x)$, *if* $P \xrightarrow{x()} P'$, *for some* P';
- $\mathsf{barb}(P,\alpha)$, *if* $P \xrightarrow{\alpha} P'$, *for some P' and $\alpha \in O_x \setminus \{x,\overline{x}\}$.*

Given some $o \in O_x$, we write $\mathsf{wbarb}(P,o)$ if there exists a P' such that $P \Longrightarrow P'$ and $\mathsf{barb}(P',o)$ holds.

Definition 21 (Barb preserving relation)**.** *A relation $\mathcal{R}$ is* barb preserving *if, for every name x, $\Omega; \Gamma; \Delta \Rightarrow P \mathcal{R} Q :: T$ and* $\mathsf{barb}(P, o)$ *imply* $\mathsf{wbarb}(Q, o)$*, for any $o \in O_x$.*

To define *contextuality*, we introduce a natural notion of *(typed) process contexts*. Intuitively, a context is a process that contains one hole, noted $\bullet$. Holes are *typed*: a hole can only be filled with a process matching its type. We shall use $K, K', \ldots$ for ranging over properly defined contexts, in the sense given below. We rely on left- and right-hand side typings for defining contexts and their properties precisely. We consider contexts with exactly one hole, but our definitions are easy to generalize. We define an extension of the syntax of processes with $\bullet$. We also extend sequents, as follows:

$$\mathcal{H}; \Omega; \Gamma; \Delta \Rightarrow K :: S$$

$\mathcal{H}$ contains a description of a hole occurring in (context) K: we have that $\bullet_{\Omega;\Gamma;\Delta\Rightarrow T}; \Omega; \Gamma; \Delta' \Rightarrow K :: S$ is the type of a context K whose hole is to be substituted by some process P such that $\Gamma; \Delta \Rightarrow P :: T$. As a result of the substitution, we obtain process $\Omega; \Gamma; \Delta' \Rightarrow K[P] :: S$. Since we consider at most one hole, $\mathcal{H}$ is either empty or has exactly one element. If $\mathcal{H}$ is empty then K is a process and we obtain its typing via the usual typing rules. We write $\Omega; \Gamma; \Delta \Rightarrow R :: T$ rather than $\cdot; \Omega; \Gamma; \Delta \Rightarrow R :: T$. The definition of typed contexts is completed by extending the type system with the following two rules:

$$\frac{}{\bullet_{\Omega;\Gamma;\Delta\Rightarrow T}; \Omega; \Gamma; \Delta \Rightarrow \bullet :: T}\ T\mathsf{hole} \qquad \frac{\Omega; \Gamma; \Delta \Rightarrow R :: T \quad \bullet_{\Omega;\Gamma;\Delta\Rightarrow T}; \Omega; \Gamma; \Delta' \Rightarrow K :: S}{\Omega; \Gamma; \Delta' \Rightarrow K[R] :: S}\ T\mathsf{fill}$$

Rule (Thole) allows us to introduce holes into typed contexts. In rule (Tfill), R is a process (it does not have any holes), and K is a context with a hole of type $\Omega; \Gamma; \Delta \Rightarrow T$. The substitution of occurrences of $\bullet$ in K with R, noted $K[R]$ is sound as long as the typings of R coincide with those declared in $\mathcal{H}$ for K.

Definition 22 (Contextuality)**.** *A relation $\mathcal{R}$ is* contextual *if it satisfies the conditions in Figure 6.5.*

Having made precise the necessary conditions, we can now formally define typed barbed congruence:

Definition 23 (Typed Barbed Congruence)**.** Typed barbed congruence*, noted $\cong$, is the largest equivalence on well-typed processes that is τ-closed, barb preserving, and contextual.*

Given the definitions above, it is easy to see why it is not straightforward to determine if two processes are indeed barbed congruent (and so, observationally equivalent), since we are quantifying over all possible typed process contexts. The typical approach is to devise a sound and complete *proof technique* for barbed congruence that does not rely on explicit checking of all contexts. This is usually achieved by defining some bisimulation on processes and then showing it to be a congruence, which is not a straightforward task in general.

A relation $\mathcal{R}$ is contextual if

0. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: y{:}A$ and $\Omega;\Gamma;y{:}A \Rightarrow [y \leftrightarrow z] :: z{:}A$ imply
$\Omega;\Gamma;\Delta \Rightarrow (\nu y)(P \mid [y \leftrightarrow z])\,\mathcal{R}\,(\nu y)(Q \mid [y \leftrightarrow z]) :: z{:}A$

1. $\Omega;\Gamma;\Delta,y{:}A \Rightarrow P\,\mathcal{R}\,Q :: x{:}B$ implies $\Omega;\Gamma;\Delta \Rightarrow x(y).P\,\mathcal{R}\,x(y).Q :: x{:}A \multimap B$

2. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: T$ implies $\Omega;\Gamma;\Delta,x{:}\mathbf{1} \Rightarrow x().P\,\mathcal{R}\,x().Q :: T$

3. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: y{:}A$ and $\Omega;\Gamma;\Delta' \Rightarrow S\,\mathcal{R}\,T :: x{:}B$ imply
$\Omega;\Gamma;\Delta,\Delta' \Rightarrow (\nu y)x\langle y\rangle.(P \mid S)\,\mathcal{R}\,(\nu y)x\langle y\rangle.(Q \mid T) :: x{:}A \otimes B$

4. $\Omega;\Gamma;\Delta' \Rightarrow P\,\mathcal{R}\,Q :: x{:}B$ and $\Omega;\Gamma;\Delta' \Rightarrow S\,\mathcal{R}\,T :: y{:}B$ imply
$\Omega;\Gamma;\Delta,\Delta' \Rightarrow (\nu y)x\langle y\rangle.(S \mid P)\,\mathcal{R}\,(\nu y)x\langle y\rangle.(T \mid Q) :: x{:}A \otimes B$

5. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: x{:}A$ and $\Omega;\Gamma;\Delta' \Rightarrow S\,\mathcal{R}\,T :: x{:}B$ imply
$\Omega;\Gamma;\Delta \Rightarrow x.\mathsf{case}(P,S)\,\mathcal{R}\,x.\mathsf{case}(Q,T) :: x{:}A \,\&\, B$

6. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: x{:}B$ and $\Omega;\Gamma;\Delta' \Rightarrow S\,\mathcal{R}\,T :: x{:}A$ imply
$\Omega;\Gamma;\Delta \Rightarrow x.\mathsf{case}(S,P)\,\mathcal{R}\,x.\mathsf{case}(T,Q) :: x{:}A \,\&\, B$

7. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: x{:}A$ implies $\Omega;\Gamma;\Delta \Rightarrow x.\mathsf{inl};P\,\mathcal{R}\,x.\mathsf{inl};Q :: x{:}A \oplus B$

8. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: x{:}B$ implies $\Omega;\Gamma;\Delta \Rightarrow x.\mathsf{inr};P\,\mathcal{R}\,x.\mathsf{inr};Q :: x{:}A \oplus B$

9. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: x{:}A$ and $\Omega;\Gamma;\Delta',x{:}A \Rightarrow S\,\mathcal{R}\,H :: T$ imply
$\Omega;\Gamma;\Delta,\Delta' \Rightarrow (\nu x)(P \mid S)\,\mathcal{R}\,(\nu x)(Q \mid H) :: T$

10. $\Omega;\Gamma;\Delta,x{:}A \Rightarrow P\,\mathcal{R}\,Q :: T$ and $\Omega;\Gamma;\Delta' \Rightarrow S\,\mathcal{R}\,H :: x{:}A$ imply
$\Omega;\Gamma;\Delta,\Delta' \Rightarrow (\nu x)(S \mid P)\,\mathcal{R}\,(\nu x)(H \mid Q) :: T$

11. $\Omega;\Gamma;\cdot \Rightarrow P\,\mathcal{R}\,Q :: y{:}A$ and $\Omega;\Gamma,u{:}A;\Delta \Rightarrow S\,\mathcal{R}\,H :: T$ imply
$\Omega;\Gamma;\Delta \Rightarrow (\nu u)(!u(y).P \mid S)\,\mathcal{R}\,(\nu u)(!u(y).Q \mid H) :: T$

12. $\Omega;\Gamma,u{:}A;\Delta \Rightarrow P\,\mathcal{R}\,Q :: T$ and $\Omega;\Gamma;\cdot \Rightarrow S\mathcal{R}H :: y{:}A$ and imply
$\Omega;\Gamma;\Delta \Rightarrow (\nu u)(!u(y).S \mid P)\,\mathcal{R}\,(\nu u)(!u(y).H \mid Q) :: T$

13. $\Omega;\Gamma,u{:}A;\Delta \Rightarrow P\,\mathcal{R}\,Q :: T$ implies $\Omega;\Gamma;\Delta,x{:}!A \Rightarrow x(u).P\,\mathcal{R}\,x(u).Q :: T$

14. $\Omega;\Gamma;\cdot \Rightarrow P\,\mathcal{R}\,Q :: y{:}A$ implies $\Omega;\Gamma;\cdot \Rightarrow (\nu u)x\langle u\rangle.!u(y).P\,\mathcal{R}\,(\nu u)x\langle u\rangle.!u(y).Q :: x{:}!A$

15. $\Omega;\Gamma;\Delta,y{:}A,x{:}B \Rightarrow P\,\mathcal{R}\,Q :: T$ implies $\Omega;\Gamma;\Delta,x{:}A \otimes B \Rightarrow x(y).P\,\mathcal{R}\,x(y).Q :: T$

16. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: y{:}A$ and $\Omega;\Gamma;\Delta',x{:}B \Rightarrow S\,\mathcal{R}\,H :: T$ imply
$\Omega;\Gamma;\Delta,\Delta',x{:}A \multimap B \Rightarrow (\nu y)x\langle y\rangle.(P \mid S)\,\mathcal{R}\,(\nu y)x\langle y\rangle.(Q \mid H) :: T$

17. $\Omega;\Gamma;\Delta,x{:}B \Rightarrow P\,\mathcal{R}\,Q :: T$ and $\Omega;\Gamma;\Delta' \Rightarrow S\,\mathcal{R}\,H :: y{:}A$ imply
$\Omega;\Gamma;\Delta,\Delta',x{:}A \multimap B \Rightarrow (\nu y)x\langle y\rangle.(P \mid S)\,\mathcal{R}\,(\nu y)x\langle y\rangle.(Q \mid H) :: T$

18. $\Omega;\Gamma,u{:}A;\Delta,y{:}A \Rightarrow P\,\mathcal{R}\,Q :: T$ implies $\Omega;\Gamma,u{:}A;\Delta \Rightarrow (\nu y)u\langle y\rangle.P\,\mathcal{R}\,(\nu y)u\langle y\rangle.Q :: T$

19. $\Omega;\Gamma;\Delta,x{:}A \Rightarrow P\,\mathcal{R}\,Q :: T$ and $\Omega;\Gamma;\Delta,x{:}B \Rightarrow S\,\mathcal{R}\,H :: T$ imply
$\Omega;\Gamma;\Delta,x{:}A \oplus B \Rightarrow x.\mathsf{case}(P,S)\,\mathcal{R}\,x.\mathsf{case}(Q,H) :: T$

20. $\Omega;\Gamma;\Delta,x{:}B \Rightarrow P\,\mathcal{R}\,Q :: T$ and $\Omega;\Gamma;\Delta,x{:}A \Rightarrow S\,\mathcal{R}\,H :: T$ imply
$\Omega;\Gamma;\Delta,x{:}A \oplus B \Rightarrow x.\mathsf{case}(S,P)\,\mathcal{R}\,x.\mathsf{case}(H,Q) :: T$

21. $\Omega;\Gamma;\Delta,x{:}A \Rightarrow P\,\mathcal{R}\,Q :: T$ implies $\Omega;\Gamma;\Delta,x{:}A \,\&\, B \Rightarrow x.\mathsf{inl};P\,\mathcal{R}\,x.\mathsf{inl};Q :: T$

22. $\Omega;\Gamma;\Delta,x{:}B \Rightarrow P\,\mathcal{R}\,Q :: T$ implies $\Omega;\Gamma;\Delta,x{:}A \,\&\, B \Rightarrow x.\mathsf{inr};P\,\mathcal{R}\,x.\mathsf{inr};Q :: T$

23. $\Omega,X;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: z{:}A$ implies $\Omega;\Gamma;\Delta \Rightarrow z(X).P\,\mathcal{R}\,z(X).Q :: z{:}\forall X.A$

24. $\Omega;\Gamma;\Delta \Rightarrow P\,\mathcal{R}\,Q :: z{:}A\{{}^{B}\!/X\}$ and $\Omega \Rightarrow B\ \mathsf{type}$ imply $\Omega;\Gamma;\Delta \Rightarrow z\langle B\rangle.P\,\mathcal{R}\,z\langle B\rangle.Q :: z{:}\exists X.A$

25. $\Omega;\Gamma;\Delta,x : A\{{}^{B}\!/X\} \Rightarrow P\,\mathcal{R}\,Q :: T$ and $\Omega \Rightarrow B\ \mathsf{type}$ imply
$\Omega;\Gamma;\Delta,x : \forall X.A \Rightarrow x\langle B\rangle.P\,\mathcal{R}\,x\langle B\rangle.Q :: T$

26. $\Omega,X;\Gamma;\Delta,x : A \Rightarrow P\mathcal{R}Q :: T$ implies $\Omega;\Gamma;\Delta,x : \exists X.A \Rightarrow x(X).P\,\mathcal{R}\,x(X).Q :: T$

Figure 6.5: Conditions for Contextual Type-Respecting Relations

6.2.2 Logical Equivalence for Polymorphic Session Types

We now address the problem of identifying a proof technique for observational equivalence in a polymorphic setting using our framework of linear logical relations. In the functional setting, we typically define a notion of observational equivalence that is analogous to the idea of barbed congruence and then develop a notion of *logical equivalence*, obtained by extending the unary logical predicate to the binary case, that characterizes observational equivalence. Moreover, in the polymorphic setting this logical equivalence also captures the idea of *parametricity*. It turns out that a similar development can be achieved in our polymorphic session-typed setting. The techniques developed here can also be directly applied to the propositional setting of Section 2.3, with the obvious changes to the formalism (omitting polymorphic types and related constructs). We do not pursue such a development here since our logical equivalence for polymorphic session types subsumes the simpler propositional case (we point the interested reader to [58] for a theory of linear logical relations for the propositional case, although formulated in a slightly different way – not using the candidates technique – that does not generalize to the polymorphic setting).

To move to a binary or *relational* version of our logical predicate, we first need the notion of an *equivalence candidate* which is the binary analogue of a reducibility candidate, consisting of of a binary relation on typed processes that is *closed* under barbed congruence.

Definition 24 (Equivalence Candidate)**.** *Let A, B be types. An* equivalence candidate *$\mathcal{R}$ at $z{:}A$ and $z{:}B$, noted $\mathcal{R} :: z{:}A \Leftrightarrow B$, is a binary relation on processes such that, for every $(P, Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$ both $\cdot \Rightarrow P :: z{:}A$ and $\cdot \Rightarrow Q :: z{:}B$ hold, together with the following conditions:*

1. *If $(P, Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$, $\cdot \vdash P \cong P' :: z{:}A$, and $\cdot \vdash Q \cong Q' :: z{:}B$ then $(P', Q') \in \mathcal{R} :: z{:}A \Leftrightarrow B$.*

2. *If $(P, Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$ then, for all P_0 such that $P_0 \Longrightarrow P$, we have $(P_0, Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$. Similarly for Q: If $(P, Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$ then, for all Q_0 such that $Q_0 \Longrightarrow Q$ then $(P, Q_0) \in \mathcal{R} :: z{:}A \Leftrightarrow B$.*

We often write $(P, Q) \in \mathcal{R} :: z{:}A \Leftrightarrow B$ as $P \, \mathcal{R} \, Q :: z{:}A \Leftrightarrow B$.

Given this notion of candidate we define logical equivalence by defining the binary version of the logical predicate of Section 6.1.1, written $\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: T[\eta : \omega \Leftrightarrow \omega']$, where both P and Q are typed in the same contexts and right-hand side typing, ω and ω' are as before, mappings from type variables to types (the former applied to P and the latter to Q) and η is now a mapping from type variables to equivalence candidates that respects ω and ω' (written $\eta : \omega \Leftrightarrow \omega'$). Just as before, we follow the same idea of inductively closing open processes to then focus on the behavior of closed processes as defined by their right-hand side typings.

Definition 25 (Logical Equivalence - Inductive Case)**.** *Let Γ, Δ be non empty typing environments. Given the sequent $\Omega; \Gamma; \Delta \Rightarrow T$, the binary relation on processes $\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: T[\eta : \omega \Leftrightarrow \omega']$*

(with $\omega, \omega' : \Omega$ and $\eta : \omega \Leftrightarrow \omega'$) is inductively defined as:

$$\begin{array}{rl}
\Gamma; \Delta, y : A \Rightarrow P \approx_{\mathrm{L}} Q :: T[\eta : \omega \Leftrightarrow \omega'] & \textit{iff} \quad \forall R_1, R_2.\ \textit{s.t.}\ R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'], \\
& \Gamma; \Delta \Rightarrow (\nu y)(\hat{\omega}(P) \mid \hat{\omega}(R_1)) \approx_{\mathrm{L}} (\nu y)(\hat{\omega}'(Q) \mid \hat{\omega}'(R_2)) :: T[\eta : \omega \Leftrightarrow \omega'] \\
\Gamma, u : A; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: T[\eta : \omega \Leftrightarrow \omega'] & \textit{iff} \quad \forall R_1, R_2.\ \textit{s.t.}\ R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'], \\
& \Gamma; \Delta \Rightarrow (\nu y)(\hat{\omega}(P) \mid !u(y).\hat{\omega}(R_1)) \approx_{\mathrm{L}} (\nu y)(\hat{\omega}'(Q) \mid !u(y).\hat{\omega}'(R_2)) :: T[\eta : \omega \Leftrightarrow \omega']
\end{array}$$

While the formalism seems quite verbose, the underlying ideas are straightforward generalizations of the logical predicate we previously introduced. For instance, in the logical predicate of Section 6.1.1, the case for $z{:}A \multimap B$ was defined as being able to observe an input action $z(x)$ on a process, for which the resulting continuation, when composed with any process in the predicate at type $x{:}A$ would be in the predicate at type $z{:}B$. The definition of logical equivalence follows exactly this pattern: two processes P and Q are logically equivalent at type $z{:}A \multimap B$ when we are able to match an input action $z(x)$ in P with an input action $z(x)$ in Q, such that for any logically equivalent processes R_1 and R_2 at type $x{:}A$, the continuations, when composed respectively with R_1 and R_2 will be logically equivalent at type $z{:}B$. The remaining cases follow a similar generalization pattern. The precise definition is given below (Definition 26).

Definition 26 (Logical Equivalence - Base Case)**.** *Given a type A and mappings ω, ω', η, we define* logical equivalence*, noted $P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \Leftrightarrow \omega']$, as the largest binary relation containing all pairs of processes (P, Q) such that (i) $\cdot \Rightarrow \hat{\omega}(P) :: z{:}\hat{\omega}(A)$; (ii) $\cdot \Rightarrow \hat{\omega}'(Q) :: z{:}\hat{\omega}'(A)$; and (iii) satisfies the conditions in Figure 6.6.*

We can now setup the formalism required to show the fundamental theorem of logical relations in this binary setting: any well-typed process is logically equivalent to itself. We begin by establishing our logical relation as one of the equivalence candidates, as we did before for the unary case.

Lemma 30. *Suppose $P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \leftrightarrow \omega']$. If, for any P_0, we have $P_0 \Longrightarrow P$ then $P_0 \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \leftrightarrow \omega']$. Also: if, for any Q_0, we have $Q_0 \Longrightarrow Q$ then $P \approx_{\mathrm{L}} Q_0 :: z{:}A[\eta : \omega \leftrightarrow \omega']$.*

Proof. By induction on the structure of A, relying on Type Preservation (Theorem 20) and the definition of η. □

Lemma 31. *Suppose $P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \leftrightarrow \omega']$. If $\cdot \Rightarrow P \cong P' :: z{:}\hat{\omega}(A)$ and $\cdot \Rightarrow Q \cong Q' :: z{:}\widehat{\omega'}(A)$ then $P' \approx_{\mathrm{L}} Q' :: z{:}A[\eta : \omega \leftrightarrow \omega']$.*

Proof. By induction on the structure of A, using contextuality of $\cong$. □

Theorem 37 (Logical Equivalence is an Equivalence Candidate)**.** *Relation $P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \leftrightarrow \omega']$ is an equivalence candidate at $z{:}\hat{\omega}(A)$ and $z{:}\widehat{\omega'}(A)$, in the sense of Definition 24.*

Proof. Follows from Lemmas 30 and 31. □

The following is a generalization of Lemma 31:

$$
\begin{array}{rcl}
P \approx_{\mathtt{L}} Q :: z{:}X[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & (P, Q) \in \eta(X)(z) \\
P \approx_{\mathtt{L}} Q :: z{:}\mathbf{1}[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P', Q'.\, (P \stackrel{z\langle\rangle}{\Longrightarrow} P' \wedge P' \not\rightarrow \wedge\, Q \stackrel{z\langle\rangle}{\Longrightarrow} Q' \wedge Q' \not\rightarrow) \Rightarrow \\
 & & (P' \equiv_{!} \mathbf{0} \wedge Q' \equiv_{!} \mathbf{0}) \\
P \approx_{\mathtt{L}} Q :: z{:}A \multimap B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P', y.\ (P \stackrel{z(y)}{\Longrightarrow} P') \Rightarrow \exists Q'.Q \stackrel{z(y)}{\Longrightarrow} Q' \ s.t. \\
 & & \forall R_1, R_2.\ R_1 \approx_{\mathtt{L}} R_2 :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \\
 & & (\nu y)(P' \mid R_1) \approx_{\mathtt{L}} (\nu y)(Q' \mid R_2) :: z{:}B[\eta : \omega \Leftrightarrow \omega'] \\
P \approx_{\mathtt{L}} Q :: z{:}A \otimes B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P', y.\ (P \stackrel{\overline{(\nu y)z\langle y\rangle}}{\Longrightarrow} P') \Rightarrow \exists Q'.Q \stackrel{\overline{(\nu y)z\langle y\rangle}}{\Longrightarrow} Q' \ s.t. \\
 & & \exists P_1, P_2, Q_1, Q_2.P' \equiv_{!} P_1 \mid P_2 \wedge Q' \equiv_{!} Q_1 \mid Q_2 \wedge \\
 & & P_1 \approx_{\mathtt{L}} Q_1 :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \wedge P_2 \approx_{\mathtt{L}} Q_2 :: z{:}B[\eta : \omega \Leftrightarrow \omega'] \\
P \approx_{\mathtt{L}} Q :: z{:}!A[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall P'.\ (P \stackrel{\overline{(\nu u)z\langle u\rangle}}{\Longrightarrow} P') \Rightarrow \exists Q', P_1, Q_1, y.Q \stackrel{\overline{(\nu u)z\langle u\rangle}}{\Longrightarrow} Q' \wedge \\
 & & P' \equiv_{!} !u(y).P_1 \wedge Q' \equiv_{!} !u(y).Q_1 \wedge P_1 \approx_{\mathtt{L}} Q_1 :: y{:}A[\eta : \omega \Leftrightarrow \omega'] \\
P \approx_{\mathtt{L}} Q :: z{:}A \mathbin{\&} B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \\
(\forall P'.(P \stackrel{\mathsf{z.inl}}{\Longrightarrow} P') & \Rightarrow & \exists Q'.(Q \stackrel{\mathsf{z.inl}}{\Longrightarrow} Q' \wedge P' \approx_{\mathtt{L}} Q' :: z{:}A[\eta : \omega \Leftrightarrow \omega'])) \wedge \\
(\forall P'.(P \stackrel{\mathsf{z.inr}}{\Longrightarrow} P') & \Rightarrow & \exists Q'.(Q \stackrel{\mathsf{z.inr}}{\Longrightarrow} Q' \wedge P' \approx_{\mathtt{L}} Q' :: z{:}B[\eta : \omega \Leftrightarrow \omega'])) \\
P \approx_{\mathtt{L}} Q :: z{:}A \oplus B[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \\
(\forall P'.(P \stackrel{\overline{\mathsf{z.inl}}}{\Longrightarrow} P') & \Rightarrow & \exists Q'.(Q \stackrel{\overline{\mathsf{z.inl}}}{\Longrightarrow} Q' \wedge P' \approx_{\mathtt{L}} Q' :: z{:}A[\eta : \omega \Leftrightarrow \omega'])) \wedge \\
(\forall P'.(P \stackrel{\overline{\mathsf{z.inr}}}{\Longrightarrow} P') & \Rightarrow & \exists Q'.(Q \stackrel{\overline{\mathsf{z.inr}}}{\Longrightarrow} Q' \wedge P' \approx_{\mathtt{L}} Q' :: z{:}B[\eta : \omega \Leftrightarrow \omega'])) \\
P \approx_{\mathtt{L}} Q :: z{:}\forall X.A[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \forall B_1, B_2, P', \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\ (P \stackrel{z(B_1)}{\Longrightarrow} P') \Rightarrow \\
\exists Q'.Q \stackrel{z(B_2)}{\Longrightarrow} Q',\ P' & \approx_{\mathtt{L}} & Q' :: z{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]] \\
P \approx_{\mathtt{L}} Q :: z{:}\exists X.A[\eta : \omega \Leftrightarrow \omega'] & \text{iff} & \exists B_1, B_2, \mathcal{R} :: -{:}B_1 \Leftrightarrow B_2.\ (P \stackrel{\overline{z\langle B\rangle}}{\Longrightarrow} P') \Rightarrow \\
\exists Q'.Q \stackrel{\overline{z\langle B\rangle}}{\Longrightarrow} Q',\ P' & \approx_{\mathtt{L}} & Q' :: z{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B_1] \Leftrightarrow \omega'[X \mapsto B_2]]
\end{array}
$$

Figure 6.6: Logical Equivalence for Polymorphic Session Types (base case).

Lemma 32. *Suppose* $\Gamma; \Delta \Rightarrow P \approx_{\mathtt{L}} Q :: z{:}A[\eta : \omega \leftrightarrow \omega']$. *If* $\Gamma; \Delta \Rightarrow P \cong P' :: z{:}\hat{\omega}(A)$ *and* $\Gamma; \Delta \Rightarrow Q \cong Q' :: z{:}\hat{\omega'}(A)$ *then* $\Gamma; \Delta \Rightarrow P' \approx_{\mathtt{L}} Q' :: z{:}A[\eta : \omega \leftrightarrow \omega']$.

Proof. By induction on the combined size of Γ and Δ, following Definition 25 and Theorem 37; the base case uses Lemma 31. □

As established earlier for the unary case, we require a compositionality theorem that characterizes the substitutions encoded by the mappings η and ω. The proof follows closely that of Theorem 29.

Theorem 38 (Compositionality)**.** *Let B be any well-formed type. Then,* $P \approx_{\mathtt{L}} Q :: z{:}A\{^{B}/_{X}\}[\eta : \omega \Leftrightarrow \omega']$ *if and only if*

$$P \approx_{\mathtt{L}} Q :: z{:}A[\eta[X \mapsto \mathcal{R}] \ : \ \omega[X \mapsto \hat{\omega}(B)] \leftrightarrow \omega'[X \mapsto \hat{\omega}'(B)]]$$

where $\mathcal{R} :: -{:}\hat{\omega}(B) \Leftrightarrow \hat{\omega'}(B)$ *stands for logical equivalence (cf. Definition 26).*

Proof. By induction on the structure of A, relying on Definition 26. □

In the proof of the fundamental theorem it is convenient to use process representatives for left-hand side contexts as before.

Definition 27. *Let* $\Gamma = u_1{:}B_1, \dots, u_k{:}B_k$, *and* $\Delta = x_1{:}A_1, \dots, x_n{:}A_n$ *be a non-linear and a linear typing environment, respectively, such that* $\Omega \triangleright \Delta$ *and* $\Omega \triangleright \Gamma$, *for some* Ω. *Letting* $I = \{1, \dots, k\}$ *and* $J = \{1, \dots, n\}$, $\omega, \omega' : \Omega$, $\eta : \omega \Leftrightarrow \omega'$, *we define the sets of pairs of processes* $\mathcal{C}_\Gamma^{\omega,\omega',\eta}$ *and* $\mathcal{C}_\Delta^{\omega,\omega',\eta}$ *as:*

$$\mathcal{C}_\Gamma^{\omega,\omega',\eta} \stackrel{def}{=} \{ \prod_{i \in I} (!u_i(y_i).\hat{\omega}(R_i), !u_i(y_i).\hat{\omega}'(S_i)) \mid R_i \approx_{\mathrm{L}} S_i :: y_i{:}B_i[\eta[\omega \Leftrightarrow \omega']] \}$$

$$\mathcal{C}_\Delta^{\omega,\omega',\eta} \stackrel{def}{=} \{ \prod_{j \in J} (\hat{\omega}(Q_j), \hat{\omega}'(U_j)) \mid Q_j \approx_{\mathrm{L}} U_j :: x_j{:}A_j[\eta[\omega \Leftrightarrow \omega']] \}$$

Lemma 33. *Let* P *and* Q *be processes such that* $\Omega; \Gamma; \Delta \Rightarrow P :: T$, $\Omega; \Gamma; \Delta \Rightarrow Q :: T$ *with* $\Gamma = u_1{:}B_1, \dots, u_k{:}B_k$ *and* $\Delta = x_1{:}A_1, \dots, x_n{:}A_n$, *with* $\Omega \triangleright \Delta$, $\Omega \triangleright \Gamma$, $\omega, \omega' : \Omega$, *and* $\eta : \omega \Leftrightarrow \omega'$. *We have:*

$$\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: T[\eta[\omega \Leftrightarrow \omega']] \ \textit{ iff } \ \forall (R_1, R_2) \in \mathcal{C}_\Gamma^{\omega,\omega',\eta}, \forall (S_1, S_2) \in \mathcal{C}_\Delta^{\omega,\omega',\eta},$$

$$(\nu \tilde{u}, \tilde{x})(P \mid R_1 \mid S_1) \approx_{\mathrm{L}} (\nu \tilde{u}, \tilde{x})(Q \mid R_2 \mid S_2) :: T[\eta[\omega \Leftrightarrow \omega']]$$

Proof. Straightforward from Def. 25 and 27. □

In the proof of the fundamental theorem, we exploit termination of well-typed processes to ensure that actions can be matched with finite weak transitions.

Theorem 39 (Fundamental Theorem – Parametricity). *If* $\Omega; \Gamma; \Delta \Rightarrow P :: z{:}A$ *then, for all* $\omega, \omega' : \Omega$ *and* $\eta : \omega \leftrightarrow \omega'$, *we have*

$$\Gamma; \Delta \Rightarrow \hat{\omega}(P) \approx_{\mathrm{L}} \hat{\omega'}(P) :: z{:}A[\eta : \omega \leftrightarrow \omega']$$

Proof. By induction on typing, exploiting termination, and Theorems 37 and 38.

Case (T$\forall$R2): $\Omega; \Gamma; \Delta \Rightarrow z(X).P :: z{:}\forall X.A$

$\Omega, X; \Gamma; \Delta \Rightarrow P :: z{:}A$ by inversion

Let $\omega_1 = \omega[X \mapsto B]$, $\omega_2 = \omega'[X \mapsto B]$ and $\eta_1 = \eta[X \mapsto \mathcal{R} :: -{:}B]$ for some B.

$\Gamma; \Delta \Rightarrow \hat{\omega_1}(P)\{B/X\} \approx_{\mathrm{L}} \hat{\omega_2}(P)\{B/X\} :: z{:}A[\eta_1{:}\omega_1 \Leftrightarrow \omega_2]$ by i.h.

Pick any $(G_1, G_2) \in \mathcal{C}_\Gamma^{\omega_1,\omega_2,\eta_1}$, $(D_1, D_2) \in \mathcal{C}_\Delta^{\omega_1,\omega_2,\eta_1}$:

$(\nu \tilde{u}, \tilde{x})(\hat{\omega_1}(P)\{B/X\} \mid G_1 \mid D_1) \approx_{\mathrm{L}} (\nu \tilde{u}, \tilde{x})(\hat{\omega_2}(P)\{B/X\} \mid G_2 \mid D_2) :: z{:}A[\eta[\omega_1 \Leftrightarrow \omega_2]]$

(a) by Lemma 37

$z(X).\hat{\omega}(P) \xrightarrow{z(B)} \hat{\omega}(P)\{B/X\}$ by l.t.s

S.T.S.: $(\nu \tilde{u}, \tilde{x})(z(X).\hat{\omega}(P) \mid G_1 \mid D_1) \approx_{\mathrm{L}} (\nu \tilde{u}, \tilde{x})(z(X).\hat{\omega}'(P) \mid G_2 \mid D_2) :: z{:}\forall X.A[\eta : \omega \Leftrightarrow \omega']$

by Lemma 37 and strengthening

$(\nu\widetilde{u},\widetilde{x})(z(X).\hat{\omega}'(P) \mid G_1 \mid D_1) \overset{z(B)}{\Longrightarrow} (\nu\widetilde{u},\widetilde{x})(\hat{\omega}'(P) \mid G_2 \mid D_2')$ by l.t.s

$(\nu\widetilde{u},\widetilde{x})(\hat{\omega}_1(P)\{B/X\} \mid G_1 \mid D_1) \approx_{\mathtt{L}} (\nu\widetilde{u},\widetilde{x})(\hat{\omega}_2(P)\{B/X\} \mid G_2 \mid D_2') :: z{:}A[\eta[\omega_1 \Leftrightarrow \omega_2]]$
by (a) and forward closure

$(\nu\widetilde{u},\widetilde{x})(z(X).\hat{\omega}(P) \mid G \mid D) \approx_{\mathtt{L}} (\nu\widetilde{u},\widetilde{x})(z(X).\hat{\omega}'(P) \mid G \mid D) :: z{:}\forall X.A[\eta : \omega \Leftrightarrow \omega']$
by Def. 26 and strengthening

Case (T$\forall$L2): $\Omega;\Gamma;\Delta, x{:}\forall X.A \Rightarrow x\langle B\rangle.P :: T$

$\Omega \vdash B$ type by inversion

$\Omega;\Gamma;\Delta, x{:}A\{B/X\} \Rightarrow P :: T$ by inversion

$\Gamma;\Delta, x{:}A\{B/X\} \Rightarrow \hat{\omega}(P) \approx_{\mathtt{L}} \hat{\omega}'(P) :: T[\eta : \omega \Longrightarrow \omega']$ (a) by i.h.

Pick any $(G_1, G_2) \in \mathcal{C}_{\Gamma}^{\omega,\omega',\eta}$ and $(D_1, D_2) \in \mathcal{C}_{\Delta}^{\omega,\omega',\eta}$, $R_1 \approx_{\mathtt{L}} R_2 :: x{:}A\{B/X\}[\eta : \omega \Leftrightarrow \omega']$:

$(\nu\widetilde{u},\widetilde{x})(\hat{\omega}(P) \mid R_1 \mid D \mid G) \approx_{\mathtt{L}} (\nu\widetilde{u},\widetilde{x})(\hat{\omega}'(P) \mid R_2 \mid D \mid G) :: T[\eta : \omega \Leftrightarrow \omega']$
(b) by (a) Lemma 37

Pick S_1, S_2 such that $S_1 \approx_{\mathtt{L}} S_2 :: x{:}\forall X.A[\eta : \omega \Leftrightarrow \omega']$:

S.T.S: $(\nu\widetilde{u},\widetilde{x})(\hat{\omega}(x\langle B\rangle.P) \mid G \mid D \mid S_1) \approx_{\mathtt{L}} (\nu\widetilde{u},\widetilde{x})(\hat{\omega}'(x\langle B\rangle.P) \mid G \mid D \mid S_2) :: T[\eta : \omega \Leftrightarrow \omega']$

$S_1 \overset{x(\hat{\omega}(B))}{\Longrightarrow} S_1'$, $S_2 \overset{x(\hat{\omega}'(B))}{\Longrightarrow} S_2'$

$\forall \mathcal{R} :: -{:}B \Leftrightarrow B.S_1' \approx_{\mathtt{L}} S_2' :: x{:}A[\eta[X \mapsto \mathcal{R}] : \omega[X \mapsto B] \Leftrightarrow \omega'[X \mapsto B]]$ by Def. 26

$S_1' \approx_{\mathtt{L}} S_2' :: x{:}A[\eta[X \mapsto \approx_{\mathtt{L}}] : \omega[X \mapsto B] \Leftrightarrow \omega'[X \mapsto B]]$ by Theorem 37

$S_1' \approx_{\mathtt{L}} S_2' :: x{:}A\{B/X\}[\eta : \omega \Leftrightarrow \omega']$ by Theorem 38

$(\nu\widetilde{u},\widetilde{x})(\hat{\omega}(x\langle B\rangle.P) \mid G \mid D \mid S_1) \Longrightarrow (\nu\widetilde{u},\widetilde{x})(\hat{\omega}(P) \mid G \mid D' \mid S_1') = M_1$

$(\nu\widetilde{u},\widetilde{x})(\hat{\omega}'(x\langle B\rangle.P) \mid G \mid D \mid S_2) \Longrightarrow (\nu\widetilde{u},\widetilde{x})(\hat{\omega}'(P) \mid G \mid D' \mid S_2') = M_2$

$M_1 \approx_{\mathtt{L}} M_2 :: T[\eta : \omega \Leftrightarrow \omega']$ by (b) and forward closure

$(\nu\widetilde{u},\widetilde{x})(\hat{\omega}(x\langle B\rangle.P) \mid G \mid D \mid S_1) \approx_{\mathtt{L}} (\nu\widetilde{u},\widetilde{x})(\hat{\omega}'(x\langle B\rangle.P) \mid G \mid D \mid S_2) :: T[\eta : \omega \Leftrightarrow \omega']$
by backward closure

Remaining cases follow a similar pattern.

□

We have thus established the fundamental theorem of logical relations for our polymorphic session-typed setting, which entails a form of parametricity in the sense of Reynolds [65], a previously unknown result for session-typed languages. Notice how the proof follows quite closely the structure of the unary case, making a case for the generality of the approach.

In Section 7.1 we use this result to directly reason about polymorphic processes, where we also show that our logical equivalence turns out to be sound and complete with respect to typed barbed congruence, serving as a proof technique for the canonical notion of observational equivalence in process calculi.

6.2.3 Logical Equivalence for Coinductive Session Types

We now devote our attention to developing a suitable notion of logical equivalence (in the sense of 6.2.2) for our calculus with coinductive session types.

To simplify the development, we restrict the process calculus of Section 6.1.2 to not include higher-order process passing features, nor term communication. The reasoning behind this restriction is that in order to give a full account of logical equivalence for the full calculus we would need to specify a suitable notion of equivalence for the functional layer. Given that monadic objects are terms of the functional language that cannot be observed within the functional layer itself, we would thus need to account for equivalence of monadic objects potentially through barbed congruence, which would itself need to appeal to equivalence of the functional language. It is not immediately obvious how to develop a reasonable notion of observation equivalence under these constraints.

While the formalization of such a development is of interest, given that it would arguably produce a notion of equivalence for higher-order processes (for which reasoning about equivalence is particularly challenging [66, 72]), it would lead us too far astray in our development. Moreover, the higher-order features and coinductive definitions are rather orthogonal features. Thus, we compromise and focus solely on the latter in the development of this section. We discuss how to extend the development to account for higher-order features in Section 6.3.

We begin by revisiting our definition of typed barbed congruence. The definition of type-respecting relations (Def. 17) is modified in the obvious way, referring the typing judgment $\Gamma; \Delta \Rightarrow_\eta P :: T$. Similarly for definitions 18 and 19 of an equivalence relation and tau-closedness. Our notion of a barb (Def. 20) is also straightforwardly adapted by omitting the clauses for type input and output and generalizing binary labelled choice to n-ary labelled choice. Contextuality (Def. 22) is updated to account for the process contexts generated by the typing rules for corecursive process definitions. In essence, we update Fig. 6.5 by erasing the typing context for type variables Ω, updating clauses 19 through 22 to account for n-ary choice and selection, omitting clauses 23 through 26 pertaining to polymorphism and adding the following clauses:

1. $\Gamma; \Delta, c{:}A\{\nu X.A/X\} \Rightarrow_\eta P \mathcal{R} Q :: T$ implies $\Gamma; \Delta, c{:}\nu X.A \Rightarrow_\eta P \mathcal{R} Q :: T$

2. $\Gamma; \Delta \Rightarrow_{\eta'} P \mathcal{R} Q :: c{:}A$ with $\eta' = \eta[X(\overline{y}) \mapsto \Gamma; \Delta \Rightarrow c{:}\nu Y.A]$ implies

$$\Gamma; \Delta \Rightarrow_\eta ((\mathsf{corec}\ X(\overline{y}).P\{\overline{y}/\overline{z}\})\ \overline{z}) \mathcal{R} (\mathsf{corec}\ X(\overline{y}).Q\{\overline{y}/\overline{z}\})\ \overline{z}) :: c{:}\nu Y.A$$

With the appropriate redefinition of barbed congruence in place, we are now in a position to define logical equivalence for coinductive session types. We begin with the updated version of an equivalence candidate.

Definition 28 (Equivalence Candidate). *Let A be a session type. An* equivalence candidate $\mathcal{R}$ *at $z{:}A$, noted $\mathcal{R} :: z{:}A$ is a binary relation on processes such that, for every $(P, Q) \in \mathcal{R} :: z{:}A$ both $\cdot \Rightarrow P :: z{:}A$ and $\cdot \Rightarrow Q :: z{:}A$ hold, together with the following conditions:*

1. If $(P, Q) \in \mathcal{R} :: z{:}A$, $\cdot \vdash P \cong P' :: z{:}A$, and $\cdot \vdash Q \cong Q' :: z{:}A$ then $(P', Q') \in \mathcal{R} :: z{:}A$.

2. *If $(P, Q) \in \mathcal{R} :: z{:}A$ then, for all P_0 such that $P_0 \Longrightarrow P$, we have $(P_0, Q) \in \mathcal{R} :: z{:}A$. Similarly for Q: If $(P, Q) \in \mathcal{R} :: z{:}A$ then, for all Q_0 such that $Q_0 \Longrightarrow Q$ then $(P, Q_0) \in \mathcal{R} :: z{:}A$.*

We often write $(P, Q) \in \mathcal{R} :: z{:}A$ as $P \, \mathcal{R} \, Q :: z{:}A$.

In analogy with the unary case, we write $\mathbb{R} :: -{:}A$ for the collection of all equivalence candidates at type A. As in the case for polymorphism, we define logical equivalence by defining the binary version of the logical predicate of Section 6.1.2, written $\Gamma; \Delta \Rightarrow_\eta P \approx_{\mathrm{L}} Q :: T[\omega]$ where P and Q are typed in the same contexts and right-hand side typing and ω is a mapping from type variables to equivalence candidates (compatible with η). As before, we follow the same outline of separating the cases of open processes and closed processes. Note that, unlike in the case for polymorphism, we do not need to maintain two separate mappings for type variables since the instantiations are the same in the typings of the two processes in the relation.

Definition 29 (Logical Equivalence - Inductive Case)**.** *Let Γ, Δ be non empty typing environments. Given the sequent $\Gamma; \Delta \Rightarrow_\eta T$, the binary relation on processes $\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: T[\omega]$, where ω is compatible with η is is inductively defined as:*

$$\begin{array}{c}\Gamma; \Delta, y : A \Rightarrow_\eta P \approx_{\mathrm{L}} Q :: T[\omega] \quad \textit{iff} \quad \forall R_1, R_2.\ \textit{s.t.}\ R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\omega], \\ \qquad\qquad \Gamma; \Delta \Rightarrow (\nu y)(P \mid R_1) \approx_{\mathrm{L}} (\nu y)(Q \mid R_2) :: T[\omega] \\ \Gamma, u : A; \Delta \Rightarrow_\eta P \approx_{\mathrm{L}} Q :: T[\omega] \quad \textit{iff} \quad \forall R_1, R_2.\ \textit{s.t.}\ R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\omega], \\ \Gamma; \Delta \Rightarrow_\eta (\nu y)(P \mid !u(y).R_1) \approx_{\mathrm{L}} (\nu y)(Q \mid !u(y).R_2) :: T[\omega]\end{array}$$

The base case consists of the natural generalization of the unary version of the logical relation to the binary setting. For brevity we omit the cases for the additives (& and $\oplus$).

Definition 30 (Logical Equivalence - Base Case)**.** *Given a type A and mappings ω, η we define* logical equivalence*, noted $P \approx_{\mathrm{L}} Q :: z{:}A[\omega]$, as the largest binary relation containing all pairs of processes (P, Q) such that (i) $\cdot \Rightarrow_\eta P :: z{:}A$; (ii) $\cdot \Rightarrow_\eta Q :: z{:}A$; and (iii) satisfies the conditions in Figure 6.7.*

We now show the appropriate auxiliary results in order to establish the fundamental theorem of logical relations for equivalence in the presence of coinductive session types. As before, we show that logical equivalence is an equivalence candidate by showing it satisfies the appropriate closure conditions. We then establish a greatest fixpoint property for the interpretation of coinductive session types and after defining the convenient notion of process representatives for contexts, we are in a position to show the fundamental theorem.

Lemma 34. *Suppose $P \approx_{\mathrm{L}} Q :: z{:}A[\omega]$. If, for any P_0, we have $P_0 \Longrightarrow P$ then $P_0 \approx_{\mathrm{L}} Q :: z{:}A[\omega]$. Also: if, for any Q_0, we have $Q_0 \Longrightarrow Q$ then $P \approx_{\mathrm{L}} Q_0 :: z{:}A[\omega]$.*

Proof. By induction on the structure of A, relying on Type Preservation and the definition of ω. □

$$
\begin{array}{rcl}
P \approx_{\mathrm{L}} Q :: z{:}X[\omega] & \text{iff} & (P,Q) \in \omega(X)(z) \\
P \approx_{\mathrm{L}} Q :: z{:}\mathbf{1}[\omega] & \text{iff} & \forall P', Q'.\ (P \xRightarrow{z\langle\rangle} P' \wedge P' \not\to \wedge\ Q \xRightarrow{z\langle\rangle} Q' \wedge Q' \not\to) \Rightarrow \\
 & & (P' \equiv_{!} \mathbf{0} \wedge Q' \equiv_{!} \mathbf{0}) \\
P \approx_{\mathrm{L}} Q :: z{:}A \multimap B[\omega] & \text{iff} & \forall P', y.\ (P \xRightarrow{z(y)} P') \Rightarrow \exists Q'.Q \xRightarrow{z(y)} Q'\ s.t. \\
 & & \forall R_1, R_2.\ R_1 \approx_{\mathrm{L}} R_2 :: y{:}A[\omega] \\
 & & (\nu y)(P' \mid R_1) \approx_{\mathrm{L}} (\nu y)(Q' \mid R_2) :: z{:}B[\omega] \\
P \approx_{\mathrm{L}} Q :: z{:}A \otimes B[\omega] & \text{iff} & \forall P', y.\ (P \xRightarrow{\overline{(\nu y)z\langle y\rangle}} P') \Rightarrow \exists Q'.Q \xRightarrow{\overline{(\nu y)z\langle y\rangle}} Q'\ s.t. \\
 & & \exists P_1, P_2, Q_1, Q_2.P' \equiv_{!} P_1 \mid P_2 \wedge Q' \equiv_{!} Q_1 \mid Q_2 \wedge \\
 & & P_1 \approx_{\mathrm{L}} Q_1 :: y{:}A[\omega] \wedge P_2 \approx_{\mathrm{L}} Q_2 :: z{:}B[\omega] \\
P \approx_{\mathrm{L}} Q :: z{:}!A[\omega] & \text{iff} & \forall P'.\ (P \xRightarrow{\overline{(\nu u)z\langle u\rangle}} P') \Rightarrow \exists Q', P_1, Q_1, y.Q \xRightarrow{\overline{(\nu u)z\langle u\rangle}} Q' \wedge \\
 & & P' \equiv_{!} !u(y).P_1 \wedge Q' \equiv_{!} !u(y).Q_1 \wedge P_1 \approx_{\mathrm{L}} Q_1 :: y{:}A[\omega] \\
P \approx_{\mathrm{L}} Q :: z{:}\nu X.A[\omega] & \text{iff} & (P,Q) \in \bigcup\{\Psi \in \mathbb{R} :: -{:}\nu X.A \mid \Psi \subseteq \cdot \approx_{\mathrm{L}} \cdot :: z{:}A[\omega[X \mapsto \Psi]]\}
\end{array}
$$

Figure 6.7: Logical Equivalence for Coinductive Session Types (base case) – abridged.

Lemma 35. *Suppose* $P \approx_{\mathrm{L}} Q :: z{:}A[\omega]$. *If* $\cdot \Rightarrow P \cong P' :: z{:}A$ *and* $\cdot \Rightarrow Q \cong Q' :: z{:}A$ *then* $P' \approx_{\mathrm{L}} Q' :: z{:}A[\omega]$.

Proof. By induction on the structure of A, using contextuality of $\cong$. □

Theorem 40 (Logical Equivalence is an Equivalence Candidate). *Relation* $P \approx_{\mathrm{L}} Q :: z{:}A[\omega]$ *is an equivalence candidate at* $z{:}A$ *in the sense of Definition 28.*

Proof. Follows from Lemmas 34 and 35. □

To establish the fixpoint property we define a similar operator to that of Def. 15.

Definition 31. *Let* $\nu X.A$ *be a strictly positive session type. We define the operator* $\phi_A : \mathbb{R} :: - : A \to \mathbb{R} :: - : A$ *as:*

$$\phi_A(s) \triangleq \cdot \approx_{\mathrm{L}} \cdot :: z{:}A[\omega[X \mapsto s]]$$

We first show the monotonicity of ϕ_A. The proof is identical to that of Lemma 28.

Lemma 36 (Monotonicity of ϕ_A). *The operator* ϕ_A *is monotonic, that is,* $s \subseteq s'$ *implies* $\phi_A(s) \subseteq \phi_A(s')$, *for any* s *and* s' *in* $\mathbb{R} :: -{:}A$.

Proof. By induction on A. □

The greatest fixpoint property follows as before.

Theorem 41 (Greatest Fixpoint). $\cdot \approx_{\mathrm{L}} \cdot :: z{:}\nu X.A[\omega]$ *is a greatest fixpoint of* ϕ_A.

Theorem 42. *Let $\nu X.A$ be a well-formed type:*

$$P \approx_{\mathrm{L}} Q :: z{:}A\{\nu X.A/X\}[\omega] \text{ iff } P \approx_{\mathrm{L}} Q :: z{:}A[\omega[X \mapsto \cdot \approx_{\mathrm{L}} \cdot :: -{:}\nu X.A[\omega]]]$$

Proof. By induction on A. □

Finally, we define the adequate notion of context representatives for this setting, which we may then use in the proof of the fundamental theorem.

Definition 32. *Let $\Gamma = u_1{:}B_1, \ldots, u_k{:}B_k$, and $\Delta = x_1{:}A_1, \ldots, x_n{:}A_n$ be a non-linear and a linear typing environment, respectively. Letting $I = \{1, \ldots, k\}$ and $J = \{1, \ldots, n\}$, with ω a mapping in the sense above. We define the sets of pairs of processes $\mathcal{C}^{\omega}_{\Gamma}$ and $\mathcal{C}^{\omega}_{\Delta}$ as:*

$$\mathcal{C}^{\omega}_{\Gamma} \overset{def}{=} \{ \prod_{i \in I} (!u_i(y_i).R_i, !u_i(y_i).S_i) \mid R_i \approx_{\mathrm{L}} S_i :: y_i{:}B_i[\omega] \}$$

$$\mathcal{C}^{\omega}_{\Delta} \overset{def}{=} \{ \prod_{j \in J} (Q_j, U_j) \mid Q_j \approx_{\mathrm{L}} U_j :: x_j{:}A_j[\omega] \}$$

Lemma 37. *Let P and Q be processes such that $\Gamma; \Delta \Rightarrow_{\eta} P :: T$, $\Gamma; \Delta \Rightarrow_{\eta} Q :: T$ with $\Gamma = u_1{:}B_1, \ldots, u_k{:}B_k$ and $\Delta = x_1{:}A_1, \ldots, x_n{:}A_n$, with ω compatible with η. We have:*

$$\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: T[\omega] \text{ iff } \forall (R_1, R_2) \in \mathcal{C}^{\omega}_{\Gamma}, \forall (S_1, S_2) \in \mathcal{C}^{\omega}_{\Delta},$$

$$(\nu \widetilde{u}, \widetilde{x})(P \mid R_1 \mid S_1) \approx_{\mathrm{L}} (\nu \widetilde{u}, \widetilde{x})(Q \mid R_2 \mid S_2) :: T[\omega]$$

Proof. Straightforward from Def. 29 and 32. □

The fundamental theorem of logical relations states that any well-typed process is logically equivalent to itself. The closure conditions pertaining to η (also present in the unary case) are necessary to maintain the inductive invariants for coinductive definitions.

Theorem 43 (Fundamental Theorem). *If $\Gamma; \Delta \Rightarrow_{\eta} P :: z{:}A$ then for any mapping ω consistent with η and for any mapping ρ such that $X \mapsto \Gamma_0; \Delta_0 \Rightarrow c_0{:}\nu Z.B \in \eta$ iff $\rho(X) \subseteq \Gamma_0; \Delta_0 \Rightarrow \cdot \approx_{\mathrm{L}} \cdot :: c_0{:}\nu Z.B[\omega]$ we have that $\Gamma; \Delta \Rightarrow \hat{\rho}(P) \approx_{\mathrm{L}} \hat{\rho}(P) :: z{:}A[\omega]$.*

Proof. By induction on typing. The proof is analogous to the unary case. The crucial case is the one where the last rule is $\nu\mathsf{R}$, where we again proceed by coinduction, exploiting termination of well-typed processes to ensure that actions can be matched with finite weak transitions.

We define a set $\mathcal{C}$ of pairs of processes, containing the corecursive definition and its unfolding, closed under the reduction, expansion and typed barbed congruence, such that it satisfies the equivalence candidate criteria at type $\nu X.A$. We then show that this set satisfies the property $\mathcal{C} \subseteq \cdot \approx_{\mathrm{L}} \cdot :: z{:}A[\omega[X \mapsto \mathcal{C}]]$.

As in the proof of Theorem 36, we make crucial use of the fact that process-level recursion can only be typed with the type variable rule, combined with guardedness and the inductive hypothesis to establish the property above.

The key insight is that since no internal interaction with recursive calls is allowed, we eventually result in a process that is by definition included in $\mathcal{C}$ – the recursive process itself, modulo renamings. This property would *not* hold otherwise, since it would be possible for internal actions to disrupt the recursive call such that it would not be possible to consistently construct a set $\mathcal{C}$ with the necessary properties. □

We have thus showed that our notion of logical equivalence for coinductively typed processes is sound with respect to typing insofar as any typed process is guaranteed to be equivalent to itself. Moreover, since logical equivalent is consistent with typed barbed congruence by construction, we may use this relation as a sound proof technique for reasoning about the observable behavior of typed processes.

6.3 Further Discussion

In this chapter we have developed our theory of linear logical relations as it applies to polymorphic and coinductive session types, noting that the theory applies directly to the simpler propositional setting of Chapter 2. We have presented formulations of the logical relations in both unary (predicate) and binary (relational) form, as a means of establishing properties of typed processes that may be expressed as predicates or binary relations. In particular, we have explored the idea of using the technique to establish termination and a form of process behavioral equivalence.

As we have seen throughout this chapter, our framework of linear logical relations naturally lends itself to generalization up to richer settings, seeing as we can maintain the overall structure of the development in a way that is consistent with the basic propositional type structure, impredicative polymorphism and coinductive session types (in both unary and binary formulations). This is made possible by the linear logic foundation that underlies our typing disciplines, enabling the development of this sophisticated proof technique in a natural and, arguably, uniform way.

A fundamental outcome of our linear logical relations framework is that it provides us with a flexible tool for reasoning about various forms of typed concurrent computation in a relatively straightforward way. Not only can we apply it in various typed settings, but it also enables us to establish a wide range of properties such as termination, equivalence or confluence (the latter is discussed in [59]), which have historically required much more sophisticated technical developments (typically hard to generalize across different languages or typing disciplines) or been outright absent from the literature (such as our parametricity result – Theorem 39 – for polymorphic session types).

Higher-Order Process Passing As we have mentioned at the start of Section 6.2.3, we have developed logical equivalence for coinductive session types in the absence of higher-order process passing features that were present in the language of Section 6.1.2 for the sake of technical simplicity. We sketch how to potentially extend the development to also account for the higher-order features. The key technical challenge is that the two-layer system of Section 6.1.2 requires us to reason about equivalence of both processes and (functional) terms, in a scenario where the functional language also embeds the process layer through the contextual monad.

For the functional layer, one way of developing observational equivalence is by considering a base type of answers to serve as the outcome of a computation in a distinguishable way. We would then define a notion of contextuality for functional terms and define observational equivalence by appealing to contexts of the "answer type". However, the functional language also includes monadic objects which cannot be discriminated within the functional language itself, but rather via the process-level spawn construct, so it is not at all clear how this approach of considering a type for answers would provide a consistent account of observational equivalence of monadic expressions (which cannot be inspected within the functional language itself). In principle, observational equivalence of monadic expressions should reduce to observational equivalence (i.e., typed barbed congruence) of processes. The key issue lies in the fact that barbed congruence would need to play the role of "observing" monadic objects, potentially also appealing to equivalence of functional terms. This mutual dependency makes defining reasonable notions of equivalence not at all obvious.

However, provided an hypothetically suitable notion of observational equivalence that integrates the two language layers, the definition of the logical equivalence should follow in a relatively straightforward way: for logical equivalence of monadic terms, appeal to logical equivalence of the underlying process; for logical equivalence of term input $\tau \supset A$ and output $\tau \wedge A$, appeal to the logical equivalence at the type τ accordingly.

Limitations and Potential Solutions While we have argued for our claim of uniformity and applicability of our framework, we note that our technique suffers from a similar limitation to that of logical relations for languages based on the λ-calculus, insofar as they may only be applied to so-called *total* languages, that is, languages without non-terminating programs.

There are multiple ways of addressing this challenge: the work developed in this chapter attempts to push the application boundaries of the formalism by giving an account of coinductive types, which encode the potential for infinite behavior in a non-divergent way that is amenable to a logical relations interpretation. This approach has natural limitations, since not only must we outright exclude any programs that include divergence but we must also eliminate programs that, despite being non-divergent, do not satisfy the criteria put in place to ensure termination (this is unavoidable since termination and productivity are undecidable problems in the general case). A way to potentially alleviate the latter is to develop more general admissibility criteria, for instance using type-based termination techniques [2, 39]. However, it is not clear how to map these techniques to our concurrent setting given that types describe *behavior* that is produced by processes rather than *data* that is constructed or inspected by functions. Type-based termination methods for functional languages typically use type annotations to place a (lower or upper) bound on the size of "values" of a given type. While this is perfectly reasonable when types describe data, it is not obvious how to apply this technique to processes and their behavior.

An alternative approach to that pursued in this thesis is to follow a different path altogether: instead of placing constraints on the language to "fit" the the formalism, we may extend the formalism to "fit" the language, naturally losing guarantees in the process. An example of this are so-called *step-indexed* logical relations for languages with non-termination [4], which are able to relate programs up-to a certain number of steps (thus the step-indexed prefix). Naturally, this technique cannot be used to ensure termination and is typically used to reason about a form of equivalence (or even simpler properties such as type safety).

Developing a step-indexed model in our setting can prove to be somewhat challenging from a technical perspective, given that step-indexed models typically conflate all non-terminating terms at a given type. This is natural in a functional setting where observables are the final values generated by evaluation, and thus no observable can be produced from a non-terminating term. In a process setting, observables are much more fine-grained insofar as we care about the interactive behavior of processes during their computation (even if it does not terminate) and not on a notion of final outcome *per se*. Therefore, conflating all non-terminating processes at a given type would result in a notion of equivalence that is unsatisfactory.

CHAPTER 7

APPLICATIONS OF LINEAR LOGICAL RELATIONS

In this chapter we continue to support our claims of using our logical foundation as a way to account for interesting phenomena of concurrent computation by developing applications of the linear logical relations techniques of the previous chapter to give an account of a few interesting aspects of concurrent computation.

In particular, we explore observational equivalence of processes. More precisely, we show that our notion of logical equivalence of the previous chapter may be used as a sound and complete proof technique for (typed) barbed congruence and show how to use it to reason about behaviorally equivalent processes. Moreover, we exploit the parametricity result of Section 6.2.2 to reason about the shape of processes in a way identical to that of the so-called "theorems for free" obtained from parametricity in System F [79].

Another interesting application of our framework of linear logical relations pertains to the familiar proof-theoretic concept of type isomorphisms, applied to session types (Section 7.2). In a concurrent setting, type isomorphisms are of particular interest since they describe a form of equivalence of behavior of processes offering *different* session types. Since session types are essentially communication protocol specifications, we can view session type isomorphisms as a form of equivalence between communication protocols which can be exploited for different purposes: develop coercions between related types; at some level, to optimize communication by considering simpler yet equivalent, or isomorphic, versions of types, etc.

Finally, we attempt to come full circle with respect to our concurrent interpretation of linear logic and show how the commuting conversions that arise during cut elimination, which do not correspond to reductions in our interpretation, can be explained via observational equivalences.

7.1 Behavioral Equivalence

One of the fundamental areas of study in concurrency theory is that of behavioral theory of concurrent processes, which addresses the issues related to behavioral characteristics of concurrent programs: how do we reason about the behavior of processes? under what circumstances can we consider that processes have the same behavior? what is an appropriate description of "behavior"?

All these questions have produced a vast collection of answers and proposed solutions. As we have mentioned in Section 6.2.1, the canonical notion of observational equivalence in process calculi is that of weak *barbed congruence*: a relation on processes that requires processes to exhibit the same (weak) *barbs* (actions along channels) in any context.

The main problem with barbed congruence is that despite being a reasonable notion of observational equivalence, it is hard to reason about in practice due to its quite "generous" definition, quantifying over all contexts and all process actions. Thus, the common approach is to develop some form of proof technique that is both sound and complete with respect to barbed congruence (and so, observational equivalence) and also arguably easier to reason about directly.

It turns out that in our logically based, well-typed setting, the logical equivalence obtained in Section 6.2.2 by generalizing our linear logical relations to the predicate setting characterizes barbed congruence and is thus a suitable proof technique for reasoning about observational equivalence of well-typed processes.

Lemma 38. *Logical equivalence (cf. Definition 25) is a contextual relation, in the sense of Definition 22.*

Proof. By induction on the clauses of Definition 22. □

Lemma 39. *If* $P \approx_{\mathrm{L}} Q :: z{:}A[\varnothing : \varnothing \leftrightarrow \varnothing]$ *then* $\cdot \vdash P \cong Q :: z{:}A$

Proof. Immediate from Lemma 31 (which ensures that $\approx_{\mathrm{L}}$ respects $\cong$) and reflexivity of $\cong$. □

Theorem 44 ($\approx_{\mathrm{L}}$ implies $\cong$ - Soundness)**.** *If* $\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \leftrightarrow \omega']$ *holds for any* $\omega, \omega' : \Omega$ *and* $\eta : \omega \leftrightarrow \omega'$*, then* $\Omega; \Gamma; \Delta \Rightarrow P \cong Q :: z{:}A$

Proof. Follows by contextuality of $\approx_{\mathrm{L}}$ (Lemma 38), which allows us to simplify the analysis to the case of an empty left-hand side typing environment. In that point, the thesis follows by Lemma 39. □

Theorem 45 ($\cong$ implies $\approx_{\mathrm{L}}$ - Completeness)**.** *If* $\Omega; \Gamma; \Delta \Rightarrow P \cong Q :: z{:}A$ *then* $\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: z{:}A[\eta : \omega \leftrightarrow \omega']$ *for some* $\omega, \omega' : \Omega$ *and* $\eta : \omega \leftrightarrow \omega'$.

Proof. Follows by combining Theorem 39 and Lemma 32. □

Theorems 44 and 45 then ensure:

Theorem 46 (Logical Equivalence and Barbed Congruence coincide)**.** *Relations* $\approx_{\mathrm{L}}$ *and* $\cong$ *coincide for well-typed processes.*

While the results above apply directly to polymorphic session types, it is immediate that this coincidence of logical equivalence and barbed congruence holds for the simple propositional setting as well.

7.1.1 Using Parametricity

We now illustrate a relatively simple application of our parametricity result for reasoning about concurrent polymorphic processes. For the sake of argument, we are interested in studying a restaurant finding system; such an application is expected to rely on some maps application, to be uploaded to a cloud server (c.f. the example of Section 3.5.1).

In our example, we would like to consider two different implementations of the system, each one relying on a different maps service. We assume that the two implementations will comply with the expected specification for the restaurant service, even if each one uses a different maps service (denoted by closed types AMaps and GMaps). This assumption may be precisely expressed by the judgment

$$s{:}!(\mathsf{api} \multimap X) \multimap !X \Rightarrow C_1 \approx_{\mathrm{L}} C_2 :: z{:}\mathsf{rest}[\eta_r : \omega_1 \Leftrightarrow \omega_2] \tag{7.1}$$

where $\eta_r(X) = \mathcal{R}, \omega_1(X) = \mathsf{AMaps}$, and $\omega_2(X) = \mathsf{GMaps}$, where $\mathcal{R}$ is an equivalence candidate that relates AMaps and GMaps, i.e., $\mathcal{R} : \mathsf{AMaps} \Leftrightarrow \mathsf{GMaps}$. The type of the restaurant finding application is denoted rest; it does not involve type variable X. Also, we assume X does not occur in the implementations C_1, C_2. Intuitively, the above captures the fact that C_1 and C_2 are similar "up to" the relation $\mathcal{R}$.

By exploiting the shape of type $\mathsf{CloudServer}$, we can ensure that *any* process S such that $\cdot \Rightarrow S :: s{:}\mathsf{CloudServer}$ behaves uniformly, offering the same generic behavior to its clients. That is to say, once the server is instantiated with an uploaded application, the behavior of the resulting system will depend only on the type provided by the application. Recall the polymorphic type of our cloud server:

$$\mathsf{CloudServer} \triangleq \forall X. !(\mathsf{api} \multimap X) \multimap !X$$

Based on the form of this type and combining inversion on typing and termination (Theorem 31), there is a process $SBody$ such that

$$S \xrightarrow{s(X)} SBody \qquad X; \cdot; \cdot \Rightarrow SBody :: s{:}!(\mathsf{api} \multimap X) \multimap !X \tag{7.2}$$

hold. By parametricity (Theorem 39) on (7.2), we obtain

$$\cdot \Rightarrow \hat{\omega}(SBody) \approx_{\mathrm{L}} \hat{\omega}'(SBody) :: s{:}!(\mathsf{api} \multimap X) \multimap !X[\eta : \omega \Leftrightarrow \omega']$$

for any ω, ω', and η. In particular, it holds for the η_r, ω_1 and ω_2 defined above:

$$\cdot \Rightarrow \hat{\omega}_1(SBody) \approx_{\mathrm{L}} \hat{\omega}_2(SBody) :: s{:}!(\mathsf{api} \multimap X) \multimap !X[\eta_r : \omega_1 \Leftrightarrow \omega_2] \tag{7.3}$$

By Definition 25, the formal relationship between C_1 and C_2 given by (7.1) implies

$$\cdot \Rightarrow (\boldsymbol{\nu} s)(\hat{\omega}_1(R_1) \mid C_1) \approx_{\mathrm{L}} (\boldsymbol{\nu} s)(\hat{\omega}_2(R_2) \mid C_2) :: z{:}\mathsf{rest}[\eta_r : \omega_1 \Leftrightarrow \omega_2]$$

for any R_1, R_2 such that $R_1 \approx_{\mathrm{L}} R_2 :: s{:}!(\mathsf{api} \multimap X) \multimap !X[\eta_r : \omega_1 \Leftrightarrow \omega_2]$. In particular, it holds for the two processes related in (7.3) above. Combining these two facts, we have:

$$\cdot \Rightarrow (\boldsymbol{\nu} s)(\hat{\omega}_1(SBody) \mid C_1) \approx_{\mathrm{L}} (\boldsymbol{\nu} s)(\hat{\omega}_2(SBody) \mid C_2) :: z{:}\mathsf{rest}[\eta_r : \omega_1 \Leftrightarrow \omega_2]$$

Since rest does not involve X, using Theorem 38 we actually have:

$$\cdot \Rightarrow (\boldsymbol{\nu} s)(\hat{\omega}_1(SBody) \mid C_1) \approx_{\mathrm{L}} (\boldsymbol{\nu} s)(\hat{\omega}_2(SBody) \mid C_2) :: z{:}\mathsf{rest}[\emptyset : \emptyset \Leftrightarrow \emptyset] \tag{7.4}$$

Now, given (7.4), and using backward closure of $\approx_{\mathrm{L}}$ under reductions (possible because of Theorem 37 and Definition 24), we obtain:

$$\cdot \Rightarrow (\boldsymbol{\nu} s)(S \mid \overline{s}\langle \mathsf{AMaps}\rangle.C_1) \approx_{\mathrm{L}} (\boldsymbol{\nu} s)(S \mid \overline{s}\langle \mathsf{GMaps}\rangle.C_2) :: z{:}\mathsf{rest}[\emptyset : \emptyset \Leftrightarrow \emptyset]$$

Then, using Corollary 46, we finally have

$$\cdot \Rightarrow (\boldsymbol{\nu} s)(S \mid \overline{s}\langle \mathsf{AMaps}\rangle.C_1) \cong (\boldsymbol{\nu} s)(S \mid \overline{s}\langle \mathsf{GMaps}\rangle.C_2) :: z{:}\mathsf{rest}$$

This simple, yet illustrative example shows how one may use our parametricity results to reason about the observable behavior of concurrent systems that interact under a polymorphic behavioral typing discipline. This style of reasoning is reminiscent of the so-called "theorems for free" reasoning using parametricity in System F [79], which provides a formal and precise justification for the intuitive concept that typing (especially in the polymorphic setting) combined with linearity introduces such strong constraints on the shape of processes that we can determine the observable behavior of processes very precisely by reasoning directly about their session typings.

7.2 Session Type Isomorphisms

In type theory, types A and B are called *isomorphic* if there are morphisms π_A of $B \vdash A$ and π_B of $A \vdash B$ which compose to the identity in both ways—see, e.g., [27]. For instance, in typed λ-calculus the types $A \times B$ and $B \times A$ (for any type A and B) are isomorphic since we can construct terms $M = \lambda x{:}A \times B.\langle \pi_2 x, \pi_1 x\rangle$ and $N = \lambda x{:}B \times A.\langle \pi_2 x, \pi_1 x\rangle$, respectively of types $A \times B \to B \times A$ and $B \times A \to A \times B$, such that both compositions $\lambda x{:}B \times A.(M\,(N\,x))$ and $\lambda x{:}A \times B.(N\,(M\,x))$ are equivalent (up to $\beta\eta$-conversion) to the identity $\lambda x{:}B \times A.x$ and $\lambda x{:}A \times B.x$.

We adapt this notion to our setting, by using proofs (processes) as morphisms, and by using logical equivalence to account for *isomorphisms* in linear logic. Given a sequence of names $\widetilde{x} = x_1, \ldots, x_n$, below we write $P^{\langle \widetilde{x} \rangle}$ to denote a process such that $\mathit{fn}(P) = \{x_1, \ldots, x_n\}$.

Definition 33 (Type Isomorphism). *Two (session) types A and B are called* isomorphic*, noted $A \simeq B$, if, for any names x, y, z, there exist processes $P^{\langle x,y \rangle}$ and $Q^{\langle y,x \rangle}$ such that:*

(i) $\cdot\,; x{:}A \Rightarrow P^{\langle x,y \rangle} :: y{:}B;$

(ii) $\cdot\,; y{:}B \Rightarrow Q^{\langle y,x \rangle} :: x{:}A;$

(iii) $\cdot\,; x{:}A \Rightarrow (\boldsymbol{\nu} y)(P^{\langle x,y \rangle} \mid Q^{\langle y,z \rangle}) \approx_{\mathrm{L}} [x \leftrightarrow z] :: z{:}A;$ *and*

(iv) $\cdot\,; y{:}B \Rightarrow (\boldsymbol{\nu} x)(Q^{\langle y,x \rangle} \mid P^{\langle x,z \rangle}) \approx_{\mathrm{L}} [y \leftrightarrow z] :: z{:}B.$

Intuitively, if A, B are service specifications then by establishing $A \simeq B$ one can claim that having A is as good as having B, given that we can construct one from the other using an isomorphism. Isomorphisms in linear logic can then be used to simplify/transform concurrent service interfaces. Moreover, they help validate our interpretation with respect to basic linear logic principles. As an example, let us consider multiplicative conjunction $\otimes$. A basic linear logic principle is $A \otimes B \vdash B \otimes A$. Our interpretation of $A \otimes B$ may appear asymmetric as, in general, a channel of type $A \otimes B$ is not typable by $B \otimes A$. Theorem 47 below states the symmetric nature of $\otimes$ as a type isomorphism: symmetry is realized by a process which *coerces* any session of type $A \otimes B$ to a session of type $B \otimes A$.

Theorem 47. *Let A, B, and C be any type. Then the following hold:*

$$\begin{array}{rrcl}
\textit{(i)} & A \otimes B & \simeq & B \otimes A \\
\textit{(ii)} & A \otimes (B \otimes C) & \simeq & (A \otimes B) \otimes C \\
\textit{(iii)} & A \multimap (B \multimap C) & \simeq & (A \otimes B) \multimap C \\
\textit{(iv)} & A \otimes \mathbf{1} & \simeq & A \\
\textit{(v)} & \mathbf{1} \multimap A & \simeq & A \\
\textit{(vi)} & (A \oplus B) \multimap C & \simeq & (A \multimap C) \,\&\, (B \multimap C) \\
\textit{(vii)} & !(A \,\&\, B) & \simeq & !A \otimes !B
\end{array}$$

Proof. We give details for the proof of *(i)* above. We check conditions (i)-(iv) of Def. 33 for processes $P^{\langle x,y\rangle}, Q^{\langle y,x\rangle}$ defined as

$$\begin{array}{rcl}
P^{\langle x,y\rangle} & = & x(u).(\nu n)\overline{y}\langle n\rangle.([x \leftrightarrow n] \mid [u \leftrightarrow y]) \\
Q^{\langle y,x\rangle} & = & y(w).(\nu m)\overline{x}\langle m\rangle.([y \leftrightarrow m] \mid [w \leftrightarrow x])
\end{array}$$

Checking (i)-(ii), i.e., $\cdot\,; x{:}A \otimes B \Rightarrow P^{\langle x,y\rangle}{::}y{:}B \otimes A$ and $\cdot\,; y{:}B \otimes A \Rightarrow Q^{\langle y,x\rangle}{::}x{:}A \otimes B$ is easy; rule (id) ensures that both typings hold for any A, B. We sketch only the proof of (iii); the proof of (iv) is analogous. Let $M = (\nu y)(P^{\langle x,y\rangle} \mid Q^{\langle y,z\rangle})$ and $N = [x \leftrightarrow z]$; we need to show $\cdot\,; x{:}A \otimes B \Rightarrow M \approx_{\mathrm{L}} N :: z{:}A \otimes B$. We thus have to show that for any $R_1 \approx_{\mathrm{L}} R_2 :: x{:}A \otimes B$, we have $\Rightarrow\ (\nu x)(R_1 \mid M) \approx_{\mathrm{L}} (\nu x)(R_2 \mid N) :: z{:}A \otimes B$.

Let $(\nu x)(R_1 \mid M) = S_1$ and $(\nu x)(R_2 \mid N) = S_2$. We first observe that since $R_1 \approx_{\mathrm{L}} R_2 :: x{:}A \otimes B$, combined with termination, type preservation and progress, we know that $S_1 \rightarrow (\nu x, u)(R_1' \mid (\nu y)((\nu n)\overline{y}\langle n\rangle.([x \leftrightarrow n] \mid [u \leftrightarrow y]) \mid y(w).(\nu m)\overline{z}\langle m\rangle.([y \leftrightarrow m] \mid [w \leftrightarrow z])) \Longrightarrow (\nu x)(R_1' \mid (\nu m)\overline{z}\langle m\rangle.([u \leftrightarrow m] \mid [x \leftrightarrow z]))$, such that $R_2 \overset{\overline{(\nu u)x\langle u\rangle}}{\Longrightarrow} R_2'$ where $R_1' \equiv_! T_1 \mid T_2$ and $R_2' \equiv_! U_1 \mid U_2$ such that $T_1 \approx_{\mathrm{L}} U_1 :: u{:}A$ and $T_2 \approx_{\mathrm{L}} U_2 :: x{:}B$. We conclude by observing that $(\nu x)(R_1' \mid (\nu m)\overline{z}\langle m\rangle.([u \leftrightarrow m] \mid [x \leftrightarrow z])) \overset{\overline{(\nu m)z\langle m\rangle}}{\Longrightarrow} R_1'\{m/u, z/x\}$, $(\nu x)(R_2 \mid N) \rightarrow R_2\{z/x\}$ and α-converting u in R_2 to m. Since we know that $\approx_{\mathrm{L}}$ is preserved under renaming, we are done. □

The type isomorphisms listed above are among those known of linear logic, which given our very close correspondence with the proof theory of linear logic is perhaps not surprising. We note that we do not exhaustively list all isomorphisms for the sake of conciseness (e.g. both & and $\oplus$ are also associative).

Beyond the proof-theoretic implications of type isomorphisms, it is instructive to consider the kinds of coercions they imply at the process level. The case of session type isomorphism (i), for which we produce the appropriate witnesses in the proof detailed above, requires a sort of "criss-cross" coercion, where taking a session of type $x{:}A \otimes B$ to one of type $y{:}B \otimes A$ entails receiving the session of type $u{:}A$ from x (which is now offering B) and then sending, say, z along y, after which we link z with x and y with u.

Another interesting isomorphism is (iii) above, which specifies a natural form of currying and uncurrying: a session that is meant to receive a session of type A followed by another of type B to then proceed as C can be coerced to a session that instead receives one of type $A \otimes B$ and then proceeds as C. We do this by considering $x{:}A \multimap (B \multimap C)$, to be coerced to $y{:}(A \otimes B) \multimap C$. We will first input along y a session $u{:}A \otimes B$, from which we may consume an output to obtain $z{:}A$ and $u{:}B$. These two channels are then sent (via fresh output followed by forwarding) to x, at which point we may forward between x to y. It is relatively easy to see how to implement the coercion in the converse direction (i.e. input the A and B sessions, construct a session $A \otimes B$, send it to the ambient session and forward). An interesting application of this isomorphism is that it enables us to, at the *interface level*, replace two inputs (the A followed by the B) with a single input (provided we do some additional work internally).

Finally, consider the perhaps peculiar looking isomorphism (vi), which identifies coercions between sessions of type $T_1 = (A \oplus B) \multimap C$ and $T_2 = (A \multimap C) \& (B \multimap C)$. The former denotes the capability to input a session that will either behave as A or B, which is then used to produce the behavior C. The latter denotes the capability to offer a choice between a session that inputs A to produce C and the ability to input B to produce C. Superficially, these appear to be very different session behaviors, but turn out to be easily coerced from one to the other: Consider a session $x{:}T_1$ to be coerced to $y{:}T_2$. We begin by offering the choice between the two alternatives. For the left branch, we input along y a session $z{:}A$ which we can use to send to x (by constructing the appropriate session) and then forward between x and y. The right branch is identical.

This isomorphism codifies the following informal intuition: Imagine you wish to implement a service whose sole aim is to offer some behavior C by receiving some payment from its clients. Since you wish to provide a flexible service, you specify that clients are able to choose between two payment methods, say, A and B. Therefore, you want to expose to the world a service of type $(A \oplus B) \multimap C$. However, you wish to organize the internal structure of your service such that you define payment "handlers" individually, that is, you specify a handler for payment method A and another for payment method B. By appealing to the session coercions above, you can mediate between these two quite different (yet isomorphic) representations of the same abstract service.

7.3 Soundness of Proof Conversions

One of the key guiding principles in our concurrent interpretation of linear logic (Chapters 2 and 3) is the identification of proof reduction with process expression (and π-calculus process) reduction. In proof theory, proof reduction in this sense is generated from the computational contents of the proof of cut admissibility (or cut elimination from a proof system with a cut rule). Throughout our development, we have repeatedly derived the appropriate reduction semantics of our language constructs from inspection of the proof reduction that arises when we attempt to simplify, or reduce, a cut of a right and a left rule of a given connective (a *principal cut*). However, the computational process of cut elimination is not just that of single-step reduction but of *full normalization*: where a single reduction eliminates one cut; cut elimination, as the name entails, removes all instances of cut from a given proof.

To achieve this, cut elimination must include not only the one-step reductions that arise from a cut of a right and a left rule on the same connective, but also a series of other computational steps that permute cuts upwards in a proof. These steps, generally known as *commuting conversions*, are necessary to produce a full normalization procedure since in general we are not forced to immediately use an assumption that is introduced by a cut. Consider, for instance, the following (unnatural, yet valid) proof:

$$\dfrac{A \mathbin{\&} B \vdash B \mathbin{\&} A \qquad \dfrac{\dfrac{A \vdash A}{B \mathbin{\&} A \vdash A}\,(\&\mathsf{L}_2)}{\mathbf{1}, B \mathbin{\&} A \vdash A}\,(\mathbf{1}\mathsf{L})}{A \mathbin{\&} B, \mathbf{1} \vdash A}\,(\mathsf{cut})$$

The instance of the cut rule above introduces $B \mathbin{\&} A$ (we omit the proof of $A \mathbin{\&} B \vdash B \mathbin{\&} A$ for conciseness) in the right premise, but instead of applying the $\&\mathsf{L}_2$ rule immediately, we first apply the $\mathbf{1}\mathsf{L}$ rule to consume the $\mathbf{1}$ from the context. Thus, this does not consist of a principal cut. For cut elimination to hold, we must be able to push the cut upwards in the derivation as follows:

$$\dfrac{\dfrac{A \mathbin{\&} B \vdash B \mathbin{\&} A \qquad \dfrac{A \vdash A}{B \mathbin{\&} A \vdash A}\,(\&\mathsf{L}_2)}{A \mathbin{\&} B \vdash A}\,(\mathsf{cut})}{A \mathbin{\&} B, \mathbf{1} \vdash A}\,(\mathbf{1}\mathsf{L})$$

In the proof above, we have pushed the instance of cut upwards in the derivation, first applying the $\mathbf{1}\mathsf{L}$ rule. This new proof can then be simplified further by applying the appropriate cut reduction *under* the instance of the $\mathbf{1}\mathsf{L}$ rule.

This form of proof transformation is present throughout the entirety of the cut elimination procedure, but up to this point has no justification in our concurrent interpretation of linear logic. Indeed, if we consider the process expression assignment for these proof transformations, not only are the process expressions quite different but their relationship is not immediately obvious. In the two proofs above we have the following process expression assignment (where $x{:}A \mathbin{\&} B \vdash P :: w{:}B \mathbin{\&} A$):

$$x{:}A \mathbin{\&} B, y{:}\mathbf{1} \vdash \mathsf{new}\ w.(P \parallel \mathsf{wait}\ y; w.\mathsf{inr}; \mathsf{fwd}\ z\ w) :: z{:}A$$

$$x{:}A \mathbin{\&} B, y{:}\mathbf{1} \vdash \mathsf{wait}\ y; \mathsf{new}\ w.(P \parallel w.\mathsf{inr}; \mathsf{fwd}\ z\ w) :: z{:}A$$

The process expression for the first proof consists of a parallel composition of two processes expressions, whereas the expression for the second first waits on y before proceeding with the parallel composition. Superficially, it would seem as if the behavior of the two process expression is quite different and thus there is no apparent connection between the two, despite the fact that they correspond to linear logic proofs that are related via a proof conversion in cut elimination.

It turns out that we can give a suitable justification to this notion of proof conversion in our concurrent framework by appealing to *observational equivalence*, via our notion of logical equivalence. It may strike the reader as somewhat odd that the two process expressions above result in observationally equivalent behaviors, but the key insight is the fact that they are *open* process expressions which produce equivalent behaviors when placed in the appropriate (typed) contexts. If we compose the given process expressions with others offering sessions $x{:}A \mathbin{\&} B$ and $y{:}\mathbf{1}$, the end result consists of two closed process expression, both offering $z{:}A$. Since we can only observe behavior along session channel z, we cannot distinguish the two closed processes due to the fact that the apparent mismatch in the order of actions occurs as internal reductions which are *not* visible.

Proof Conversions We now make the informal argument above precise. We begin by formally introducing the proof transformations that arise in cut-elimination which are not identified by reductions on process expressions or permutations in execution states (the latter consist of permutations of multiple instances of the cut rule), which we denote as *proof conversions*. For conciseness and notational convenience due to the fact that logical equivalence is defined on π-calculus processes, we illustrate proof conversions directly on π-calculus processes. It is straightforward to map them back to their original process expressions. Moreover, we illustrate only the conversions of the propositional setting of Chapter 2, although similar reasoning applies to proof conversions from other settings.

Proof conversions induce a congruence on typed processes, which we denote $\simeq_c$. Figure 7.1 presents a sample of process equalities extracted from them, for the sake of conciseness; the full list can be found in A.5. Each equality $P \simeq_c Q$ in the figure is associated to appropriate right- and left-hand side typings; this way, e.g., equality (7.18) in Figure 7.1—related to two applications of rule ($\otimes$L) —could be stated as

$$\Gamma; \Delta, x{:}A \otimes B, z{:}C \otimes D \Rightarrow x(y).z(w).P \simeq_c z(w).x(y).P :: T$$

where Γ and Δ are typing contexts, A, B, C, D are types, and T is a right-hand side typing. For the sake of conciseness, we omit the full typings, emphasizing the differences in the concurrent assignment of the proofs.

As we have argued above, proof conversions describe the interplay of two rules in a type-preserving way: regardless of the order in which the two rules are applied, they lead to typing derivations with the same right- and left-hand side typings, but with syntactically differing assignments.

Definition 34 (Proof Conversions)**.** *We define $\simeq_c$ as the least congruence on processes induced by the equalities in Figures A.1, A.2, A.3, and A.4.*

We separate proof conversions into five classes, denoted (A)–(E):

$$
\begin{aligned}
(\boldsymbol{\nu} x)(P \mid (\boldsymbol{\nu} y)z\langle y\rangle.(Q \mid R)) &\simeq_c (\boldsymbol{\nu} y)z\langle y\rangle.((\boldsymbol{\nu} x)(P \mid Q) \mid R) \quad &(7.5)\\
(\boldsymbol{\nu} x)(P \mid y(z).Q) &\simeq_c y(z).(\boldsymbol{\nu} x)(P \mid Q) \quad &(7.6)\\
(\boldsymbol{\nu} x)(P \mid y.\mathsf{inl}; Q) &\simeq_c y.\mathsf{inl}; (\boldsymbol{\nu} x)(P \mid Q) \quad &(7.7)\\
(\boldsymbol{\nu} x)(P \mid (\boldsymbol{\nu} y)u\langle y\rangle.Q) &\simeq_c (\boldsymbol{\nu} y)u\langle y\rangle.(\boldsymbol{\nu} x)(P \mid Q) \quad &(7.8)\\
(\boldsymbol{\nu} x)(P \mid y.\mathsf{case}(Q, R)) &\simeq_c y.\mathsf{case}((\boldsymbol{\nu} x)(P \mid Q), (\boldsymbol{\nu} x)(P \mid R)) \quad &(7.9)\\
(\boldsymbol{\nu} u)(!u(y).P \mid z\langle\rangle.\mathbf{0}) &\simeq_c z\langle\rangle.\mathbf{0} \quad &(7.10)\\
(\boldsymbol{\nu} u)(!u(y).P \,|\, (\boldsymbol{\nu} z)x\langle z\rangle.(Q \,|\, R)) &\simeq_c (\boldsymbol{\nu} z)x\langle z\rangle.\big((\boldsymbol{\nu} u)(!u(y).P \,|\, Q) \,|\, (\boldsymbol{\nu} u)(!u(y).P \,|\, R)\big) \quad &\\
&&(7.11)\\
(\boldsymbol{\nu} u)((!u(y).P) \mid y(z).Q) &\simeq_c y(z).(\boldsymbol{\nu} u)((!u(y).P) \mid Q) \quad &(7.12)\\
(\boldsymbol{\nu} u)((!u(z).P) \mid y.\mathsf{inl}; Q) &\simeq_c y.\mathsf{inl}; (\boldsymbol{\nu} u)((!u(z).P) \mid Q) \quad &(7.13)\\
(\boldsymbol{\nu} u)(!u(z).P \mid y.\mathsf{case}(Q, R)) &\simeq_c y.\mathsf{case}\big((\boldsymbol{\nu} u)(!u(z).P \mid Q), (\boldsymbol{\nu} u)(!u(z).P \mid R)\big) \quad &(7.14)\\
(\boldsymbol{\nu} u)(!u(y).P \mid !x(z).Q) &\simeq_c !x(z).(\boldsymbol{\nu} u)(!u(y). \mid Q) \quad &(7.15)\\
(\boldsymbol{\nu} u)(!u(y).P \mid (\boldsymbol{\nu} y)v\langle y\rangle.Q) &\simeq_c (\boldsymbol{\nu} y)v\langle y\rangle.(\boldsymbol{\nu} u)(!u(y).P \mid Q)) \quad &(7.16)\\
(\boldsymbol{\nu} w)z\langle w\rangle.(R \mid (\boldsymbol{\nu} y)x\langle y\rangle.(P \mid Q)) &\simeq_c (\boldsymbol{\nu} y)x\langle y\rangle.(P \mid (\boldsymbol{\nu} w)z\langle w\rangle.(R \mid Q)) \quad &(7.17)\\
x(y).z(w).P &\simeq_c z(w).x(y).P \quad &(7.18)
\end{aligned}
$$

Figure 7.1: A sample of process equalities induced by proof conversions

(A) *Permutations between rule (*cut*) and a right or left rule.* This class of conversions represents the interaction of a process *offering* a session C on x, with some process *requiring* such a session (as illustrated in the example above where we commute a cut with the $\mathbf{1}\mathsf{L}$ rule); this process varies according to the particular rule considered. As an example, the following inference represents the interplay of rules ($\multimap$L) and (cut):

$$
\dfrac{\Gamma; \Delta_1 \Rightarrow P :: x{:}C \qquad \dfrac{\Gamma; \Delta_2, x{:}C \Rightarrow Q :: z{:}A \quad \Gamma; \Delta_3, y{:}B \Rightarrow R :: T}{\Gamma; \Delta_2, \Delta_3, x{:}C, y{:}A \multimap B \Rightarrow (\boldsymbol{\nu} z)y\langle z\rangle.(Q \mid R) :: T}\,(\multimap\mathsf{L})}{\Gamma; \Delta, y{:}A \multimap B \Rightarrow (\boldsymbol{\nu} x)(P \mid (\boldsymbol{\nu} z)y\langle z\rangle.(Q \mid R)) :: T}\,(\mathsf{cut})
$$

where $\Delta = \Delta_1, \Delta_2, \Delta_3$. Permutability is justified by the following inference:

$$
\dfrac{\dfrac{\Gamma; \Delta_1 \Rightarrow P :: x{:}C \quad \Gamma; \Delta_2, x{:}C \Rightarrow Q :: z{:}A}{\Gamma; \Delta_1, \Delta_2 \Rightarrow (\boldsymbol{\nu} x)(P \mid Q) :: z{:}A}\,(\mathsf{cut}) \qquad \Gamma; \Delta_3, y{:}B \Rightarrow R :: T}{\Gamma; \Delta, y{:}A \multimap B \Rightarrow (\boldsymbol{\nu} z)y\langle z\rangle.((\boldsymbol{\nu} x)(P \mid Q) \mid R) :: T}\,(\multimap\mathsf{L})
$$

This class of permutations is given by the equalities in Fig. A.1.

(B) *Permutations between rule (*cut*) and a left rule.* In contrast to permutations in class (A), this class of conversions represents the interaction of a process *requiring* a session C on x, with some process *offering* such a session. This distinction is due to the shape of rule (cut). To see the difference with the permutations in class (A), consider the inferences given above with those for the permutation below, which also concerns rules ($\multimap$L) and (Tcut). Letting

$\Delta = \Delta_1, \Delta_2, \Delta_3$, we have:

$$\dfrac{\dfrac{\Gamma;\Delta_1 \Rightarrow P :: z{:}A \quad \Gamma;\Delta_2, y{:}B \Rightarrow Q :: x{:}C}{\Gamma;\Delta_1,\Delta_2, y{:}A \multimap B \Rightarrow (\nu z)y\langle z\rangle.(P \mid Q) :: x{:}C}\,(\multimap\mathsf{L}) \quad \Gamma;\Delta_3, x{:}C \vdash R :: T}{\Gamma;\Delta, y{:}A \multimap B \Rightarrow\ (\nu x)((\nu z)y\langle z\rangle.(P \mid Q) \mid R) :: T}\,(\mathsf{cut})$$

Permutability is then justified by the following inference:

$$\dfrac{\Gamma;\Delta_1 \Rightarrow P :: z{:}A \qquad \dfrac{\Gamma;\Delta_2, y{:}B \Rightarrow Q :: x{:}C \quad \Gamma;\Delta_3, x{:}C \Rightarrow R :: T}{\Gamma;\Delta_2,\Delta_3, y{:}B \Rightarrow (\nu x)(Q \mid R) :: T}\,(\mathsf{cut})}{\Gamma;\Delta, y{:}A \multimap B \Rightarrow (\nu z)y\langle z\rangle.(P \mid (\nu x)(Q \mid R)) :: T}\,(\multimap\mathsf{L})$$

This class of permutations is given by the equalities in Fig. A.1.

(C) *Permutations between rule* ($\mathsf{cut}^!$) *and a right or left rule.* This class of permutations is analogous to the classes (A) and (B), but considering rule ($\mathsf{cut}^!$) instead of (cut). As an example, the following permutation concerns the interplay of rule ($\mathsf{cut}^!$) with rule ($\oplus\mathsf{R}_1$):

$$\dfrac{\dfrac{\Gamma;\cdot \Rightarrow P :: y{:}C \quad \Gamma, u{:}C;\Delta \Rightarrow Q :: z{:}A}{\Gamma;\Delta \Rightarrow (\nu u)(!u(y).P \mid Q) :: z{:}A}\,(\mathsf{cut}^!)}{\Gamma;\Delta \Rightarrow z.\mathsf{inl};(\nu u)(!u(y).P \mid Q) :: z{:}A \oplus B}\,(\oplus\mathsf{R}_1)$$

Then, permutability is justified by the following inference:

$$\dfrac{\Gamma;\cdot \Rightarrow P :: y{:}C \quad \dfrac{\Gamma, u{:}C;\Delta \Rightarrow Q :: z{:}A}{\Gamma, u{:}C;\Delta \Rightarrow z.\mathsf{inl};Q :: z{:}A \oplus B}\,(\oplus\mathsf{R}_1)}{\Gamma;\Delta \Rightarrow (\nu u)(!u(y).P \mid z.\mathsf{inl};Q) :: z{:}A \oplus B}\,(\mathsf{cut}^!)$$

This class of permutations is given by the equalities in Fig. A.2.

(D) *Permutations between two left rules.* Classes (A)–(C) consider permutations in which one of the involved rules is some form of cut. Permutations which do not involve cuts are also possible; they represent type-preserving transformations for prefixes corresponding to independent (i.e. non-interfering) sessions. We consider permutations involving two left rules. As an example, the permutation below concerns the interplay of rule ($\oplus$L) with rule ($\otimes$L):

$$\dfrac{\dfrac{\Gamma;\Delta, z{:}C, x{:}B, y{:}A \Rightarrow P :: T}{\Gamma;\Delta, z{:}C, x{:}A \otimes B \Rightarrow x(y).P :: T}\,(\otimes\mathsf{L}) \quad \dfrac{\Gamma;\Delta, z{:}D, x{:}B, y{:}A \Rightarrow P :: T}{\Gamma;\Delta, z{:}D, x{:}A \otimes B \Rightarrow x(y).Q :: T}\,(\otimes\mathsf{L})}{\Gamma;\Delta, z{:}C \oplus D, x{:}A \otimes B \Rightarrow z.\mathsf{case}(x(y).P, x(y).Q) :: T}\,(\oplus\mathsf{L})$$

Permutability is justified by the following inference:

$$\dfrac{\dfrac{\Gamma;\Delta, z{:}C, x{:}B, y{:}A \Rightarrow P :: T \quad \Gamma;\Delta, z{:}D, x{:}B, y{:}A \Rightarrow P :: T}{\Gamma;\Delta, z{:}C \oplus D, x{:}B, y{:}A \Rightarrow z.\mathsf{case}(P, Q) :: T}\,(\oplus\mathsf{L})}{\Gamma;\Delta, z{:}C \oplus D, x{:}A \otimes B \Rightarrow x(y).z.\mathsf{case}(P, Q) :: T}\,(\oplus\mathsf{L})$$

This class is given by the equalities in Fig. A.3.

(E) *Permutations between a left and a right rule.* This class of permutations also involves rules acting on two independent sessions: one rule acts on the left-hand side of the derivation while the other acts on the right-hand side. As an example, the permutation below concerns the interplay of rules ($\&\mathsf{L}_1$) and ($\otimes\mathsf{R}$):

$$\dfrac{\dfrac{\Gamma;\Delta_1,z{:}C \Rightarrow P :: y{:}A \quad \Gamma;\Delta_2 \Rightarrow Q :: x{:}B}{\Gamma;\Delta,z{:}C \Rightarrow (\boldsymbol{\nu} y)x\langle y\rangle.(P \mid Q) :: x{:}A\otimes B}\,(\otimes\mathsf{R})}{\Gamma;\Delta,z{:}C\,\&\,D \Rightarrow z.\mathsf{inl};(\boldsymbol{\nu} y)x\langle y\rangle.(P \mid Q) :: x{:}A\otimes B}\,(\&\mathsf{L}_1)$$

where $\Delta = \Delta_1,\Delta_2$. Permutability is justified by the inference:

$$\dfrac{\dfrac{\Gamma;\Delta_1,z{:}C \Rightarrow P :: z{:}A}{\Gamma;\Delta,z{:}C\,\&\,D \Rightarrow z.\mathsf{inl};P :: y{:}A}\,(\&\mathsf{L}_1) \quad \Gamma;\Delta_2 \Rightarrow Q :: x{:}B}{\Gamma;\Delta,z{:}C\,\&\,D \Rightarrow (\boldsymbol{\nu} y)x\langle y\rangle.(z.\mathsf{inl};P \mid Q) :: x{:}A\otimes B}\,(\otimes\mathsf{R})$$

This class is given by the equalities in Fig. A.4.

Having fully presented the notion of proof conversions, we show that the congruence $\simeq_c$ induced on processes by proof conversions is sound with respect to logical equivalence, and thus observational equivalence.

Again, the key insight is that the equalities generated by $\simeq_c$ concern *open* processes, where we are able to permute certain actions in a type-safe way due to the fact that, once placed in the appropriate typed contexts, the permuted actions are not observable. This can be readily seen, for instance, in the equation,

$$\Gamma;\Delta,x{:}A\otimes B,z{:}C\otimes D \Rightarrow x(y).z(w).P \simeq_c z(w).x(y).P :: v{:}E$$

which allows for two input prefixes along independent ambient session to be commuted. When we consider the two processes placed in the appropriate contexts, the actions along session channels x and z are not visible, only those along v, therefore making it reasonable to commute the two input prefixes.

Theorem 48 (Soundness of Proof Conversions)**.** *Let P,Q be processes such that:*

(i) $\Gamma;\Delta \Rightarrow P :: z{:}A$*,*

(ii) $\Gamma;\Delta \Rightarrow Q :: z{:}A$ *and,*

(iii) $P \simeq_c Q$*.*

We have that $\Gamma;\Delta \Rightarrow P \approx_{\mathrm{L}} Q :: z{:}A[\varnothing : \varnothing \Leftrightarrow \varnothing]$

Proof. By case analysis on proof conversions. We exploit termination of well-typed processes to ensure that actions can be matched with finite weak transitions.

We detail the case for the first proof conversion in Figure A.1 —see A.5.1 for other cases. This proof conversion corresponds to the interplay of rules ($\otimes\mathsf{R}$) and (cut). We have to show that $\Gamma;\Delta \Rightarrow M \approx_{\mathrm{L}} N :: z{:}A\otimes B[\varnothing : \varnothing \Leftrightarrow \varnothing]$ where

$$\Delta = \Delta_1,\Delta_2,\Delta_3 \quad \Gamma;\Delta_1 \Rightarrow P_1 :: x{:}C \quad \Gamma;\Delta_2,x{:}C \Rightarrow P_2 :: y{:}A \quad \Gamma;\Delta_3 \Rightarrow P_3 :: z{:}B \tag{7.19}$$

$$M = (\boldsymbol{\nu} x)(P_1 \mid (\boldsymbol{\nu} y)z\langle y\rangle.(P_2 \mid P_3)) \quad N = (\boldsymbol{\nu} y)z\langle y\rangle.((\boldsymbol{\nu} x)(P_1 \mid P_2) \mid P_3)$$

By Lemma 37 we have to show that for any $(R_1,R_2) \in \mathcal{C}_\Gamma$ and any $(S_1,S_2) \in \mathcal{C}_\Delta$ we have:

$$(\nu\tilde{x},\tilde{u})(R_1 \mid S_1 \mid M) \approx_{\mathrm{L}} (\nu\tilde{x},\tilde{u})(R_2 \mid S_2 \mid N) :: z{:}A \otimes B$$

Suppose $(\nu\tilde{x},\tilde{u})(R_1 \mid S_1 \mid M) \overset{\overline{(\nu y)z\langle y\rangle}}{\Longrightarrow} M_1$. By construction we know that no internal actions can occur in R_1. Before observing the output along z the only possible internal actions can come from S_1 or P_1. In the former case, we can match those actions with reductions from S_2 (due to forward closure) and we thus have:

$$(\nu\tilde{x},\tilde{u})(R_1 \mid S_1 \mid M) \Longrightarrow (\nu\tilde{x},\tilde{u})(R_1 \mid S_1' \mid M) \xrightarrow{\overline{(\nu y)z\langle y\rangle}} (\nu\tilde{x},\tilde{u})(R_1 \mid S_1' \mid (\nu x)(P_1 \mid P_2 \mid P_3))$$

$$(\nu\tilde{x},\tilde{u})(R_2 \mid S_2 \mid N) \Longrightarrow (\nu\tilde{x},\tilde{u})(R_2 \mid S_2' \mid N) \xrightarrow{\overline{(\nu y)z\langle y\rangle}} (\nu\tilde{x},\tilde{u})(R_2 \mid S_2' \mid (\nu x)(P_1 \mid P_2 \mid P_3))$$

Where we may conclude by appealing to parametricity and observing that any subsequent reductions can be matched by the two processes. In the latter case we have:

$$\begin{gathered}(\nu\tilde{x},\tilde{u})(R_1 \mid S_1 \mid (\nu x)(P_1 \mid (\nu y)z\langle y\rangle.(P_2 \mid P_3))) \Longrightarrow \\ (\nu\tilde{x},\tilde{u})(R_1 \mid S_1 \mid (\nu x)(P_1' \mid (\nu y)z\langle y\rangle.(P_2 \mid P_3))) \\ \xrightarrow{\overline{(\nu y)z\langle y\rangle}} (\nu\tilde{x},\tilde{u})(R_1 \mid S_1 \mid (\nu x)(P_1' \mid (P_2 \mid P_3)))\end{gathered}$$

which we match as follows:

$$\begin{gathered}(\nu\tilde{x},\tilde{u})(R_2 \mid S_2 \mid N) \xrightarrow{\overline{(\nu y)z\langle y\rangle}} (\nu\tilde{x},\tilde{u})(R_2 \mid S_2 \mid (\nu x)(P_1 \mid P_2 \mid P_3)) \Longrightarrow \\ (\nu\tilde{x},\tilde{u})(R_2 \mid S_2 \mid (\nu x)(P_1' \mid P_2 \mid P_3))\end{gathered}$$

As before, we may conclude by appealing to parametricity and observing that any subsequent reductions can be matched by the two processes. The case where reductions occur in both S_1 and P_1 are handled by matching reductions in S_2, observing the output and then matching the reductions in P_1. □

We have thus shown that the proof transformations that arise in the cut elimination procedure can be given a reasonable concurrent justification in our framework: principal cut reductions are mapped to process (expression) reductions; proof conversions consist of sound behavioral equivalence principles, shown using our linear logical relations technique; finally, permutations of multiple instances of cut correspond immediately with structural congruence $\equiv$ (or, at the level of process expressions, with permutations in SSOS execution states).

7.4 Further Discussion

In this chapter we have developed three applications of our linear logical relations framework: We have shown that logical equivalence characterizes observational equivalence in the polymorphic (and propositional) settings – Section 7.1, using it to reason about the observable behavior of processes by analyzing their ascribed session typings in the style of reasoning using parametricity that may be performed for System F.

We developed a notion of session type isomorphism (Section 7.2), appealing to our logical equivalence, which entails a form of specification equivalence insofar as it is possible to define coercion processes between isomorphic session types that compose in both directions into a process that is observationally equivalent to the identity forwarder, mimicking the notion of type isomorphism from type theory.

Finally, we established the soundness of a class of proof transformations that arise during the cut elimination procedure in linear logic when considered in our concurrent setting (Section 7.3), by showing that the congruence induced on processes by these proof transformations is sound with respect to logical equivalence, and thus, observational equivalence.

The latter is a particularly important result since it further solidifies our claim of having a robust logical foundation for concurrent computation by being able to give an appropriate concurrent account of the entirety of cut elimination based on reduction (and structural congruence in the π-calculus setting) and on a notion of logical equivalence, extracted from our linear logical relations framework, which serves as a proof technique for observational equivalence.

We highlight the fact that while logical relations for concurrent / process calculi settings have been explored in the literature, for instance in the works of Sangiorgi [69], Caires [15], and Boudol [14], they had never before been developed for session-typed settings and thus the kind of applications developed in this chapter are typically not found in other works on session types.

Other Applications While we have focused on the three applications mentioned above, it is possible to use our linear logical relations technique to establish other properties of concurrent computation. For instance, in [59] we use a similar technique to establish a confluence property of well-typed π-calculus processes.

In the sections above we have mostly restricted our attention to the polymorphic and propositional settings. As we mention in the discussion of Chapter 6, given that the behavioral theory of higher-order process calculi (in the sense of process passing) is particularly intricate, it would be of interest to explore a suitable notion of logical equivalence in that setting.

Another interesting development would be to consider the behavioral theory of an asynchronous process expression assignment (c.f. [25]) and how it relates to observational equivalence in the synchronous setting.

Chapter 8

Conclusion

This document has been developed with the intent of supporting the thesis:

Thesis Statement: Linear logic, specifically in its intuitionistic formulation, is a suitable logical foundation for message-passing concurrent computation, providing an elegant framework in which to express and reason about a multitude of naturally occurring phenomena in such a concurrent setting.

To support this thesis we have developed a basic interpretation of propositional linear logic as a session-typed concurrent language (Chapter 2) and expanded this interpretation to both the first and second-order settings (Chapter 3), providing logically based accounts of value dependent and polymorphic session types, the former of which we have also combined with notions of proof irrelevance and affirmation to give a high-level account of proof-carrying code in a concurrent, session based setting. We have carried out this development using a novel process expression assignment to the rules of linear logic sequent calculus in tandem with a related π-calculus process assignment. A remarkable aspect of the logical interpretation is how straightforward it is to account for new language constructs and features in a uniform way. Moreover, we make use of the good properties of the underlying logic to produce a strong type safety result entailing both session fidelity and global progress (entailing deadlock-freedom of well-typed programs) in a technically simple, and arguably elegant way, scaling from the basic propositional interpretation to the first and second-order settings.

To further support the concept of intuitionistic linear logic as a logical foundation for session-based concurrency, we make use of the interpretation as the basis for a programming language combining session-based concurrency and functional programming using a contextual monad (Chapter 4), resulting in an expressive language that supports higher-order concurrency (viz. higher-order process calculi) and general recursion, with session fidelity and deadlock freedom by typing. We also study the necessary restrictions to general recursion that re-establish the connections with logic by eliminating divergence from the language.

Finally, in an effort to support the thesis that our linear logical foundation provides us with an elegant and general framework for reasoning about session-based concurrency, we develop a theory of linear logical relations (Chapter 6) for session typed π-calculus processes that can be used as a uniform proof technique to establish sophisticated properties such as termination and confluence when developed as a logical predicate, as well as provide behavioral reasoning techniques by developing a notion of logical equivalence on processes when considered in the relational setting. Moreover, we develop applications of the linear logical relations framework to reason about concurrent session-typed computation (Chapter 7), showing that logical equivalence is a sound and complete proof technique for the canonical notion of observational equivalence in a typed π-calculus setting, which we use to perform parametricity-style reasoning on polymorphic processes. We also develop the concept of a session type isomorphism, in analogy to the notion of type isomorphism in type theory, which mapped to our concurrent setting entails a form of protocol compatibility or equivalence. Our final application of logical equivalence is to establish that our concurrent interpretation of linear logic is fully compatible with the proof-theoretic notion of cut elimination, showing that all proof transformations that arise in cut elimination, when mapped to transformations of concurrent processes, are sound with respect to logical equivalence.

8.1 Related Work

We now go over some works in the literature that are related to the developments present in this document.

While Girard himself foreshadowed the connections of linear logic and concurrency, the first work to flesh out the technical details of this connection is that of Abramsky [3], relating (classical) linear logic to a λ-calculus with parallel composition. This interpretation was later refined by Belin and Scott [8] to a π-calculus with non-standard prefix commutations as structural identities, necessary to fully capture all the commuting conversions of linear logic. We may broadly characterize these as applications of the π-calculus as a convenient language for analyzing linear logic proof objects, while our aim is to develop the linear propositions-as-types paradigm as a foundation for session-based concurrency.

The explicit connection to session types was first developed by Caires and Pfenning [16], proposing a correspondence in the sense of Curry-Howard but not identifying all proof conversions of linear logic as process reductions (and thus not requiring the prefix commutations of the work of Belin and Scott). This correspondence was developed further in [16, 17], accounting for value dependent types and parametric polymorphism in the sense presented in this document; in [76], developing concurrent evaluation strategies for λ-calculus as they arise from the mapping of the translation of linear to intuitionistic logic to the process setting; and also in [25], accounting for asynchronous communication and also a slightly different, more logically faithful term assignment. It is the synchronous version of this assignment that serves as a basis for the development in this document. The work of Wadler [81] also develops a correspondence of linear logic and session types, in the classical setting, which results in a language and associated formalisms quite different from our own.

Our exploration of the concept of value-dependent session types is related to the work of [12], which encodes logical assertions on exchanged data into the session type itself. Our development provides a new account of dependent session types [13] that is completely grounded in logic, and is free from special-purpose technical machinery that is usually required in this setting. From the point of view of dependently typed languages with concurrency, perhaps the most significantly related is the work on F star, a distributed functional programming language [73]. F star does not directly employ session types, but they can be encoded in the language. In the non-dependently typed setting, Wadler's GV [81] (based on a language by Gay and Vasconcelos [32]) is quite close to our integration of functional and concurrent programming. However, GV is itself linear since it does not separate the functional and the concurrent components of the language through a monad as we do.

Polymorphism for process calculi has been studied extensively in the literature. However, our work seems to be the first developing termination and relational parametricity results for polymorphic session types, by relying solely on linear logic principles. Turner [78] investigated a form of polymorphism based only on existential quantification for a simply-typed π-calculus (roughly, the discipline in which types of names describe the objects they can carry). In processes, polymorphism is expressed as explicit type parameters in input/output prefixes: messages are composed of a type and values of that type, and so type variables are relevant at the level of messages. Sangiorgi and Pierce [63] proposed a behavioral theory for Turner's framework. Neither of these works address termination of well-typed processes nor studies relational parametricity. Building on [63], Jeffrey and Rathke [45] show that weak bisimulation is fully abstract for observational equivalence for an asynchronous polymorphic π-calculus with name equality testing. Berger et al. [10, 11] proposed a polymorphic π-calculus with both universal and existential quantification. The type system considered in [10, 11] is very different from ours, relying on a combination of linearity, so-called action types, and duality principles.

The usage of logical relations (or closely related techniques) in the context of concurrency and process calculi has been investigated by Berger, Honda, and Yoshida [10, 11, 83], Sangiorgi [69], Caires [15], and Boudol [14]. None of these works consider session types, nor are grounded in a logical interpretation as ours, and thus differ substantially from our developments. Boudol [14] relies on the classical realizability technique (together with a type and effect system) to establish termination in a higher-order imperative language. Caires [15] proposes a semantic approach to showing soundness for type systems for concurrency, by relying on a spatial logic interpretation of behavioral types. The works by Yoshida, Berger, Honda [83] and by Sangiorgi [69], aim at identifying terminating fragments of the π-calculus by using types, relying on arguments based on logical relations. While the connection of logical relations and typed observational equivalences is particular to our work, we can identify in the literature the behavioral equivalences for binary session-typed processes of [48]. The work studies the behavioral theory of a π-calculus with *asynchronous, event-based* binary session communication, capturing the distinction between order-preserving communications (those inside already established connections) and non-order-preserving communications (those outside such connections). The behavioral theory in [48] accounts for principles for prefix commutation that are similar to those induced by our proof conversions.

8.2 Future Work

In this section we summarize the most potentially promising topics for further research, exploiting the logical foundation and language framework set out in this dissertation. We are by no means extensive in this discussion, but rather point out the major research topics.

Concurrent Type Theory Perhaps the most important avenue for future investigation on a more theoretical setting is the development of a general type theory for concurrent session-based computation, generalizing our value-dependent session types to a fully dependent setting and potentially integrating our contextual monadic construction as a way of connecting the two language layers. As is usually the case in dependent type theories, the crux of this development lies in a suitable notion of definitional equality of types, terms and process expressions, which in principle should be decidable yet expressive enough to enable reasoning on language terms in the style of functional type theories. Another crucial point is the interplay of definitional equality with coinductive definitions (and proofs by coinduction), which play a predominant role in reasoning about concurrent computation in general. It would be also interesting to explore more permissive techniques for ensuring productivity of coinductive definitions in our setting.

Linear Logical Relations for Languages without Termination As we have briefly discussed in Section 6.3, our linear logical relations framework can only be applied to languages without divergence. In order to provide general reasoning techniques for session-typed languages with general recursion we would likely need to develop step-indexed formulations of our logical relations formalism. At this point, it is not obvious how to adapt the step-indexed formalisms of functional and imperative programming languages to a setting with concurrency, nor what kind of equivalences and properties we would be able to effectively characterize with such an approach.

Towards a Realistic Language Implementation From a more pragmatical point of view, our concurrent session-based programming language provides us with many potentially interesting avenues of research. The language focuses on the integration of synchronous concurrent processes that adhere to a session typing discipline with a λ-calculus, but does not provide us with ways of expressing (logical or physical) distribution of code. Conceptually, it is conceivable that an account of distributed code can be given by considering ideas from hybrid or modal logic, but at a practical level we would need to develop a suite of monitoring and dynamic verification techniques to ensure good properties of distributed code (which is, in general, not available for typechecking purposes). This naturally leads to the development of blame assignment mechanisms for breakages of the typing discipline by misbehaving (or malicious) components. A combination of these techniques with the (value) dependent session types of Section 3.1 would be particularly interesting as a highly expressive distributed language.

Moreover, when moving towards realistic implementations of our language we would need to deal with the fact that real-world communication is asynchronous by nature. This would require a conceptual extension of the asynchronous interpretation of intuitionistic linear logic of [25] to our language setting, for which the behavioral theory and associated reasoning techniques have yet to be developed. Furthermore, it is likely that we would also need to provide with a higher degree of parallelism, as argued in Section 5.1, for the sake of efficiency, as well as relaxed linearity constraints for usability by a general audience (for instance, considering affine typing disciplines).

BIBLIOGRAPHY

[1] M. Abadi and A. D. Gordon. “A calculus for cryptographic protocols: The spi calculus”. In: *Computer and Communications Security*. ACM Press, 1997, pp. 36–47.

[2] A. Abel. “Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types”. In: *Proceedings of FICS’2012*. 2012, pp. 1–11.

[3] S. Abramsky. “Computational Interpretations of Linear Logic”. In: *Theoret. Comput. Sci.* 111.1–2 (1993), pp. 3–57.

[4] A. J. Ahmed. “Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types”. In: *European Symposium on Programming 2006*. 2006, pp. 69–83.

[5] T. Altenkirch, J. Chapman, and T. Uustalu. “Monads Need Not Be Endofunctors”. In: *FOSSACS*. 2010, pp. 297–311.

[6] S. Awodey and A. Bauer. “Propositions as [Types]”. In: *J. Log. Comput.* 14.4 (2004), pp. 447–471.

[7] A. Barber. *Dual Intuitionistic Linear Logic*. Tech. rep. LFCS-96-347. Univ. of Edinburgh, 1996.

[8] G. Bellin and P. Scott. “On the π-Calculus and Linear Logic”. In: *Theoret. Comput. Sci.* 135.1 (1994), pp. 11–65.

[9] P. N. Benton. “A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Preliminary Report)”. In: *Comp. Sci. Log.* Springer-Verlag, 1994, pp. 121–135.

[10] M. Berger, K. Honda, and N. Yoshida. “Genericity and the pi-Calculus”. In: *Proc. of FoSSaCS*. Vol. 2620. LNCS. Springer, 2003, pp. 103–119.

[11] M. Berger, K. Honda, and N. Yoshida. “Genericity and the pi-calculus”. In: *Acta Inf.* 42.2-3 (2005), pp. 83–141.

[12] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. “A Theory of Design-by-Contract for Distributed Multiparty Interactions”. In: *21st Int. Conf. Concur. Theory*. LNCS 6269, 2010, pp. 162–176.

[13] E. Bonelli, A. Compagnoni, and E. L. Gunter. “Correspondence Assertions for Process Synchronization in Concurrent Communications”. In: *J. of Func. Prog.* 15.2 (2005), pp. 219–247.

[14] G. Boudol. “Typing termination in a higher-order concurrent imperative language”. In: *Inf. Comput.* 208.6 (2010), pp. 716–736.

[15] L. Caires. “Logical Semantics of Types for Concurrency”. In: *International Conference on Algebra and Coalgebra in Computer Science*. CALCO’07. Springer LNCS 4624, 2007, pp. 16–35.

[16] L. Caires and F. Pfenning. “Session Types as Intuitionistic Linear Propositions”. In: *CONCUR’10*. 2010, pp. 222–236.

[17] L. Caires, J. A. Pérez, F. Pfenning, and B. Toninho. “Behavioral Polymorphism and Parametricity in Session-Based Communication”. In: *ESOP*. 2013, pp. 330–349.

[18] L. Caires, F. Pfenning, and B. Toninho. “Linear logic propositions as session types”. In: *Math. Struct. in Comp. Sci.* (2013). Accepted for publication.

[19] L. Cardelli and A. D. Gordon. “Mobile Ambients”. In: *POPL’98*. ACM Press, 1998.

[20] I. Cervesato and A. Scedrov. “Relating State-Based and Process-Based Concurrency through Linear Logic”. In: *Information and Computation* 207.10 (2009), pp. 1044–1077.

[21] B.-Y. E. Chang, K. Chaudhuri, and F. Pfenning. *A Judgmental Analysis of Linear Logic*. Tech. rep. CMU-CS-03-131R. Carnegie Mellon Univ., 2003.

[22] T. Coquand and G. Huet. “The calculus of constructions”. In: *Inf. & Comput.* 76 (2-3 1988), pp. 95–120.

[23] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[24] R. Davies and F. Pfenning. “A modal analysis of staged computation”. In: *J. ACM* 48.3 (2001), pp. 555–604.

[25] H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. “Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication”. In: *Computer Science Logic*. 2012.

[26] M. Dezani-Ciancaglini, U. de’ Liguoro, and N. Yoshida. “On Progress for Structured Communications”. In: *TGC’07*. 2008, pp. 257–275.

[27] R. Di Cosmo. “A short survey of isomorphisms of types”. In: *Mathematical Structures in Computer Science* 15.5 (2005), pp. 825–838.

[28] C. Faggian and M. Piccolo. “Ludics is a Model for the Finitary Linear Pi-Calculus,” in: *Typed Lambda Calculi and Applications, TLCA 2007*. LNCS. Springer-Verlag, 2007, pp. 148–162.

[29] M. Fairtlough and M. Mendler. “Propositional Lax Logic”. In: *Information and Computation* 137.1 (Aug. 1997), pp. 1–33.

[30] D. Garg, L. Bauer, K. Bowers, F. Pfenning, and M. Reiter. “A Linear Logic of Affirmation and Knowledge”. In: *11th Eur. Symp. Res. Comput. Secur.* LNCS 4189, 2006, pp. 297–312.

[31] S. Gay and M. Hole. “Subtyping for Session Types in the Pi Calculus”. In: *Acta Informatica* 42.2-3 (2005), pp. 191–225.

[32] S. Gay and V. T. Vasconcelos. “Linear Type Theory for Asynchronous Session Types”. In: *J. Funct. Programming* 20.1 (2010), pp. 19–50.

[33] S. J. Gay. “Bounded polymorphism in session types”. In: *Math. Struc. in Comp. Sci.* 18.5 (2008), pp. 895–930.

[34] E. Gimenez. “Structural Recursive Definitions in Type Theory”. In: *Automata, Languages and Programming, 25th International Colloquium, ICALP’98*. Springer-Verlag, 1998, pp. 13–17.

[35] J. Y. Girard. “Une extension de l’interprétation de Gödel à l’analyse, et son application à l’élimination de coupures dans l’analyse et la théorie des types”. In: *Proc. of the 2nd Scandinavian Logic Symposium*. North-Holland Publishing Co., 1971, pp. 63–92.

[36] J.-Y. Girard. “Linear Logic”. In: *Theoret. Comput. Sci.* 50.1 (1987), pp. 1–102.

[37] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, 1989. ISBN: 0521371813.

[38] M. Giunti and V. T. Vasconcelos. “A Linear Account of Session Types in the π-Calculus”. In: *21st Int. Conf. Concur. Theory*. LNCS 6269, 2010, pp. 432–446.

[39] B. Grégoire and J. L. Sacchini. “On Strong Normalization of the Calculus of Constructions with Type-Based Termination”. In: *LPAR*. 2010, pp. 333–347.

[40] R. Harper, F. Honsell, and G. Plotkin. “A framework for defining logics”. In: *J. ACM* 40 (1 1993), pp. 143–184.

[41] K. Honda. “Types for Dyadic Interaction”. In: *CONCUR’93*. 1993, pp. 509–523.

[42] K. Honda and O. Laurent. “An Exact Correspondence between a Typed π-Calculus and Polarised Proof-Nets”. In: *Theoret. Comput. Sci.* 411.22–24 (2010), pp. 2223–2238.

[43] K. Honda, V. T. Vasconcelos, and M. Kubo. “Language Primitives and Type Discipline for Structured Communication-Based Programming”. In: *ESOP*. 1998, pp. 122–138.

[44] W. A. Howard. “The formulae-as-types notion of construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard. Academic Press, 1980, pp. 479–490.

[45] A. Jeffrey and J. Rathke. “Full abstraction for polymorphic pi-calculus”. In: *Theor. Comput. Sci.* 390.2-3 (2008), pp. 171–196.

[46] N. Kobayashi. “A Partially Deadlock-Free Typed Process Calculus”. In: *ACM Tr. Progr. Lang. Sys.* 20.2 (1998), pp. 436–482.

[47] N. Kobayashi, B. C. Pierce, and D. N. Turner. “Linearity and the Pi-Calculus”. In: *23rd Symposium on Principles of Programming Languages*. POPL’96. ACM, 1996, pp. 358–371.

[48] D. Kouzapas, N. Yoshida, R. Hu, and K. Honda. “On Asynchronous Session Semantics”. In: *Proc. of FMOODS-FORTE’2011*. Vol. 6722. LNCS. Springer, 2011, pp. 228–243.

[49] P. Martin-Löf. “Constructive Mathematics and Computer Programming”. In: *Logic, Methodology and Philosophy of Science VI*. North-Holland, 1980, pp. 153–175.

[50] R. Milner. *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[51] R. Milner. “Functions as Processes”. In: *Math. Struct. in Comp. Sci.* 2.2 (1992), pp. 119–141.

[52] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[53] D. Mostrous and N. Yoshida. “Two session typing systems for higher-order mobile processes”. In: *TLCA’07*. 2007, pp. 321–335.

[54] T. Murphy, K. Crary, R. Harper, and F. Pfenning. “A Symmetric Modal Lambda Calculus for Distributed Computing”. In: *LICS*. 2004, pp. 286–295.

[55] U. Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. SE-412 96 Göteborg, Sweden: Chalmers University of Technology, 2007.

[56] P. W. O’Hearn. “Resources, concurrency, and local reasoning”. In: *Theoretical Computer Science* 375.1–3 (2007), pp. 271 –307. ISSN: 0304-3975.

[57] J. Parrow and B. Victor. “The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes”. In: *Logic in Computer Science*. LICS ’98. IEEE Computer Society, 1998, pp. 176–.

[58] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. “Linear Logical Relations for Session-Based Concurrency”. In: *ESOP’12*. 2012, pp. 539–558.

[59] J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. “Linear logical relations and observational equivalences for session-based concurrency”. In: *Information and Computation* (2014). to appear.

[60] S. L. Peyton Jones and P. Wadler. “Imperative functional programming”. In: *Principles of Prog. Lang.* POPL’93. 1993, pp. 71–84.

[61] F. Pfenning. “Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory”. In: *16th Symposium on Logic in Computer Science*. LICS’01. IEEE Computer Society, 2001, pp. 221–230.

[62] F. Pfenning and R. J. Simmons. “Substructural Operational Semantics as Ordered Logic Programming”. In: *Logic in Comp. Sci.* 2009, pp. 101–110.

[63] B. C. Pierce and D. Sangiorgi. “Behavioral equivalence in the polymorphic pi-calculus”. In: *J. ACM* 47.3 (2000), pp. 531–584.

[64] J. C. Reynolds. “Towards a theory of type structure”. In: *Programming Symposium, Proceedings Colloque sur la Programmation*. London, UK, UK: Springer-Verlag, 1974, pp. 408–423.

[65] J. C. Reynolds. “Types, Abstraction and Parametric Polymorphism”. In: *Information Processing 83. Proceedings of the IFIP 9th World Computer Congress*. Ed. by R. E. A. Mason. Elsevier, 1983, pp. 513–523.

[66] D. Sangiorgi. “Bisimulation in higher-order calculi”. In: *Proc. IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET’94)*. North-Holland, 1994, pp. 207–224.

[67] D. Sangiorgi. “Pi-Calculus, Internal Mobility, and Agent Passing Calculi”. In: *Theoretical Computer Science* 167.1&2 (1996), pp. 235–274.

[68] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[69] D. Sangiorgi. “Termination of processes”. In: *Mathematical Structures in Computer Science* 16.1 (2006), pp. 1–39.

[70] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[71] D. Sangiorgi and J. Rutten, eds. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2012.

[72] D. Sangiorgi, N. Kobayashi, and E. Sumii. “Environmental Bisimulations for Higher-Order Languages”. In: *Proc. 22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE Computer Society, 2007, pp. 293–302.

[73] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. “Secure distributed programming with value-dependent types”. In: *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ICFP ’11. ACM, 2011, pp. 266–278.

[74] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. 2013. URL: `http://coq.inria.fr`.

[75] B. Toninho, L. Caires, and F. Pfenning. “Dependent session types via intuitionistic linear type theory”. In: *PPDP’11*. 2011, pp. 161–172.

[76] B. Toninho, L. Caires, and F. Pfenning. “Functions as Session-Typed Processes”. In: *15th Int. Conf. Foundations of Software Science and Computation Structures*. LNCS 7213, 2012.

[77] B. Toninho, L. Caires, and F. Pfenning. “Higher-Order Processes, Functions, and Sessions: A Monadic Integration”. In: *ESOP’13*. 2013, pp. 350–369.

[78] D. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. Tech. rep. ECS-LFCS-96-345, Univ. of Edinburgh, 1996.

[79] P. Wadler. “Theorems for free!” In: *Functional Programming Languages and Computer Architecture*. ACM Press, 1989, pp. 347–359.

[80] P. Wadler. “The Marriage of Effects and Monads”. In: *International Conference on Functional Programming*. ICFP ’98. ACM, 1998, pp. 63–74.

[81] P. Wadler. “Propositions as Sessions”. In: *ICFP’12*. 2012, pp. 273–286.

[82] P. Wadler. “Propositions as Types”. In: (2014). Unpublished Survey.

[83] N. Yoshida, M. Berger, and K. Honda. “Strong normalisation in the pi -calculus”. In: *Inf. Comput.* 191.2 (2004), pp. 145–202.

[84] N. Yoshida, K. Honda, and M. Berger. “Linearity and Bisimulation”. In: *J. Logic and Algebraic Programming* 72.2 (2007), pp. 207–238.

[85] N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. “Parameterised Multiparty Session Types”. In: *Proc. of FoSSaCS*. Vol. 6014. Springer, 2010, pp. 128–145.

APPENDIX A

PROOFS

A.1 Inversion Lemma

Lemma 40. *Let* $\Gamma; \Delta \Rightarrow P :: x{:}C$.

1. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = \mathbf{1}$ *then* $\alpha = \overline{x\langle\rangle}$
2. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : \mathbf{1} \in \Delta$ *then* $\alpha = y()$.
3. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \otimes B$ *then* $\alpha = \overline{(\nu y)x\langle y\rangle}$.
4. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \otimes B \in \Delta$ *then* $\alpha = y(z)$.
5. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \multimap B$ *then* $\alpha = x(y)$.
6. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \multimap B \in \Delta$ *then* $\alpha = \overline{(\nu z)y\langle z\rangle}$.
7. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \mathbin{\&} B$ *then* $\alpha = x.\mathsf{inl}$ *or* $\alpha = x.\mathsf{inr}$.
8. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \mathbin{\&} B \in \Delta$ *then* $\alpha = \overline{y.\mathsf{inl}}$ *or* $\alpha = \overline{y.\mathsf{inr}}$.
9. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = A \oplus B$ *then* $\alpha = \overline{x.\mathsf{inl}}$ *or* $\alpha = \overline{x.\mathsf{inr}}$.
10. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : A \oplus B \in \Delta$ *then* $\alpha = y.\mathsf{inl}$ *or* $\alpha = y.\mathsf{inr}$.
11. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = x$ *and* $C = {!}A$ *then* $\alpha = \overline{(\boldsymbol{\nu} u)x\langle u\rangle}$.
12. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = y$ *and* $y : {!}A \in \Delta$ *then* $\alpha = y(u)$.
13. *If* $P \xrightarrow{\alpha} Q$ *and* $s(\alpha) = u$ *and* $u : A \in \Gamma$ *then* $\alpha = \overline{(\boldsymbol{\nu} z)u\langle z\rangle}$.

Proof. By induction on typing.

1. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = x$ and $C = \mathbf{1}$ then $\alpha = \overline{x\langle\rangle}$

Case: $\mathbf{1}\mathsf{R}$

$\alpha = \overline{x\langle\rangle}$ by the l.t.s

Case: cut with $P \equiv (\boldsymbol{\nu} w)(P_1 \mid P_2)$

$\alpha = \overline{x\langle\rangle}$ by i.h. and $x \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\boldsymbol{\nu} u)(P_1 \mid P_2)$

$\alpha = \overline{x\langle\rangle}$ by i.h.

Case: All other rules do not have $s(\alpha) = x$ and $C = \mathbf{1}$

2. If $P \overset{\alpha}{\to} Q$ and $s(\alpha) = y$ and $y : \mathbf{1} \in \Delta$ then $\alpha = y()$.

Case: $\mathbf{1}\mathsf{L}$

$\alpha = y()$ by the l.t.s

Case: cut with $P \equiv (\boldsymbol{\nu} w)(P_1 \mid P_2)$

Subcase: $y \in \Delta_1$

$\alpha = y()$ by i.h. on $y \notin fn(P_2)$

Subcase: $y \in \Delta_2$

$\alpha = y()$ by i.h. and $y \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\boldsymbol{\nu} u)(P_1 \mid P_2)$

$\alpha = y()$ by i.h.

3. If $P \overset{\alpha}{\to} Q$ and $s(\alpha) = x$ and $C = A \otimes B$ then $\alpha = \overline{(\nu y)x\langle y\rangle}$

Case: $\otimes\mathsf{R}$

$\alpha = \overline{(\nu y)x\langle y\rangle}$ by the l.t.s

Case: cut with $P \equiv (\boldsymbol{\nu} w)(P_1 \mid P_2)$

$\alpha = \overline{(\nu y)x\langle y\rangle}$ by i.h. $x \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\boldsymbol{\nu} u)(P_1 \mid P_2)$

$\alpha = \overline{(\nu y)x\langle y\rangle}$ by i.h.

Case: All other rules do not have $s(\alpha) = x$ and $C = A \otimes B$

4. If $P \overset{\alpha}{\to} Q$ and $s(\alpha) = y$ and $y : A \otimes B \in \Delta$ then $\alpha = y(z)$

Case: $\otimes\mathsf{L}$

$\alpha = y(z)$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

Subcase: $y \in \Delta_1$

$\alpha = y(z)$ by i.h. and $y \notin fn(P_2)$

Subcase: $y \in \Delta_2$

$\alpha = y(z)$ by i.h. and $y \notin fn(P_1)$

Case: $\mathrm{cut}^!$ with $P \equiv (\boldsymbol{\nu} w)(P_1 \mid P_2)$

$\alpha = y(z)$ by i.h.

5. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = x$ and $C = A \multimap B$ then $\alpha = x(y)$

Case: $\multimap\mathsf{R}$

$\alpha = x(y)$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

$\alpha = x(y)$ by i.h. and $x \notin fn(P_1)$

Case: $\mathrm{cut}^!$ with $P \equiv (\boldsymbol{\nu} u)(P_1 \mid P_2)$

$\alpha = x(y)$ by i.h.

Case: All other rules do not have $s(\alpha) = x$ and $C = A \multimap B$

6. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = y$ and $y : A \multimap B \in \Delta$ then $\alpha = \overline{(\nu z)y\langle z\rangle}$

Case: $\multimap\mathsf{L}$

$\alpha = \overline{(\nu z)y\langle z\rangle}$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

Subcase: $y \in \Delta_1$

$\alpha = \overline{(\nu z)y\langle z\rangle}$ by i.h. and $y \notin fn(P_2)$

Subcase: $y \in \Delta_2$

$\alpha = \overline{(\nu z)y\langle z\rangle}$ by i.h. and $y \notin fn(P_1)$

Case: $\mathrm{cut}^!$ with $P \equiv (\boldsymbol{\nu} w)(P_1 \mid P_2)$

$\alpha = \overline{(\nu z)y\langle z\rangle}$ by i.h. on D_2

7. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = x$ and $C = A \,\&\, B$ then $\alpha = x.\mathsf{inl}$ or $\alpha = x.\mathsf{inr}$

Case: $\&\mathsf{R}$

$\alpha = x.\mathsf{inl}$ or $\alpha = x.\mathsf{inr}$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

$\alpha = x.\mathsf{inl}$ or $\alpha = x.\mathsf{inr}$ by i.h. and $x \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\nu u)(P_1 \mid P_2)$

$\alpha = x.\mathsf{inl}$ or $\alpha = x.\mathsf{inr}$ by i.h.

Case: All other rules do not have $s(\alpha) = x$ and $C = A \,\&\, B$

8. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = y$ and $y : A \,\&\, B \in \Delta$ then $\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$

Case: $\&\mathsf{L}_1$

$\alpha = \overline{y.\mathsf{inl}}$ by the l.t.s

Case: $\&\mathsf{L}_2$

$\alpha = \overline{y.\mathsf{inr}}$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

Subcase: $y \in \Delta_1$
$\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$ by i.h. and $y \notin fn(P_2)$
Subcase: $y \in \Delta_2$
$\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$ by i.h. and $y \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\nu u)(P_1 \mid P_2)$

$\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$ by i.h.

9. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = x$ and $C = A \oplus B$ then $\alpha = \overline{x.\mathsf{inl}}$ or $\alpha = \overline{x.\mathsf{inr}}$

Case: $\oplus\mathsf{R}_1$

$\alpha = \overline{x.\mathsf{inl}}$ by the l.t.s

Case: $\oplus\mathsf{R}_2$

$\alpha = \overline{x.\mathsf{inr}}$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

$\alpha = \overline{x.\mathsf{inl}}$ or $\alpha = \overline{x.\mathsf{inr}}$ by i.h. and $x \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\nu u)(P_1 \mid P_2)$

$\alpha = \overline{x.\mathsf{inl}}$ or $\alpha = \overline{x.\mathsf{inr}}$ by i.h.

Case: All other rules do not have $s(\alpha) = x$ and $C = A \oplus B$

10. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = y$ and $y : A \oplus B \in \Delta$ then $\alpha = y.\mathsf{inl}$ or $\alpha = y.\mathsf{inr}$

Case: $\oplus\mathsf{L}$

$\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

Subcase: $y \in \Delta_1$
$\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$ by i.h. and $y \notin fn(P_2)$
Subcase: $y \in \Delta_2$
$\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$ by i.h. and $y \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\nu u)(P_1 \mid P_2)$

$\alpha = \overline{y.\mathsf{inl}}$ or $\alpha = \overline{y.\mathsf{inr}}$ by i.h.

11. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = x$ and $C = !A$ then $\alpha = \overline{(\nu u)x\langle u\rangle}$

Case: $!\mathsf{R}$

$\alpha = \overline{(\nu u)x\langle u\rangle}$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

$\alpha = \overline{(\nu u)x\langle u\rangle}$ by i.h. and $x \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\nu v)(P_1 \mid P_2)$

$\alpha = \overline{(\nu u)x\langle u\rangle}$ by i.h.

Case: All other rules do not have $s(\alpha) = x$ and $C = !A$

12. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = y$ and $y : !A \in \Delta$ then $\alpha = y(u)$.

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

Subcase: $y \in \Delta_1$
$\alpha = y(u)$ by i.h. and $y \notin fn(P_2)$
Subcase: $y \in \Delta_2$
$\alpha = y(u)$ by i.h. and $y \notin fn(P_1)$

Case: $\mathsf{cut}^!$ with $P \equiv (\nu w)(P_1 \mid P_2)$

$\alpha = y(u)$ by i.h.

13. If $P \xrightarrow{\alpha} Q$ and $s(\alpha) = u$ and $u : A \in \Gamma$ then $\alpha = \overline{(\nu z)u\langle z\rangle}$.

Case: copy

$\alpha = \overline{(\nu z)u\langle z\rangle}$ by the l.t.s

Case: cut with $P \equiv (\nu w)(P_1 \mid P_2)$

$\alpha = \overline{(\nu z)u\langle z\rangle}$ by i.h.

Case: cut$^!$ with $P \equiv (\nu v)(P_1 \mid P_2)$

$\alpha = \overline{(\nu z)u\langle z\rangle}$ by i.h.

□

A.2 Reduction Lemmas - Value Dependent Session Types

Lemma 41 (Reduction Lemma - $\forall$). *Assume*

(a) $\Psi;\Gamma;\Delta_1 \Rightarrow P :: x{:}\forall y{:}\tau.A$ *with* $P \xrightarrow{x(V)} P'$

(b) $\Psi;\Gamma;\Delta_2, x{:}\forall y{:}\tau.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{x\langle V\rangle}} Q'$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. By simultaneous induction on the typing of P and Q. The possible cases for the typing of P are $\forall$R, cut and cut$^!$. The possible cases for Q are $\forall$L, cut and cut$^!$.

Case: P from $\forall$R and Q from $\forall$L.

$P \equiv x(y).P_1$ and $E \equiv x\langle V\rangle.Q_1$	
$(\nu x)(x(y).P_1 \mid x\langle V\rangle.Q_1) \rightarrow (\nu x)(P_1\{N/y\} \mid Q_1)$	by principal cut reduction
$\Psi;\Gamma;\Delta_1 \Rightarrow P_1\{N/y\} :: A\{N/y\}$	by Lemma 12
$\Psi;\Gamma;\Delta_2, x{:}A\{N/y\} \Rightarrow Q_1 :: z{:}C$	by inversion
$\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$	by cut.

Other cases are identical to the proof of Lemma 1.

□

Lemma 42 (Reduction Lemma - $\exists$). *Assume*

(a) $\Psi;\Gamma;\Delta_1 \Rightarrow P :: x{:}\exists y{:}\tau.A$ *with* $P \xrightarrow{\overline{x\langle V\rangle}} P'$ *and* $\Psi \vdash V{:}\tau$

(b) $\Psi;\Gamma;\Delta_2, x{:}\exists y{:}\tau.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x(V)} Q'$ *and* $\Psi \vdash V{:}\tau$

Then:

(c) $(\nu x)(P \mid Q) \rightarrow (\nu x)(P' \mid Q') \equiv R$ *such that* $\Psi;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. By simultaneous induction on the typing of P and Q. The possible cases for the typing of P are $\exists$R, cut and cut$^!$. The possible cases for Q are $\exists$L, cut and cut$^!$.

Case: P from $\exists$R and Q from $\exists$L.

$P \equiv x\langle V\rangle.P_1$ and $Q \equiv x(y).Q_1$
$(\nu x)(x\langle V\rangle.P_1 \mid x(y).Q_1) \to (\nu x)(P_1 \mid Q_1\{V/y\})$ by principal cut reduction
$\Psi;\Gamma;\Delta_2, x{:}A\{N/y\} \Rightarrow Q_1\{N/y\} :: z{:}C$ by Lemma 12
$\Psi;\Gamma;\Delta_1 \Rightarrow P_1 :: x{:}A\{V/y\}$ by inversion
$\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$ by cut.

Other cases are identical to the proof of Lemma 1.

□

A.3 Reduction Lemmas - Polymorphic Session Types

Lemma 43 (Reduction Lemma - $\forall$2). *Assume*

(a) $\Omega;\Gamma;\Delta_1 \Rightarrow P :: x{:}\forall X.A$ *with* $P \xrightarrow{x(B)} P'$ *and* $\Omega \vdash B$ type

(b) $\Omega;\Gamma;\Delta_2, x{:}\forall X.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{\overline{x\langle B\rangle}} Q'$ *and* $\Omega \vdash B$ type

Then:

(c) $(\nu x)(P \mid Q) \to (\nu x)(P' \mid Q') \equiv R$ *such that* $\Omega;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. By simultaneous induction on typing for P and Q. The possible cases for P are $\forall$R2, cut and cut$^!$. The possible cases for Q are $\forall$L2, cut and cut$^!$.

Case: P from $\forall$R2 and Q from $\forall$L2

$P \equiv x(X).P_1$ and $Q \equiv x\langle B\rangle.Q_1$
$(\nu x)(x(X).P_1 \mid x\langle B\rangle.Q_1) \to (\nu x)(P_1\{B/X\} \mid Q_1)$ by principal cut reduction
$\Omega;\Gamma;\Delta_1 \Rightarrow P_1\{B/X\} :: x{:}A\{B/X\}$ by substitution
$\Omega;\Gamma;\Delta_2, x{:}A\{B/X\} \Rightarrow Q_1 :: z{:}C$ by inversion
$\Omega;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$ by cut.

Other cases are identical to the proof of Lemma 1.

□

Lemma 44 (Reduction Lemma - $\exists$2). *Assume*

(a) $\Omega;\Gamma;\Delta_1 \Rightarrow P :: x{:}\exists X.A$ *with* $P \xrightarrow{\overline{x\langle B\rangle}} P'$ *and* $\Omega \vdash B$ type

(b) $\Omega;\Gamma;\Delta_2, x{:}\exists X.A \Rightarrow Q :: z{:}C$ *with* $Q \xrightarrow{x(B)} Q'$ *and* $\Omega \vdash B$ type

Then:

(c) $(\nu x)(P \mid Q) \to (\nu x)(P' \mid Q') \equiv R$ *such that* $\Omega;\Gamma;\Delta_1,\Delta_2 \Rightarrow R :: z{:}C$

Proof. By simultaneous induction on typing for P and Q. The possible cases for P are $\exists$R2, cut and cut$^!$. The possible cases for Q are $\exists$L2, cut and cut$^!$.

Case: P from $\exists$R2 and Q from $\exists$L2

$P \equiv x\langle B\rangle.P_1$ and $Q \equiv x(X).Q_1$

$(\nu x)(x\langle B\rangle.P_1 \mid x(X).Q_1) \rightarrow (\nu x)(P_1 \mid Q_1\{B/X\})$ by principal cut reduction

$\Omega;\Gamma;\Delta_2, x{:}A\{B/X\} \Rightarrow Q_1\{B/X\} :: z{:}C$ by substitution

$\Omega;\Gamma;\Delta_1, \Rightarrow P_1 :: x{:}A\{B/X\}$ by inversion

$\Omega;\Gamma;\Delta_1, \Delta_2 \Rightarrow R :: z{:}C$ by cut.

Other cases are identical to the proof of Lemma 1.

□

A.4 Logical Predicate for Polymorphic Session Types

A.4.1 Proof of Theorem 28

Proof. We proceed by induction on the structure of the type A.

Case: $A = X$

$P \in \eta(X)(z)$ by Def.

Result follows by Def. of $\eta(X)(z)$.

Case: $A = A_1 \multimap A_2$

T.S: If for all P_i such that $P \Longrightarrow P_i$ we have $P_i \in \mathcal{T}^\omega_\eta[z{:}A_1 \multimap A_2]$ then

$P \in \mathcal{T}^\omega_\eta[z{:}A_1 \multimap A_2]$

S.T.S: $\forall P''y.(P \overset{z(y)}{\Longrightarrow} P'') \Rightarrow \forall Q \in \mathcal{T}^\omega_\eta[y : A].(\nu y)(P'' \mid Q) \in \mathcal{T}^\omega_\eta[z{:}B]$

$\forall P''y.(P_i \overset{z(y)}{\Longrightarrow} P'') \Rightarrow \forall Q \in \mathcal{T}^\omega_\eta[y : A].(\nu y)(P'' \mid Q) \in \mathcal{T}^\omega_\eta[z{:}B]$ (a) by Def. 11

$P \Longrightarrow P_i \overset{z(y)}{\Longrightarrow} P''$ by assumption and Def.

$P \overset{z(y)}{\Longrightarrow} P''$ (b) by the reasoning above

$\forall Q \in \mathcal{T}^\omega_\eta[y : A].(\nu y)(P'' \mid Q) \in \mathcal{T}^\omega_\eta[z{:}B]$ (c) by (a) and (b)

$P \in \mathcal{T}^\omega_\eta[z{:}A_1 \multimap A_2]$ by (c), satisfying (3)

T.S: If $P \in \mathcal{T}^\omega_\eta[z{:}A_1 \multimap A_2]$ then $P \Downarrow P \overset{z(y)}{\Longrightarrow} P'$ (a) by assumption, for some P'

$\forall Q \in \mathcal{T}^\omega_\eta[y : A].(\nu y)(P' \mid Q) \in \mathcal{T}^\omega_\eta[z{:}B]$ by (a) and $P \in \mathcal{T}^\omega_\eta[z{:}A_1 \multimap A_2]$

$Q \Downarrow$ by i.h.

$(\nu y)(P' \mid Q) \Downarrow$ (b) by i.h.

$P' \Downarrow$ (c) by (b)

$P \Downarrow$ by (c) and (a), satisfying (2)

T.S.: If $P \in \mathcal{T}^\omega_\eta[z{:}A_1 \multimap A_2]$ and $P \Longrightarrow P'$ then $P' \in \mathcal{T}^\omega_\eta[z{:}A_1 \multimap A_2]$

Follows directly from definition of weak transition.

Case: $A = A_1 \otimes A_2$

Similar to above.

Case: $A = A_1 \,\&\, A_2$

Similar to above.

Case: $A = A_1 \oplus A_2$

Similar to above.

Case: $A = !A_1$

Similar to above.

Case: $A = \forall X.A_1$

T.S: If for all P_i such that $P \Longrightarrow P_i$ we have $P_i \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A_1]$ then
$P \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A_1]$

S.T.S: $\forall B, P', \mathsf{R}[w : B].\ (B\ \mathsf{type} \wedge P \overset{z(B)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto \mathsf{R}[w:B]]}[z{:}A])$

$\forall B, P', \mathsf{R}[w : B].\ (B\ \mathsf{type} \wedge P_i \overset{z(B)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto \mathsf{R}[w:B]]}[z{:}A])$

(a) by Def. of $P_i \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A_1]$

$P \Longrightarrow P_i \overset{z(B)}{\Longrightarrow} P'$ (b) by assumption and (a)

$P \overset{z(B)}{\Longrightarrow} P'$ (c)

$P' \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto \mathsf{R}[w:B]]}[z{:}A]$ (d) by (a) and (b)

$P \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A_1]$ by (c) and (d), satisfying (3)

T.S: If $P \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A_1]$ then $P \Downarrow$

$\forall B, P', \mathsf{R}[w : B].\ (B\ \mathsf{type} \wedge P \overset{z(B)}{\Longrightarrow} P') \Rightarrow P' \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto \mathsf{R}[w:B]]}[z{:}A])$ (a) by Def. 11

$P \overset{z(B)}{\Longrightarrow} P'$ (b) for some P', B

$P' \in \mathcal{T}^{\omega[X\mapsto B]}_{\eta[X\mapsto \mathsf{R}[w:B]]}[z{:}A]$ by (a) and (b)

$P' \Downarrow$ (c) by i.h.

$P \Downarrow$ by (b) and (c), satisfying (2)

T.S.: If $P \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A_1]$ and $P \Longrightarrow P'$ then $P' \in \mathcal{T}^{\omega}_{\eta}[z{:}\forall X.A_1]$
Follows directly from definition of weak transition.

□

A.5 Proof Conversions

Definition 35 (Proof Conversions)**.** *We define $\simeq_c$ as the least congruence on processes induced by the process equalities in Figures A.1, A.2, A.3, and A.4.*

We note that not all permutations are sound nor are possible. In particular, for permutability of two inference rules to be sound, one of them has to be a left rule; the permutation of two right

$$\Gamma; \Delta \Rightarrow (\nu x)(P \mid \overline{z}\langle y\rangle.(Q \mid R)) \simeq_c \overline{z}\langle y\rangle.((\nu x)(P \mid Q) \mid R) :: z{:}A \otimes B \tag{A-1}$$

$$\Gamma; \Delta \Rightarrow (\nu x)(P \mid \overline{z}\langle y\rangle.(Q \mid R)) \simeq_c \overline{z}\langle y\rangle.(Q \mid (\nu x)(P \mid R)) :: z{:}A \otimes B \tag{A-2}$$

$$\Gamma; \Delta, y{:}A \otimes B \Rightarrow (\nu x)(P \mid y(z).Q) \simeq_c y(z).(\nu x)(P \mid Q) :: T \tag{A-3}$$

$$\Gamma; \Delta, y{:}A \otimes B \Rightarrow (\nu x)(y(z).P \mid Q) \simeq_c y(z).(\nu x)(P \mid Q) :: T \tag{A-4}$$

$$\Gamma; \Delta \Rightarrow (\nu x)(P \mid z(y).Q) \simeq_c z(y).(\nu x)(P \mid Q) :: z{:}A \multimap B \tag{A-5}$$

$$\Gamma; \Delta, y{:}A \multimap B \Rightarrow (\nu x)(P \mid \overline{y}\langle z\rangle.(Q \mid R)) \simeq_c \overline{y}\langle z\rangle.((\nu x)(P \mid Q) \mid R) :: T \tag{A-6}$$

$$\Gamma; \Delta, y{:}A \multimap B \Rightarrow (\nu x)(P \mid \overline{y}\langle z\rangle.(Q \mid R)) \simeq_c \overline{y}\langle z\rangle.(Q \mid (\nu x)(P \mid R)) :: T \tag{A-7}$$

$$\Gamma; \Delta, y{:}A \multimap B \Rightarrow (\nu x)(\overline{y}\langle z\rangle.(Q \mid P) \mid R) \simeq_c \overline{y}\langle z\rangle.(Q \mid (\nu x)(P \mid R)) :: T \tag{A-8}$$

$$\begin{aligned}\Gamma; \Delta \Rightarrow {} & (\nu x)(P \mid z.\mathsf{case}(Q, R)) \simeq_c \\ & z.\mathsf{case}((\nu x)(P \mid Q), (\nu x)(P \mid R)) :: z{:}A \mathbin{\&} B\end{aligned} \tag{A-9}$$

$$\Gamma; \Delta, y{:}A \mathbin{\&} B \Rightarrow (\nu x)(P \mid y.\mathsf{inl}; Q) \simeq_c y.\mathsf{inl}; (\nu x)(P \mid Q) :: T \tag{A-10}$$

$$\Gamma; \Delta, y{:}A \mathbin{\&} B \Rightarrow (\nu x)(P \mid y.\mathsf{inr}; R) \simeq_c y.\mathsf{inr}; (\nu x)(P \mid R) :: T \tag{A-11}$$

$$\Gamma; \Delta \Rightarrow (\nu x)(P \mid z.\mathsf{inl}; Q) \simeq_c z.\mathsf{inl}; (\nu x)(P \mid Q) :: z{:}A \oplus B \tag{A-12}$$

$$\Gamma; \Delta \Rightarrow (\nu x)(P \mid z.\mathsf{inr}; R) \simeq_c z.\mathsf{inr}; (\nu x)(P \mid R) :: z{:}A \oplus B \tag{A-13}$$

$$\begin{aligned}\Gamma; \Delta, y{:}A \oplus B \Rightarrow {} & (\nu x)(P \mid y.\mathsf{case}(Q, R)) \simeq_c \\ & y.\mathsf{case}((\nu x)(P \mid Q), (\nu x)(P \mid R)) :: T\end{aligned} \tag{A-14}$$

$$\Gamma, u{:}A; \Delta \Rightarrow (\nu x)(P \mid \overline{u}\langle y\rangle.Q) \simeq_c \overline{u}\langle y\rangle.(\nu x)(P \mid Q) :: T \tag{A-15}$$

$$\Gamma; \Delta, y{:}A \otimes B \Rightarrow (\nu x)(y(z).P \mid R) \simeq_c y(z).(\nu x)(P \mid R) :: T \tag{B-1}$$

$$\Gamma; \Delta, y{:}A \multimap B \Rightarrow (\nu x)(\overline{y}\langle z\rangle.(P \mid Q) \mid R) \simeq_c \overline{y}\langle z\rangle.(P \mid (\nu x)(Q \mid R)) :: T \tag{B-2}$$

$$\Gamma; \Delta, y{:}A \mathbin{\&} B \Rightarrow (\nu x)(y.\mathsf{inl}; P \mid R) \simeq_c y.\mathsf{inl}; (\nu x)(P \mid R) :: T \tag{B-3}$$

$$\Gamma; \Delta, y{:}A \mathbin{\&} B \Rightarrow (\nu x)(y.\mathsf{inr}; P \mid R) \simeq_c y.\mathsf{inr}; (\nu x)(P \mid R) :: T \tag{B-4}$$

$$\begin{aligned}\Gamma; \Delta, y{:}A \oplus B \Rightarrow {} & (\nu x)(y.\mathsf{case}(P, Q) \mid R) \simeq_c \\ & y.\mathsf{case}((\nu x)(P \mid R), (\nu x)(Q \mid R)) :: T\end{aligned} \tag{B-5}$$

$$\Gamma, u{:}A; \Delta \Rightarrow (\nu x)(\overline{u}\langle y\rangle.P \mid R) \simeq_c \overline{u}\langle y\rangle.(\nu x)(P \mid R) :: T \tag{B-6}$$

$$\Gamma, u{:}A; \Delta \Rightarrow (\nu x)(P \mid \overline{u}\langle y\rangle.R) \simeq_c \overline{u}\langle y\rangle.(\nu x)(P \mid R) :: T \tag{B-7}$$

Figure A.1: Process equalities induced by proof conversions: Classes (A) and (B)

rules leads to unsound transformations. In the figures, we consider only combinations with rule ($\&\mathsf{L}_1$); permutations involving ($\&\mathsf{L}_2$) are easily derivable. For conciseness, we omit permutations involving rule ($\mathbf{1}\mathsf{L}$), given that it can permute with all rules (promoting or demoting the dummy input prefix accordingly). For readability purposes, we use the common shorthand $\overline{z}\langle y\rangle.P$ to denote the bound-output $(\nu y)z\langle y\rangle.P$.

A.5.1 Additional cases for the Proof of Theorem 48

We repeat the theorem statement and detail some additional cases, complementing the proof given in the main document.

Theorem 49 (Soundness of Proof Conversions)**.** *Let P, Q be processes such that:*

(i) $\Gamma; \Delta \Rightarrow P :: z{:}A$,

(ii) $\Gamma; \Delta \Rightarrow Q :: z{:}A$ *and,*

(iii) $P \simeq_c Q$.

$$\Gamma; \cdot \Rightarrow (\boldsymbol{\nu} u)((!u(y).P) \mid z\langle\rangle.\mathbf{0}) \simeq_c z\langle\rangle.\mathbf{0} :: z{:}\mathbf{1} \tag{C-1}$$

$$\begin{aligned}\Gamma; \Delta \Rightarrow {}& (\boldsymbol{\nu} u)((!u(y).P) \mid \overline{x}\langle z\rangle.(Q \mid R)) \simeq_c \\ & \overline{x}\langle z\rangle.((\boldsymbol{\nu} u)((!u(y).P) \mid Q) \mid (\boldsymbol{\nu} u)((!u(y).P) \mid R)) :: x{:}A \otimes B\end{aligned} \tag{C-2}$$

$$\Gamma; \Delta, y{:}A \otimes B \Rightarrow (\boldsymbol{\nu} u)((!u(y).P) \mid y(z).Q) \simeq_c y(z).(\boldsymbol{\nu} u)((!u(y).P) \mid Q) :: T \tag{C-3}$$

$$\Gamma; \Delta \Rightarrow (\boldsymbol{\nu} u)(!u(y).P \mid z(y).Q) \simeq_c z(y).(\boldsymbol{\nu} u)(!u(y).P \mid Q) :: z{:}A \multimap B \tag{C-4}$$

$$\begin{aligned}\Gamma; \Delta, y{:}A \multimap B \Rightarrow {}& (\boldsymbol{\nu} u)((!u(w).P) \mid \overline{y}\langle z\rangle.(Q \mid R)) \simeq_c \\ & \overline{y}\langle z\rangle.(((\boldsymbol{\nu} u)(!u(w).P \mid Q) \mid (\boldsymbol{\nu} u)((!u(w).P) \mid R))) :: T\end{aligned} \tag{C-5}$$

$$\begin{aligned}\Gamma; \Delta \Rightarrow {}& (\boldsymbol{\nu} u)((!u(y).P) \mid z.\mathsf{case}(Q, R)) \simeq_c \\ & z.\mathsf{case}((\boldsymbol{\nu} u)((!u(y).P) \mid Q), (\boldsymbol{\nu} u)((!u(y).P) \mid R))::z{:}A \,\&\, B\end{aligned} \tag{C-6}$$

$$\Gamma; \Delta, y{:}A \,\&\, B \Rightarrow (\boldsymbol{\nu} u)(!u(z).P \mid y.\mathsf{inl}; Q) \simeq_c y.\mathsf{inl}; (\boldsymbol{\nu} u)(!u(z).P \mid Q) :: T \tag{C-7}$$

$$\Gamma; \Delta, y{:}A \,\&\, B \Rightarrow (\boldsymbol{\nu} u)(!u(z).P \mid y.\mathsf{inr}; R) \simeq_c y.\mathsf{inr}; (\boldsymbol{\nu} u)(!u(z).P \mid R) :: T \tag{C-8}$$

$$\begin{aligned}\Gamma; \Delta \Rightarrow {}& (\boldsymbol{\nu} u)(!u(y).P \mid z.\mathsf{inl}; Q) \simeq_c \\ & z.\mathsf{inl}; (\boldsymbol{\nu} u)(!u(y).P \mid Q) :: z{:}A \oplus B\end{aligned} \tag{C-9}$$

$$\begin{aligned}\Gamma; \Delta \Rightarrow {}& (\boldsymbol{\nu} u)(!u(y).P \mid z.\mathsf{inr}; R) \simeq_c \\ & z.\mathsf{inr}; (\boldsymbol{\nu} u)(!u(y).P \mid R) :: z{:}A \oplus B\end{aligned} \tag{C-10}$$

$$\begin{aligned}\Gamma; \Delta, y{:}A \oplus B \Rightarrow {}& (\boldsymbol{\nu} u)(!u(z).P \mid y.\mathsf{case}(Q, R)) \simeq_c \\ & y.\mathsf{case}((\boldsymbol{\nu} u)(!u(z).P \mid Q), (\boldsymbol{\nu} u)(!u(z).P \mid R)) :: T\end{aligned} \tag{C-11}$$

$$\Gamma; \Delta \Rightarrow (\boldsymbol{\nu} u)(!u(y).P \mid !x(z).Q) \simeq_c !x(z).(\boldsymbol{\nu} u)(!u(y).P \mid Q) :: x{:}!A \tag{C-12}$$

$$\Gamma; \Delta \Rightarrow (\boldsymbol{\nu} u)(!u(y).P \mid \overline{v}\langle y\rangle.Q) \simeq_c \overline{v}\langle y\rangle.(\boldsymbol{\nu} u)(!u(y).P \mid Q)) :: T \tag{C-13}$$

Figure A.2: Process equalities induced by proof conversions: Class (C)

We have that $\Gamma; \Delta \Rightarrow P \approx_{\mathrm{L}} Q :: z{:}A[\emptyset : \emptyset \Leftrightarrow \emptyset]$

Proof. We detail the cases of proof conversions A-4 (cf. Figure A.1), and C-11 (cf. Figure A.2).

Proof conversion A-4

We have that

$$\Gamma; \Delta, y{:}A \otimes B \Rightarrow (\boldsymbol{\nu} x)(P_1 \mid y(z).P_2) :: T$$
$$\Gamma; \Delta, y{:}A \otimes B \Rightarrow y(z).(\boldsymbol{\nu} x)(P_1 \mid P_2) :: T$$

with

$$\Gamma; \Delta_1 \Rightarrow P_1 :: x{:}C \quad \Gamma; \Delta_2, x{:}C, z{:}A, y{:}B \Rightarrow P_2 :: T \tag{A.1}$$

and $\Delta = \Delta_1, \Delta_2$. Let $M = (\boldsymbol{\nu} x)(P_1 \mid y(z).P_2)$ and $N = y(z).(\boldsymbol{\nu} x)(P_1 \mid P_2)$.

By Lemma 37 we have to show that for any $(R_1, R_2) \in \mathcal{C}_\Gamma$, any $(S_1, S_2) \in \mathcal{C}_\Delta$ and any $Q_1 \approx_{\mathrm{L}} Q_2 :: y{:}A \otimes B$ we have:

$$(\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_1 \mid S_1 \mid Q_1 \mid M) \approx_{\mathrm{L}} (\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_2 \mid S_2 \mid Q_2 \mid N) :: T$$

Suppose $Q_1 \overset{\overline{(\nu z)y\langle z\rangle}}{\Longrightarrow} Q_1'$. We know that $Q_2 \overset{\overline{(\nu z)y\langle z\rangle}}{\Longrightarrow} Q_2'$, such that the continuations can be structurally decomposed (via $\equiv_!$) into logically equivalent processes at $z{:}A$ and $y{:}B$. We consider how the synchronization of Q_1 and M can take place: we may have weak internal

$$\Gamma; \Delta, x{:}A \otimes B, z{:}C \otimes D \Rightarrow x(y).z(w).P \simeq_c z(w).x(y).P :: T \tag{D-1}$$

$$\Gamma; \Delta, z{:}D \multimap C, x{:}A \multimap B \Rightarrow \overline{z}\langle w\rangle.(R \mid \overline{x}\langle y\rangle.(P \mid Q)) \simeq_c \overline{x}\langle y\rangle.(P \mid \overline{z}\langle w\rangle.(R \mid Q)) :: T \tag{D-2}$$

$$\Gamma; \Delta, z{:}D \multimap C, x{:}A \multimap B \Rightarrow \overline{z}\langle w\rangle.(R \mid \overline{x}\langle y\rangle.(P \mid Q)) \simeq_c \overline{x}\langle y\rangle.(\overline{z}\langle w\rangle.(R \mid P) \mid Q) :: T \tag{D-3}$$

$$\Gamma; \Delta, w{:}C \multimap D, x{:}A \otimes B \Rightarrow \overline{w}\langle z\rangle.(Q \mid x(y).P) \simeq_c x(y).\overline{w}\langle z\rangle.(Q \mid P) :: T \tag{D-4}$$

$$\Gamma; \Delta, w{:}C \multimap D, x{:}A \otimes B \Rightarrow \overline{w}\langle z\rangle.(x(y).P \mid Q) \simeq_c x(y).\overline{w}\langle z\rangle.(P \mid Q) :: T \tag{D-5}$$

$$\Gamma, u{:}A, v{:}C; \Delta \Rightarrow \overline{u}\langle y\rangle.\overline{v}\langle x\rangle.P \simeq_c \overline{v}\langle x\rangle.\overline{u}\langle y\rangle.P :: T \tag{D-6}$$

$$\Gamma, u{:}C; \Delta, x{:}A \multimap B \Rightarrow \overline{u}\langle z\rangle.\overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(\overline{u}\langle z\rangle.P \mid Q) :: T \tag{D-7}$$

$$\Gamma, u{:}C; \Delta, x{:}A \multimap B \Rightarrow \overline{u}\langle z\rangle.\overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(P \mid \overline{u}\langle z\rangle.Q) :: T \tag{D-8}$$

$$\Gamma, u{:}A; \Delta, z{:}C \otimes D \Rightarrow \overline{u}\langle y\rangle.z(w).P \simeq_c z(w).\overline{u}\langle y\rangle.P :: T \tag{D-9}$$

$$\Gamma; \Delta, x{:}A \oplus B, y{:}C \oplus D \Rightarrow y.\mathsf{case}(x.\mathsf{case}(P_1, Q_1), x.\mathsf{case}(P_2, Q_2)) \simeq_c x.\mathsf{case}(y.\mathsf{case}(P_1, P_2), y.\mathsf{case}(Q_1, Q_2)) :: T \tag{D-10}$$

$$\Gamma, u{:}C; \Delta, x{:}A \oplus B \Rightarrow \overline{u}\langle z\rangle.x.\mathsf{case}(P, Q) \simeq_c x.\mathsf{case}(\overline{u}\langle z\rangle.P\,,\, \overline{u}\langle z\rangle.Q) :: T \tag{D-11}$$

$$\Gamma; \Delta, w{:}A \multimap E, z{:}C \oplus D \Rightarrow z.\mathsf{case}(\overline{w}\langle y\rangle.(P \mid R_1)\,,\, \overline{w}\langle y\rangle.(P \mid R_2)) \simeq_c \overline{w}\langle y\rangle.(P \mid z.\mathsf{case}(R_1, R_2)) :: T \tag{D-12}$$

$$\Gamma; \Delta, z{:}C \oplus D, x{:}A \otimes B \Rightarrow z.\mathsf{case}(x(y).P, x(y).Q) \simeq_c x(y).z.\mathsf{case}(P, Q) :: T \tag{D-13}$$

$$\Gamma; \Delta, x{:}A \mathbin{\&} B, y{:}C \mathbin{\&} D \Rightarrow x.\mathsf{inl}; y.\mathsf{inl}; P \simeq_c y.\mathsf{inl}; x.\mathsf{inl}; P :: T \tag{D-14}$$

$$\Gamma; \Delta, x{:}A \oplus B, y{:}C \mathbin{\&} D \Rightarrow x.\mathsf{case}(y.\mathsf{inl}; P, y.\mathsf{inl}; Q) \simeq_c y.\mathsf{inl}; x.\mathsf{case}(P, Q) :: T \tag{D-15}$$

$$\Gamma, u{:}C; \Delta, z{:}A \mathbin{\&} B \Rightarrow z.\mathsf{inl}; \overline{u}\langle y\rangle.P \simeq_c \overline{u}\langle y\rangle.z.\mathsf{inl}; P :: T \tag{D-16}$$

$$\Gamma; \Delta, z{:}C \mathbin{\&} D, x{:}A \multimap B \Rightarrow z.\mathsf{inl}; \overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(z.\mathsf{inl}; P \mid Q) :: T \tag{D-17}$$

$$\Gamma; \Delta, z{:}C \mathbin{\&} D, x{:}A \multimap B \Rightarrow z.\mathsf{inl}; \overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(P \mid z.\mathsf{inl}; Q) :: T \tag{D-18}$$

$$\Gamma; \Delta, z{:}C \mathbin{\&} D, x{:}A \otimes B \Rightarrow z.\mathsf{inl}; x(y).P \simeq_c x(y).z.\mathsf{inl}; P :: T \tag{D-19}$$

Figure A.3: Process equalities induced by proof conversions: Class (D).

transitions from S_1, Q_1 or P_1. Weak internal transitions of Q_1 can always be matched by those of Q_2. Weak internal transitions of S_1 can similarly be matched by S_2. Weak internal transitions of P_1 in N are blocked. Synchronization between S_1 and P_1 is possible, which cannot be matched outright between S_2 and P_1 due to the input prefix. Let us consider this latter case:

$$(\boldsymbol{\nu}\widetilde{x}, \widetilde{u})(R_1 \mid S_1 \mid Q_1 \mid (\boldsymbol{\nu}x)(P_1 \mid y(z).P_2)) \Longrightarrow (\boldsymbol{\nu}\widetilde{x}, \widetilde{u})(R_1 \mid S_1' \mid Q_1 \mid (\boldsymbol{\nu}x)(P_1' \mid y(z).P_2))$$
$$\Longrightarrow (\boldsymbol{\nu}\widetilde{x}, \widetilde{u})(R_1 \mid S_1' \mid Q_1' \mid (\boldsymbol{\nu}x)(P_1' \mid P_2))$$

$$(\boldsymbol{\nu}\widetilde{x}, \widetilde{u})(R_2 \mid S_2 \mid Q_2 \mid y(z).(\boldsymbol{\nu}x)(P_1 \mid P_2)) \Longrightarrow (\boldsymbol{\nu}\widetilde{x}, \widetilde{u})(R_2 \mid S_2' \mid Q_2 \mid y(z).(\boldsymbol{\nu}x)(P_1 \mid P_2))$$
$$\Longrightarrow (\boldsymbol{\nu}\widetilde{x}, \widetilde{u})(R_2 \mid S_2' \mid Q_2' \mid (\boldsymbol{\nu}x)(P_1' \mid P_2))$$

In essence, we "delay" the weak transitions that result from synchronizations between S_1 and P_1 until after the output from Q_2. We can conclude the proof by parametricity, backward closure (and forward closure for reductions of P_1). The other weak internal transition combinations follow accordingly.

Proof conversion C-11

$$\Gamma; \Delta, z{:}C \mathbin{\&} D \Rightarrow z.\mathsf{inl}; \overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(P \mid z.\mathsf{inl}; Q) :: x{:}A \otimes B \tag{E-1}$$

$$\Gamma; \Delta, z{:}C \mathbin{\&} D \Rightarrow z.\mathsf{inl}; \overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(z.\mathsf{inl}; P \mid Q) :: x{:}A \otimes B \tag{E-2}$$

$$\begin{aligned}\Gamma; \Delta, z{:}D \oplus E \Rightarrow\ & z.\mathsf{case}(\overline{x}\langle y\rangle.(P_1 \mid Q)\,,\ \overline{x}\langle y\rangle.(P_2 \mid Q)) \simeq_c \\ & \overline{x}\langle y\rangle.(Q \mid z.\mathsf{case}(P_1\,,\ P_2)) :: x{:}A \otimes B\end{aligned} \tag{E-3}$$

$$\begin{aligned}\Gamma; \Delta, z{:}D \oplus E \Rightarrow\ & z.\mathsf{case}(\overline{x}\langle y\rangle.(Q \mid P_1)\,,\ \overline{x}\langle y\rangle.(Q \mid P_2)) \simeq_c \\ & \overline{x}\langle y\rangle.(z.\mathsf{case}(P_1\,,\ P_2) \mid Q) :: x{:}A \otimes B\end{aligned} \tag{E-4}$$

$$\Gamma, u{:}C; \Delta \Rightarrow \overline{u}\langle w\rangle.\overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(\overline{u}\langle w\rangle.P \mid Q) :: x{:}A \otimes B \tag{E-5}$$

$$\Gamma, u{:}C; \Delta \Rightarrow \overline{u}\langle w\rangle.\overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(P \mid \overline{u}\langle w\rangle.Q) :: x{:}A \otimes B \tag{E-6}$$

$$\begin{aligned}\Gamma; \Delta, w{:}C \multimap D \Rightarrow\ & \overline{w}\langle z\rangle.(R \mid \overline{x}\langle y\rangle.(P \mid Q)) \simeq_c \\ & \overline{x}\langle y\rangle.(P \mid \overline{w}\langle z\rangle.(R \mid Q)) :: x{:}A \otimes B\end{aligned} \tag{E-7}$$

$$\begin{aligned}\Gamma; \Delta, x{:}C \multimap D \Rightarrow\ & \overline{z}\langle y\rangle.(\overline{x}\langle w\rangle.(P \mid Q) \mid R) \simeq_c \\ & \overline{x}\langle w\rangle.(P \mid \overline{z}\langle y\rangle.(R \mid Q)) :: z{:}A \otimes B\end{aligned} \tag{E-8}$$

$$\Gamma; \Delta, z{:}C \otimes D \Rightarrow z(w).\overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(z(w).P \mid Q) :: x{:}A \otimes B \tag{E-9}$$

$$\Gamma; \Delta, z{:}C \otimes D \Rightarrow z(w).\overline{x}\langle y\rangle.(P \mid Q) \simeq_c \overline{x}\langle y\rangle.(P \mid z(w).Q) :: x{:}A \otimes B \tag{E-10}$$

$$\Gamma; \Delta, z{:}C \mathbin{\&} D \Rightarrow z.\mathsf{inl}; x(y).P \simeq_c x(y).z.\mathsf{inl}; P :: x{:}A \multimap B \tag{E-11}$$

$$\Gamma; \Delta, z{:}C \oplus D \Rightarrow x(y).z.\mathsf{case}(P, Q) \simeq_c z.\mathsf{case}(x(y).P\,,\ x(y).Q) :: x{:}A \multimap B \tag{E-12}$$

$$\Gamma, u{:}C; \Delta \Rightarrow \overline{u}\langle w\rangle.x(y).P \simeq_c x(y).\overline{u}\langle w\rangle.P :: x{:}A \multimap B \tag{E-13}$$

$$\Gamma; \Delta, w{:}C \multimap D \Rightarrow \overline{w}\langle z\rangle.(R \mid x(y).P)) \simeq_c x(y).\overline{w}\langle z\rangle.(R \mid P)) :: x{:}A \multimap B \tag{E-14}$$

$$\Gamma; \Delta, z{:}C \otimes D \Rightarrow x(y).z(w).P \simeq_c z(w).x(y).P :: x{:}A \multimap B \tag{E-15}$$

$$\Gamma; \Delta, y{:}C \mathbin{\&} D \Rightarrow y.\mathsf{inl}; x.\mathsf{case}(P, Q) \simeq_c x.\mathsf{case}(y.\mathsf{inl}; P\,,\ y.\mathsf{inl}; Q) :: x{:}A \mathbin{\&} B \tag{E-16}$$

$$\begin{aligned}\Gamma; \Delta, y{:}C \oplus D \Rightarrow\ & x.\mathsf{case}(y.\mathsf{case}(P_1, Q_1), y.\mathsf{case}(P_2, Q_2)) \simeq_c \\ & y.\mathsf{case}(x.\mathsf{case}(P_1, P_2), x.\mathsf{case}(Q_1, Q_2)) :: x{:}A \mathbin{\&} B\end{aligned} \tag{E-17}$$

$$\Gamma; u{:}A; \Delta \Rightarrow x.\mathsf{case}(\overline{u}\langle y\rangle.P\,,\ \overline{u}\langle y\rangle.Q) \simeq_c \overline{u}\langle y\rangle.x.\mathsf{case}(P, Q) :: x{:}A \mathbin{\&} B \tag{E-18}$$

$$\begin{aligned}\Gamma; \Delta, z{:}C \multimap D \Rightarrow\ & \overline{z}\langle y\rangle.(R \mid x.\mathsf{case}(P, Q)) \simeq_c \\ & x.\mathsf{case}(\overline{z}\langle y\rangle.(R \mid P)\,,\ \overline{z}\langle y\rangle.(R \mid Q)) :: x{:}A \mathbin{\&} B\end{aligned} \tag{E-19}$$

$$\Gamma; \Delta, x{:}A \otimes B \Rightarrow z.\mathsf{case}(x(y).P\,,\ x(y).Q) \simeq_c x(y).z.\mathsf{case}(P, Q) :: z{:}C \mathbin{\&} D \tag{E-20}$$

$$\Gamma; \Delta, y{:}C \mathbin{\&} D \Rightarrow y.\mathsf{inl}; x.\mathsf{inl}; P \simeq_c x.\mathsf{inl}; y.\mathsf{inl}; P :: x{:}A \oplus B \tag{E-21}$$

$$\Gamma; \Delta, y{:}A \oplus B \Rightarrow x.\mathsf{inl}; y.\mathsf{case}(P, Q) \simeq_c y.\mathsf{case}(x.\mathsf{inl}; P, x.\mathsf{inl}; Q) :: x{:}A \oplus B \tag{E-22}$$

$$\Gamma; u{:}A; \Delta \Rightarrow x.\mathsf{inl}; \overline{u}\langle y\rangle.P \simeq_c \overline{u}\langle y\rangle.x.\mathsf{inl}; P :: x{:}A \oplus B \tag{E-23}$$

$$\Gamma; \Delta, z{:}D \multimap C \Rightarrow \overline{z}\langle y\rangle.(Q \mid x.\mathsf{inl}; P) \simeq_c x.\mathsf{inl}; \overline{z}\langle y\rangle.(Q \mid P) :: x{:}A \oplus B \tag{E-24}$$

$$\Gamma; \Delta, x{:}A \otimes B \Rightarrow x(y).z.\mathsf{inl}; P \simeq_c z.\mathsf{inl}; x(y).P :: z{:}C \oplus D \tag{E-25}$$

$$\Gamma; \Delta, x{:}!C \Rightarrow \overline{z}\langle y\rangle.(x(u).P \mid Q) \simeq_c (x(u).\overline{z}\langle y\rangle.(P \mid Q)) :: z{:}A \otimes B \tag{E-26}$$

$$\Gamma; \Delta, x{:}!C \Rightarrow \overline{z}\langle y\rangle.(P \mid x(u).Q) \simeq_c x(u).\overline{z}\langle y\rangle.(P \mid Q)) :: z{:}A \otimes B \tag{E-27}$$

Figure A.4: Process equalities induced by proof conversions: Class (E).

We have that

$$\Gamma; \Delta, y{:}A \oplus B \Rightarrow (\boldsymbol{\nu} u)((!u(z).P_1) \mid y.\mathsf{case}(P_2, P_3)) :: T$$

$$\Gamma; \Delta, y{:}A \oplus B \Rightarrow y.\mathsf{case}((\boldsymbol{\nu} u)((!u(z).P_1) \mid P_2), (\boldsymbol{\nu} u)((!u(z).P_1) \mid P_3)) :: T$$

with

$$\Gamma; \cdot \Rightarrow P_1 :: z{:}C \quad \Gamma, u{:}C; \Delta_1, y{:}A \Rightarrow P_2 :: T \quad \Gamma, u{:}C; \Delta_2, y{:}B \Rightarrow P_3 :: T \tag{A.2}$$

and $\Delta = \Delta_1, \Delta_2$.

Let $M = (\boldsymbol{\nu} u)((!u(z).P_1) \mid y.\mathsf{case}(P_2, P_3))$ and

$N = y.\mathsf{case}((\boldsymbol{\nu} u)((!u(z).P_1) \mid P_2), (\boldsymbol{\nu} u)((!u(z).P_1) \mid P_3))$.

By Lemma 37 we have to show that for any $(R_1, R_2) \in \mathcal{C}_\Gamma$, any $(S_1, S_2) \in \mathcal{C}_\Delta$ and any $Q_1 \approx_{\mathrm{L}} Q_2 :: y{:}A \oplus B$ we have:

$$(\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_1 \mid S_1 \mid Q_1 \mid M) \approx_{\mathrm{L}} (\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_2 \mid S_2 \mid Q_2 \mid N) :: T$$

Suppose $Q_1 \xRightarrow{\overline{y.\mathsf{inl}}} Q_1'$. We know that $Q_2 \xRightarrow{\overline{y.\mathsf{inl}}} Q_2'$ such that $Q_1' \approx_{\mathrm{L}} Q_2' :: y{:}A$. No internal actions are possible in R_1. All internal transitions from S_1 can be matched by S_2 and synchronizations between S_1 and P_2 or P_3 are blocked before the synchronization with Q_1. Similarly for S_2. We thus have,

$$(\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_1 \mid S_1 \mid Q_1 \mid (\boldsymbol{\nu} u)((!u(z).P_1) \mid y.\mathsf{case}(P_2, P_3))) \Longrightarrow$$
$$(\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_1 \mid S_1' \mid Q_1' \mid (\boldsymbol{\nu} u)((!u(z).P_1) \mid P_2))$$

$$(\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_2 \mid S_2 \mid Q_2 \mid y.\mathsf{case}((\boldsymbol{\nu} u)((!u(z).P_1) \mid P_2), (\boldsymbol{\nu} u)((!u(z).P_1) \mid P_3))) \Longrightarrow$$
$$(\boldsymbol{\nu} \widetilde{x}, \widetilde{u})(R_2 \mid S_2' \mid Q_2' \mid (\boldsymbol{\nu} u)((!u(z).P_1) \mid P_2))$$

We can conclude the proof by appealing to parametricity and backward closure. Other possibilities for internal action follow by similar reasoning, appealing to forward closure. The case where $Q_1 \xRightarrow{\overline{y.\mathsf{inr}}} Q_1'$ is identical to the above.

□

2015

Bernardo Toninho

A Logical Foundation for Session-based Concurrent Computation