Università di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

INF 01

Ph.D. Thesis

A Language-based Approach to Distributed Resources

Viet Dung Dinh

Supervisors	Referees
Prof. Chiara Bodei	Prof. António Ravara
Prof. Gian Luigi Ferrari	Dr. Emilio Tuosto

Chair

Prof. Pierpaolo Degano

May 28, 2012

Abstract

Modern computing paradigms for distributed applications advocate a strong control on shared resources available on demand in order to guarantee their correct usages. An illustrative example of such paradigms is Cloud Computing. In this dissertation, we study formal models for distributed applications, paying particular attention to resource usage analysis. Formal methods for specifying and analysing different aspects of resource management could play an important role for the widespread usages of distributed resources. They provide not only the theoretical framework to understand the stages underlying the design and implementation issues, but also the mathematically-based techniques for the specification and verifications of properties of such systems. In this dissertation, we introduce two models, called $\lambda^{\{\}}$-calculus and G-Local π-calculus, which are extensions of λ-calculus and π-calculus respectively.

The $\lambda^{\{\}}$-calculus is an extension of concurrent λ-calculus enriched with suitable mechanisms to express and enforce application-level security policies governing usages of resources available on demand in the clouds. We focus on the server side of cloud systems, by adopting a *pro-active* approach, where explicit security policies, which are expressed as a set of execution traces, regulate server's behaviour. By providing an abstract cloud semantics, we ensure that enforcing security policies embedded in cloud applications is *sound*.

The G-Local π-calculus is built on top of the standard π-calculus by introducing new primitives to manage resources. Unlike the previous model, where resources are highly abstract, resources in this approach are modelled as stateful entities with local states and global policies. A high degree of loose coupling among applications and resources is achieved through the *publish/subscribe* model. Furthermore, we develop two static, language-based techniques, namely Control Flow Analysis (CFA) and Type and Effect Systems, to reason about resource usages and therefore able to predict *bad* usages of resources. The CFA mainly focuses on reachability properties related to resource usages. It computes an over-approximation of resource usages of applications. As a result, if the approximation does not contain *bad* usages, then it guarantees that applications *correctly* use resources. The type and effect system provides a closer view of resource behaviour. Resource behaviour is extracted in the form of side effect of the type system. We exploit side effect to verify regular linear time properties, expressed by Linear Time Logic formulas, of resource usages.

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisors, Prof. Chiara Bodei and Prof. Gian Luigi Ferrari, who guided me through technical issues of my work and helped me focus on research. I would have unable to complete this thesis without their support, lessons and patience.

I wish to thank my external reviewers, Prof. António Ravara and Dr. Emilio Tuosto, for their valuable and detailed comments, and the thesis committee members, Prof. Antonio Brogi and Prof. Pierpaolo Degano, for their precious comments and suggestions.

I am indebted to my friends and colleagues An, Claudio, Dung, Giovanni, Hieu, Igor, Ha, Lopa, Luca, Lam, Mateo, Minh, Naveen, Peter, Rebecca and Rui, and to friends from my football team, Kim, Liem, Tim, The and Vu. With them, life in Pisa was far more enjoyable.

Finally, I would like to thank my parents and my sister for their love and support throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Nowadays, the evolution of network infrastructures and computing technologies heavily impacts on the design of software applications. It is reflected by the shift from (traditional) applications running in a well-determined environment to distributed applications running into a dynamic evolving environment. This trend has led to introducing or revising different computing paradigms such as Service-oriented Computing, Cloud Computing and Ubiquitous Computing. Service-oriented Computing [91] is based on the idea of providing a *network of services*, which are basically loosely-coupled basic computing entities. The network of services is exploited to create a flexible way to assemble services into effective applications. The advantage of having high performance network infrastructures allows to rapidly deploy and scale services at runtime, i.e. *on-demand deployment*. This is one of the key ideas of Cloud Computing [37]. Basically, cloud-based applications allow for an intensive usage of distributed resources. The integration of iCloud on Apple products, e.g. iOS and Mac OSX, to store/access information on cloud storage is an illustrative example of this trend. The promise of Ubiquitous Computing [109] is to embed "computing devices" into daily activities and let them work transparently to provide feedback or adjust themselves in accordance with novel configured settings. Close to Ubiquitous Computing is the idea of Internet of Things [7], where "things" or identifiable objects that are connected over the Internet are capable of gathering, processing, analysing information around them. The fact that applications in these visionary paradigms are able to access a variety of ubiquitous resources requires a development of a new framework to design and implement applications, where resource management is a central concern.

In this dissertation, we focus on the design of suitable mechanisms to control the distributed management of resources. Resources can be *geographically distributed* (possibly over continents) and *independent*, and could be accessed at any time from anywhere. The geographic distribution recalls the *loosely coupled* design methodology of SOC. However, unlike SOC services, which are often autonomous and interact with users through pre-defined protocols (that is, users need to follow the protocols provided by SOC services), distributed resources are subjects to usage policies, provided that users must employ them correctly. The *publish-subscribe* paradigm assumes a notable role in this view. Indeed, the publish-subscribe paradigm is not only a natural choice to represent distributed resources, but it also emphasises the fact that resources have to be published by external parties and therefore have to be available to everyone through appropriate requests. This form of "plug and play" strongly requires suitable mechanisms to guarantee correctness of usages.

Understanding the foundations of the distributed management of resources could support state-of-the-art advances of programming language constructs, algorithms and reasoning techniques for resource-aware programming. In this perspective, formal methods for specifying and analysing system behaviours can offer an important support. On the one hand, they provide the theoretical framework to understand the stages underlying the design and the implementation issues of software systems. On the other hand, they support the mathematically-based techniques for the specification and verifications of properties of such systems. Implementation of distributed applications with intensive resource usages in turn requires development of *resource-aware programming languages*. In other words, the programming model should have first-class primitives for resource management and able to describe interactions between resources and applications that use them.

In the last few years, many formalisms have been developed to manage resource usages. The focus of these research activities ([11, 64, 19, 85], to mention only a few) is mainly on verifying abstract resource behaviour at a high level view, i.e. without an explicit model of resources. The high level abstraction of resources is often too general to describe a variety of resources. We believe that developing *explicit models of resources* is the first step for understanding resource behaviours. Resources should be modelled as independent entities with their states and properties that are subject to security policies. In this dissertation, we advocate the idea of history-based access control [1] to specify *trace-based* properties of resources. Indeed, we think that trace-based approach equipped with *suitable reasoning techniques* allow us to smoothly verify resource usages.

Process calculi are a natural choice to model distributed resources. In the π-calculus, resources are just names. Behavioural types [63, 4] allow to express properties of names related to distribution and concurrency. However, names themselves are too abstract to express interesting properties, for instance, whether an application uses correctly resources or not. To address this, the works presented in [64, 19, 66] introduce an abstract model of resources in terms of execution traces. The works reported in [64, 19] abstract away resource management, while in [66] resource management is provided by the semantics of private names. Alternatively, the work presented in [36] represents resources as sets of constraints: this choice allows one to represent service level agreement between users and resource providers. Still, this view does not guarantee the correctness of resource usages. The work reported in [47] introduced a monoidal structure of resources, whose semantics is related to the sharing semantics provided by the binary operator in the monoid. However, co-evolution of processes and resources causes co-dependency, hence a rigid interaction between processes and resources. We prefer for an alternative view, based on the idea of emphasising loosely coupling nature of interaction between resources and processes. The above discussion urges the need of a novel and innovative approach to usages of distributed resources, which provides a more precise view of distributed-resource behaviour. We believe that the model has to support a loosely coupled design methodology and it provides a basis for verifying correctness of resource usages.

The aim of this thesis is to bring together a variety of techniques to address issues arising in the new environment underpinned by the fast growth of network infrastructure and computing technologies. First, in our approach, resources are first class entities in the programming model and are explicitly modelled. Second, language-based techniques naturally permit to deal with resource-aware programming constructs. Third, language-based techniques allow one to establish a high level of abstraction not only for reasoning semantically on the behaviour of the whole system, but also for extracting properties of individual components, e.g. resources. Finally, we develop algorithms to verify correctness of resource usages, which is a primary concern in our approach.

1.2 Structural Operational Semantics

Understanding the precise semantics of programming languages plays an important role in the development of high-level programming languages. One of the main approaches to define formal semantics is *operational semantics*, introduced in the sixties [80, 77]. A main *break through* has been provided by Plotkin [93] with the introduction of so called *structural operational semantics* (SOS). SOS provides a way of describing the meaning of computing systems through a set of inference rules, which describe the evolution of the systems in a compositional manner. An inference rule is defined of the form:

$$\frac{promises}{conclusion}$$

where if the premises are satisfied, so does the conclusion. In this way, SOS defines the meaning of a program in terms of the meaning of its parts, thus providing a structural, i.e., *syntax-directed* and *inductive* view of operational semantics.

SOS gives the basis for formal tools to statically analyse behaviours of computing systems due to its compositional nature. Rule-based syntax-directed approach for describing program behaviours in compositional manner gives a basis for proving properties of computing systems, that can be obtained or derived from properties of its components.

1.3 The λ-calculus

The λ-calculus was first introduced by Church during the 1930s as a formal system for studying computable recursive functions. Later, in the 1960s, Landin exploited the λ-calculus as the core mechanism of programming languages [74, 75]. Following Landin's insight, the λ-calculus has been largely used in programming language design and implementation, and in the study of type systems. Its importance arises from the fact that it can be viewed simultaneously as a simple programming language (in the functional style) able to describe computations and as a mathematical object on which rigorous statements can be proved.

Despite its simple definition, the λ-calculus not only plays an important role in the development of programming languages, but it also finds application in many fields of computer science. One of the major applications is type theory. The simply typed λ-calculus, introduced in [46], provides a typed interpretation of the λ-calculus. The types, assigned to λ-elements, correspond to propositions in the intuitionistic logic via the type system, built on the simply typed λ-calculus. This correspondence is known under the name of Curry-Howard isomorphism. From the logic point of view, the proofs of logical formulas can be seen as programs, and therefore λ-elements. As a programming language, the formula that a program proves is the type of that program. By exploiting the dual view of the type system, one can specify properties of programs using logical formulas: the type system ensures that programs meet the required specification. From this point of view, type checking usually provide static guarantee: no error of a certain kind can occur at runtime.

1.4 Process Algebras

Process algebras provide a rather high level view of interactive systems, and a valuable tool for specifying and analysing concurrent systems. Fundamental to process algebras is the parallel operator, allowing the decomposition of systems in terms of their concurrent components. Seminal process algebras are

- CCS, Milner's Calculus of Communicating Systems [83],
- CSP, Hoare's Communicating Sequential Processes [34],
- ACP, Bergstra and Klop's Algebra of Communicating Processes [21].

A number of extensions, based on these calculi, has been proposed to deal with various aspects of concurrency. In the π-calculus, in [84], a notion of *mobility*, i.e. dynamic change of the topological structure of processes, is presented. In [44], the locations or scopes of processes are exploited to handle *administrative domains*. Recently, applications of process algebras also exist to address issues in biology (see [95, 43]).

1.5 Static Program Analysis

Static program analysis is the analysis of software performed without actually executing programs. The analysis is applied to some version of the source code. Program analysis offers techniques for computing at compile-time, safe and efficient approximations of the set of configurations or behaviours arising dynamically, i.e. at run-time. By checking these approximations, one can verify several interesting properties of programs. There are three major approaches to program analysis:

- Control Flow Analysis;
- Abstract Interpretation; and
- Type Systems.

Control Flow Analysis. Control Flow Analysis (CFA) has been introduced in the sixties [96]. CFA was mainly developed for functional languages [100, 65], but it found applications in other languages as well, for instance, in concurrent languages [26]. Basically, CFA provides a framework to compute which values or information can reach certain program points or can be assigned to a specific variable. The idea behind CFA is the specification of rules for transferring all possible information, from one program point to another. Thus, CFA usually gives an over-approximation of actual executions of programs. The correctness of programs is then guaranteed if no *bad* execution is found in the over-approximation.

Abstract Interpretation. Often, concrete and precise information about program properties is in general not computable within finite constraints in time and space. By abstracting the concrete semantics of computing systems, abstract program properties can be easily obtained to a certain degree of abstraction. This is the idea behind abstract interpretation, introduced in [48, 49]. Abstract Interpretation can be viewed as a theory of sound approximation of the semantics of computer programs. It can be viewed as a partial execution of a computing system, since it executes on the abstract semantics without performing all the computations. The relevant feature of abstract interpretation is that a property proved in the abstract semantics also holds in the concrete semantics.

Type Systems. We have already pointed out that type systems are a formal tool for reasoning about programs. By associating a *type* to each computed value, type systems provides a tractable syntactic method of proving the absence of certain programming errors.

1.6 Contributions

The dissertation aims at introducing a foundational framework for specifying and proving properties of distributed resources. The framework is based on the following ingredients:

- **Development of an abstract resource-aware programming language** for providing a basis for programming abstractions for resource-awareness, and for managing resource usages.
- **Models of resources**: explicit mechanisms to express and enforce policies governing usages of resources.
- **Reasoning techniques** to statically check the properties of program behaviour and ensure their safe executions at runtime with respect to resource usages.

Our proposal provides a contribution for the development of languages and software engineering methodologies for securing the design and implementation of distributed resources. We develop two models, based on the λ- and π-calculi, respectively. Both calculi are extended with mechanism to control resource usages. More precisely, the main contributions of the work are the following:

- *The $\lambda^{\{\}}$-calculus*: we use the $\lambda^{\{\}}$-calculus to study cloud-based systems. In this calculus, we view cloud services as *functions with side effects* (the abstract behaviour of cloud services). They are subjected to *security policies.* Sandboxing critical code with security policies ensures that all *bad* behaviours, i.e. those that violate policies, are excluded at run-time. A cloud server is abstractly designed as a triple composed by: i) the global cloud state that represents dependencies among services and resources; ii) the set of active services that serve client requests; iii) the service environment that maps service names into scripts to run the service.

- *The G-Local π-calculus*: we extend the π-calculus by introducing explicit resources and primitives to manage them. Interactions between processes and resources are explicitly modelled and are governed by usage policies. Resources are stateful entities endowed with usage policies. Resource configurations are modelled by structural rules, and therefore they are not under the control of processes. The explicit model of resources allows us to describe various resources and their properties. Moreover, we also provide reasoning techniques to analyse resource behaviour. We adapt two techniques, namely CFA and Type Systems.

  - We extend the CFA introduced in [26] to analyse reachability properties of resources in the G-Local π-calculus. The analysis computes an over-approximation of resource behaviours, which are described by a set of possible traces and their possible contexts.

– A type and effect system is developed for the G-Local π-calculus still to verify resource usages. The novelty of our approach is to separate resource behaviour from process behaviour. To this end, we apply a *symmetric* treatment of input/output on resource-related parts in the type system. Resource behaviour is expressed in terms of basic parallel processes. This allows us to verify *regular* linear temporal properties of resources. Verification of resource usages is decidable in our approach, although we left implementation issues for future work.

1.7 Outline of the Work

The thesis is structured as follows.

- In Chapter 2, we review the main technical background, required in our development. More precisely, in Section 2.1, some of basic notions on transition systems are presented. The λ- and π-calculi are presented in Sections 2.3 and 2.4, respectively. Their properties, expressed as Linear Time Logics, are also introduced. Moreover, we review static analyses, namely CFA and Type Systems, for the λ-calculus and π-calculus.
- In Chapter 3, we introduce the $\lambda^{\{\}}$-calculus, as an extension of the λ-calculus. Furthermore, an abstract cloud semantics is provided.
- In Chapter 4, we introduce the G-Local π-calculus, as an extension of the standard π-calculus with mechanisms to manage resources. We also present a CFA for analysing resource behaviour.
- In Chapter 5, we develop a type and effect system for the G-Local π-calculus, which allows us to verify resource usages against LTL formulas.
- In Chapter 6, we conclude the thesis.

1.8 Origins of the Chapters

Part of the material presented in this thesis has been appeared in some publications or has been submitted for publication, in particular:

- The $\lambda^{\{\}}$-calculus and its abstract cloud semantics presented in Chapter 2 is introduced in [28, 27].
- The G-Local π-calculus and CFA presented Chapter 3 is introduced or has been submitted in [32, 30, 31].

Chapter 2

Background

In this chapter, we present concepts and notations that will be used through the text. In the first part, we introduce transition systems, over which standard class of models representing computing systems are built. We describe linear time properties of computing systems through linear time logics. Then, we briefly show how to verify linear time properties for finite transition systems. In particular, we discuss the model checking of basic parallel processes, a weak model of concurrency.

In the second part, we give a brief description of two formal models, λ- and π-calculus, which serve as a basis to develop the formal models in the next chapters. The λ-calculus is the foundational calculus for the sequential model, while the π-calculus is the foundational calculus for the concurrent model. They play an important role in computing: both of them give an elegant way to express various *computing functionalities* or *programs* that we encounter in computing. Many static analyses have been developed for λ-calculus [86] and π-calculus [26, 4]. Here, we will focus on Control Flow Analysis and Type Systems which are the two main formal tools we develop for our formal models in the next chapters.

2.1 Preliminaries

2.1.1 Transition Systems

In theoretical computer science, transition systems are often used as models to describe the behaviour of various computing systems. They consist of a set of states and transitions between states. A state *describes* some information about a system at a certain moment of its behaviour. For instance, the state of a computer program is a set of the current values of all program variables together with the program counter that indicates the next program statement to be executed. A transition *describes* how a system evolves from one state to another and possibly contains information about the transition itself (in such case we call it a labelled transition). In computer programs, a transition corresponds to the execution of a statement and may involve the modification of some variables and of the program counter.

Definition 2.1.1 (Labelled Transition Systems)**.** A labelled transition system (LTS) is a structure $(Q, A, \rightarrow, I)$, where Q is a set of states q, A is a set of actions (or labels), the relation $\rightarrow \subseteq Q \times A \times Q$ is called the transition relation and $I \subseteq Q$ is a set of initial states. We often write $q_1 \overset{\alpha}{\rightarrow} q_2$ for $(q_1, \alpha, q_2) \in \rightarrow$ (q_1 is called predecessor, while q_2 - successor). A labelled transition system $(Q, A, \rightarrow, I)$ is *finite* if Q and A are finite sets.

A path η in a given labelled transition system LTS is a finite of infinite sequence of actions and states such that

$$\eta = q_0 \overset{\alpha_1}{\rightarrow} q_1 \overset{\alpha_2}{\rightarrow} q_2 \dots,$$

where $q_i \overset{\alpha_{i+1}}{\rightarrow} q_{i+1}$ for all $i \geq 0$. A *run* is a maximal path, i.e. a path that is either infinite or terminated in a state without successors. We denote by $paths(q)$ the set of paths from q and $runs(q)$ for the set of runs from q, $paths(LTS) = \bigcup_{q \in I} paths(q)$ and $runs(LTS) = \bigcup_{q \in I} runs(q)$.

Example 2.1.2 (Mobile Reader)**.** Consider reading e-books from an online store on tablet devices. A user, when reading an e-book, may write some annotations. The way of using the online store depends on which kind of connections, low-bandwidth or high-bandwidth, a tablet device has. In the former case, the tablet needs to *load* an e-book from the store to local memory before any other actions and if users make annotations on the e-book, it requires to *store* them back on the online store. In the latter case, the user directly reads/writes e-books, however it is required that the user eventually releases the connection due to the high cost of the connection. We use rd, wr, ld and st to model operations of reading e-books, writing annotations, loading e-books from the online store to local memory and storing them back to the online store, respectively. The action rel denotes the operation of releasing the connection.

We assume two tablet devices. Their specifications are given by the labelled transition systems in Fig. 2.1. The figure on the left corresponds to the first device, while on the right - the second device. The set A of actions is $\{rd, wr, ld, st, rel\}$. The set of initial states is $I = \{l_1, h_1\}$. The first device always loads e-books to the local memory before any other actions, hence it satisfies the policy of the low-bandwidth connection. The second device intends to work with the high-bandwidth connection, since it guarantees read/write operations without loading e-books to its local memory. However, its infinite run without performing rel violates the policy of the high-bandwidth connection.

2.1.2 Automata and Languages

We define an automaton as a labelled transition system. Finite state automata are a class of automata, which is used in many different areas, including computer science,

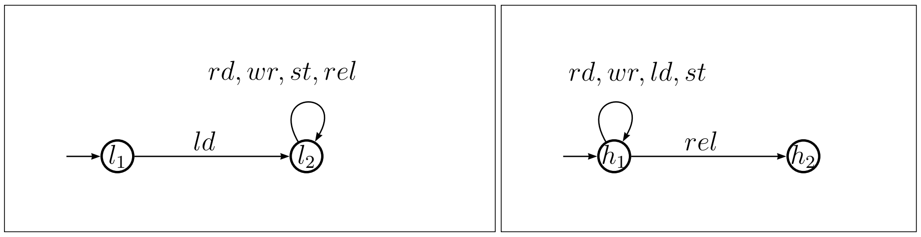

Figure 2.1: Transition systems of a mobile reader with a low-bandwidth (on the left) and a high-bandwidth (on the right) connection

mathematics and logics. In this text, we use them as a formalism to specify various properties of behaviour of computing systems (see below).

Definition 2.1.3 (Languages)**.** Given a set A of actions. A finite trace over A is a finite sequence $\alpha_1\alpha_2\ldots\alpha_n$, where $\alpha_i \in A, 1 \leq i \leq n$. A infinite trace over A is a infinite sequence $\alpha_1\alpha_2\alpha_3\ldots$, where $\alpha_i \in A, 1 \leq i$. We use η, η' to range over traces (both finite and infinite). A^* denotes a set of all finite traces over A. A^ω denotes a set of all infinite traces over A and $A^\infty = A^* \cup A^\omega$. A language over A is a subset of A^∞.

Given a labelled transition system $LTS = (Q, A, \rightarrow, I)$, each finite path of LTS

$$r = q_0 \overset{\alpha_1}{\rightarrow} q_1 \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_n}{\rightarrow} q_n,$$

corresponds to a finite trace $\eta = \alpha_1 \ldots \alpha_n$ (similarly for infinite executions). A trace $\alpha_1 \ldots \alpha_i$, where $i \leq n$, is called a *prefix* of η. $\alpha_1 \ldots \alpha_i$, where $i < n$, is called a *proper prefix* of η. We write $pref(\eta)$ for a set of all finite prefixes of η, i.e.

$$pref(\eta) = \{\hat{\eta} | \hat{\eta} \text{ is a finite prefix of } \eta\}$$

We use $Traces(q)$ to denote the set of traces generated by the executions starting from the state q of LTS and $Traces(LTS) = \bigcup_{q \in I} Traces(q)$.

Example 2.1.4. In the example of the mobile reader, *ld.rd.rd.wr* in the low-bandwidth connection or *rd.ld.wr.rel* in the high-bandwidth connection are possible traces.

Remark 2.1.5. By abuse of notation, we denote by η, η' both paths and traces (it will be clear from the context).

Definition 2.1.6 (Automata on finite traces)**.** A finite state automaton (FSA) $\mathcal{A}$ is a structure $(Q, A, \rightarrow, I, F)$, where $(Q, A, \rightarrow, I)$ is a finite labelled transition systems, where $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.

A run of a finite trace $\alpha_1\alpha_2\ldots\alpha_n \in A^*$ in $\mathcal{A}$ is a finite sequence of states $q_0q_1\ldots q_n$ such that

$$r = q_0 \overset{\alpha_1}{\rightarrow} q_1 \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_n}{\rightarrow} q_n$$

with $q_i \overset{\alpha_{i+1}}{\rightarrow} q_{i+1}$ for all $0 \leq i \leq n$ and $q_0 \in I$. A run $q_0 q_1 \ldots q_n$ is called *accepting* if $q_n \in F$. A finite trace $\alpha_1\alpha_2 \ldots \alpha_n \in A^*$ is called *accepted* if there is an accepting run for it. We denote by $L(\mathcal{A})$ the set of all accepted traces of $\mathcal{A}$ (sometimes called the language generated by $\mathcal{A}$).

Definition 2.1.7 (Regular Languages)**.** A language over A is called *regular* if it is generated by a finite state automaton.

Informally, an FSA can recognise a set of finite traces, but not a set of infinite traces. In case of infinite traces, we need a different formalism.

Definition 2.1.8 (ω-Languages)**.** Given a set of labels $\mathcal{A}$, an ω-language is a subset of $\mathcal{A}^\omega$.

Among ω-languages, the class of ω-regular languages enjoys many fundamental decision problems and has been successfully used in the specification and formal verification of computing systems [8].

Definition 2.1.9 (ω-Regular Languages)**.** For a regular language $L \subseteq A^*$, we define an ω-language L^ω as the set of all infinite concatenations of traces in L, i.e.

$$L^\omega = \{w_1 w_2 w_3 \ldots | w_i \in L, i \geq 1\}$$

An ω-language $L \in A^\infty$ is called ω-regular language if it has a form

- ${L_0}^\omega$, where L_0 is a regular language.
- $L_1 L_2$, the concatenation of a regular language L_1 and an ω-regular language L_2, i.e. $\{w_1 w_2 | w_1 \in L_1 \wedge w_2 \in L_2\}$.
- $L_1 \cup L_2$, the union of regular languages where L_1, L_2 are ω-languages.

Definition 2.1.10 (Automata on Infinite Traces)**.** A Buchi automaton $\mathcal{A}$ is a structure $(Q, A, \rightarrow, I, F)$, where $(Q, A, \rightarrow, I)$ is a finite labelled transition systems, where $I \subseteq Q$is the set of initial states and $F \subseteq Q$ is the set of final states.

A run of a infinite trace $\alpha_1\alpha_2\alpha_3 \cdots \in A^\omega$ in $\mathcal{A}$ is infinite sequence of states $q_0 q_1 q_2 \ldots$ such that

$$r = q_0 \overset{\alpha_1}{\rightarrow} q_1 \overset{\alpha_2}{\rightarrow} q_2 \overset{\alpha_3}{\rightarrow} q_3 \ldots$$

with $q_i \overset{\alpha_{i+1}}{\rightarrow} q_{i+1}$ for all $0 \leq i$ and $q_0 \in I$. A run $q_0 q_1 q_2 \ldots$ is called *accepting* if $q_i \in F$ for infinitely many indices $i \in \mathcal{N}$. A infinite trace $\alpha_1\alpha_2\alpha_3 \cdots \in A^*$ is called *accepted* if there is an accepting run for it. We denote by $L(\mathcal{A})$ the set of all accepted traces of $\mathcal{A}$ (sometimes called the language generated by $\mathcal{A}$).

A Buchi automaton is called *deterministic* if for each $q \in Q$ and $\alpha \in A$ there is at most one q' such that $q \overset{\alpha}{\rightarrow} q' \in \rightarrow$. Otherwise, it is called *non-deterministic.*

Lemma 2.1.11 (ω-languages and Non-deterministic Buchi Automata)**.** *The class of languages accepted by Non-deterministic Buchi Automata agrees with the class of ω-regular languages.*

2.1.3 Properties of Computing Systems

For verification purpose, one needs to check *some conditions* in a given state or across a number of states of the transition system model of the computing system under consideration. These conditions are usually referred as *properties* of the system. Properties about a single state are called *state-based*, whereas properties about several successive states are called *linear-time* or *path-based*.

Example 2.1.12. In the mobile reader example, it is desirable to have the property that requires that whenever the mobile reader takes a high-bandwidth connect, it eventually releases it.

Here we consider two important classes of linear-time properties: *regular safety properties* and *regular liveness properties*. What make them interesting is that these properties can be checked in an automated manner [8], using decidable decision problems in the field of automata. Following formulation in [8], we define linear time properties as follows.

Definition 2.1.13 (Linear Time Properties)**.** Given a set A of actions, a linear-time property P over A is a subset of A^ω.

Example 2.1.14. In the mobile reader example, a set of traces, which begin by an ld and are followed by infinite number of rd and wr, i.e. $ld.(\{rd, wr\})^\omega$ represents a possible linear time property.

Intuitively, a safety property asserts "that nothing bad will happen" during the evolution of the transition system. Here we consider regular safety properties, i.e., safety properties whose bad prefixes constitute a regular language, and hence this set can be represented by a finite state automaton.

Definition 2.1.15 (Safety Properties)**.** A linear-time property P_{safe} over A is a safety property if for all traces $\eta \in A^\omega \setminus P_{safe}$ there exists a finite prefix $\hat{\eta}$ of η such that

$$P_{safe} \cap \{\eta' \in A^\omega | \hat{\eta} \text{ is a finite prefix of } \eta'\} = \emptyset$$

The prefix $\hat{\eta}$ is called a *bad* prefix for P_{safe}. A bad prefix for P_{safe} is minimal if no proper prefix of $\hat{\eta}$ is a bad prefix for P_{safe}. A safe property P_{safe} is called *regular* if the set of all its bad prefixes is a regular language.

Remark 2.1.16. Notice that a bad prefix $\hat{\eta}$ of a linear-time property P_{safe} could be a prefix of many traces, which are not in P_{safe}.

Lemma 2.1.17 (Criterion for Regularity of Safety Properties)**.** *A safety property P_{safe} over $\mathcal{A}$ is called regular if its set of bad minimal prefixes constitutes a regular language over $\mathcal{A}$.*

Briefly, the algorithm to check a safety property P_{safe} for a given finite transition system relies on the reduction to the reachability problem in a product construction of the transition system with a finite automaton that recognises the bad prefixes of P_{safe}. The reachability problem of the resulted automaton is then analysed. If a final state is reachable, then the property does not hold in the transition system. (The interested readers are referred to [8] for more details).

A liveness property asserts that "something good eventually happens", and is used mainly to ensure progress. For instance, in the example of the mobile reader with the low-bandwidth connection the property that requires that the action *save* is eventually reached after an occurrence of *write*.

Definition 2.1.18 (Liveness Properties)**.** Given a set A of actions and a linear time property P over A, we write $pref(P)$ for $\bigcup_{\eta \in P} pref(\eta)$. A linear-time property P_{live} over A is a liveness property if $pref(P_{live}) = A^*$. A liveness property P_{live} is called *regular* if it is recognized by a Buchi automaton.

In general, liveness properties are harder to verify than safety properties for a given finite transition system. However, one can use the same strategy for checking regular safety properties to construct a product of the transition system with a Buchi automaton that recognizes $A^\omega \setminus P_{live}$, then *solve* its reachability problem to check whether a final state is reachable.

2.1.4 Temporal Logics

Temporal logics is a logical formalism suited for specifying linear time properties. The notion of time in temporal logics can be interpreted in either linear or branching view. In the linear view, at each moment there is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternative courses. We follow the formulation introduced in [35].

Linear Temporal Logic There are many classes of temporal logics based on a linear-time perspective. We consider only Linear Temporal Logic (LTL), a variant of temporal logics. LTL is a powerful temporal logic, first introduced by Pnueli in [94], able to express many interesting properties of computing systems.

Definition 2.1.19 (Linear Temporal Logic)**.** The LTL formulas over the set $\mathcal{A}$ of actions are defined by the following grammar:

$$\varphi ::= \text{true} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid (\alpha)\varphi \mid \varphi_1 \; U \; \varphi_2,$$

where $\alpha \in \mathcal{A}$.

LTL includes the standard logical operators: conjunction $\wedge$ and negation $\neg$. Note that the full power of propositional logic is obtained from them. LTL introduces two new modal operator: the *next* operator $(\alpha)\varphi$ and the *until* operator $\varphi_1 U \varphi_2$.

Intuitively, $(\alpha)\varphi$ holds at the current state on a path if φ holds at the next state, which is a result of performing an action α, on the path, whereas $\varphi_1 U \varphi_2$ holds on a path if there is a future state on the path for which φ_2 holds and φ_1 holds until that future state.

Convention 2.1.20. As usual, we write $\varphi_1 \vee \varphi_2$ for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2$ for $\neg\varphi_1 \vee \varphi_2$ and $\varphi_1 \equiv \varphi_2$ for $\varphi_1 \Rightarrow \varphi_2 \;\wedge\; \varphi_2 \Rightarrow \varphi_1$.

From the *until* operator we can derive two modal operators: F ("eventually", sometimes in the future) and G ("always", from now on forever). Formally, they are defined as follows

$$F\varphi \overset{def}{=} \mathit{true}\ U\varphi \quad G\varphi \overset{def}{=} \neg F\neg\varphi$$

Intuitively, $F\varphi$ ensures that φ will eventually be true in the future and $G\varphi$ holds if it is not the case that $\neg\varphi$ will eventually true in the future, that is φ always holds. Other derived modal operators are the following:

never φ:	$G\neg\varphi$
infinitely often φ:	$GF\varphi$
eventually forever φ:	$FG\varphi$
every "request" will eventually lead to a "response":	$G($ request $\Rightarrow F$ response $)$

The formula $G\neg\varphi$, where φ describes a *bad* behaviour, represents a safety property, while the others represent liveness properties. For instance, $G($ request $\Rightarrow$ F response $)$ says that it is always true that whenever a *request* arrives, a *response* eventually replies.

Taking the linear view, LTL formulas stand for properties of paths. This means that a path can either fulfil an LTL formula or not. Technically speaking, the interpretation of an LTL formula is defined in terms of maximal paths or runs.

Before going into the details, we need some notations. Given a labelled transition system LTS and a path $\eta = q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} q_3 \xrightarrow{\alpha_3} \dots$ of LTS, $\eta(1)$ denotes the first state of η, i.e., q_1, and η^1 denotes the path $q_2 \xrightarrow{\alpha_2} q_3 \xrightarrow{\alpha_3} \dots$. Similarly, $\eta(i)$ denotes the i-th state of η, i.e., q_i, and η^i denotes the path $q_i \xrightarrow{\alpha_i} q_{i+1} \xrightarrow{\alpha_{i+1}} \dots$. With these notations, the denotation $\|\varphi\|$ of a formula φ with respect to a labelled transition system $(Q, \mathcal{A}, \rightarrow, I)$ is $\|\varphi\| = \{\eta | \eta \models_{LTS} \varphi\}$, where the relation $\models_{LTS}$ is inductively defined by the following rules:

$$\begin{array}{l}
\eta \models \mathit{true} \\
\eta \models \neg\varphi \text{ if } \eta \models \varphi \text{ does not hold} \\
\eta \models \varphi_1 \wedge \varphi_2 \text{ if } \eta \models \varphi_1 \text{ and } \eta \models \varphi_2 \\
\eta \models (\alpha)\varphi \text{ if } \eta(1) \xrightarrow{\alpha} \eta(2) \text{ and } \eta^2 \models \varphi \\
\eta \models \varphi_1\ U\ \varphi_2 \text{ if } \exists i :\ \eta^i \models \varphi_2 \text{ and } \forall j \leq i :\ \eta^j \models \varphi_1
\end{array}$$

Remark 2.1.21. Sometimes we also interpret an LTL-formula as the set of traces corresponding to its set of paths.

Definition 2.1.22. Given a transition system LTS, a state s of LTS satisfies an LTL formula, denoted by $s \models \varphi$, if $Paths(s) \subseteq \|\varphi\|$ and LTS satisfies an LTL formula, denoted by $LTS \models \varphi$, if for all initial states s_0 of LTS, $s_0 \models \varphi$.

Automata-based LTL model checking. Now, we briefly describe a model-checking algorithm based on Buchi automata for LTL. Given a finite transition system LTS and an LTL formula φ that formalises a requirement on LTS, the problem is to check whether $LTS \models \varphi$. If it is refuted, an error trace needs to be provided for debugging purposes. The counterexample consists of an appropriate finite prefix of an infinite trace in TS where it does not hold.

The model checking algorithm presented in the following is based on the automata-based approach as originally suggested by Vardi and Wolper in [106]. This approach is based on the fact that each LTL formula φ can be represented by a non-deterministic Buchi automaton (NBA). The basic idea is to try to disprove $LTS \models \varphi$ by looking for a path η in LTS such that $\eta \models \neg\varphi$. If such a path is found, a prefix of η is returned as error trace. If no such trace is encountered, it is concluded that $LTS \models \varphi$.

The essential steps of the model-checking algorithm rely on the following observations:

$$\begin{aligned} LTS \models \varphi \quad & \text{iff } Paths(LTS) \subseteq \|\varphi\|_{LTS} \\ & \text{iff } Paths(LTS) \setminus \|\varphi\|_{LTS} = \emptyset \\ & \text{iff } Paths(LTS) \cap \|\neg\varphi\| = \emptyset \end{aligned}$$

Hence, for NBA A with $L_\omega(A) = \|\neg\varphi\|_{LTS}$, we have $LTS \models \varphi$ if and only if $Paths(LTS) \cap \|\neg\varphi\| = \emptyset$. Thus, to check whether φ holds for LTS one first constructs an NBA for the negation of the input formula φ (representing the *bad behaviours*) and then applies the techniques for the intersection problem (above).

Computation Tree Logic. Branching time refers to the fact that at each moment there may be several different possible futures. Thus, the interpretation of formulas in branching time logics is defined in terms of an infinite, directed tree of states rather than an infinite sequence. Each traversal of the tree starting in its root represents a single path. The tree itself thus represents all possible paths, and is directly obtained from a transition system by *unfolding* at the state of interest. The tree rooted at state q thus represents all possible runs in the transition system that start in q.

In this text, we consider Computation Tree Logic (CTL), an expressive variant of branching time logics. The syntax of CTL formulas is given by the following definition.

Definition 2.1.23 (Computer Tree Logic)**.** CTL formulas over the set A of actions are defined by the following grammar:

$$\varphi ::= \text{true} \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\alpha\rangle\varphi \mid E[\varphi_1\ U\ \varphi_2] | \ A[\varphi_1\ U\ \varphi_2],$$

where $\alpha \in A$.

In LTL, a formula holding in a state s requires that a formula φ holds in state s if all possible runs that start in s satisfy φ. Intuitively, we can state properties over all possible runs that start in a state, but not about some of such runs. CTL overcomes this by introducing the existential/universal quantification in the *next* and *until* operators, therefore CTL is able to specify such properties. More precisely, the prefixes E and A stand for the existential and universal quantification in their interpretations respectively.

The existential "eventually" EF and "always" EG operators can be derived from the *until* operator.

$$EF\varphi \overset{def}{=} E[true\ U\varphi]$$
$$EG\varphi \overset{def}{=} \neg A[true\ U\neg\varphi]$$

Note that we also have their universal versions, i.e. $[\alpha]\varphi \equiv \neg\langle\alpha\rangle\neg\varphi$, $AF\varphi \equiv \neg EF\neg\varphi$ and $AG\varphi \equiv \neg EG\neg\varphi$. The fragment of CTL containing the logical operators, the existential next operator and the "eventually" EF (the "always" EG operator, resp.) is called *the logic EF* (*the logic EG*, resp.). The combination of the logics EF and EG yields the more expressive logic, called *Unified System of Branching-Time Logic* [20] (called *the logic UB*). Let $LTS = (Q, A, \rightarrow, I)$ be a labelled transition system with a set I of initial states. The denotation of a formula is a subset of states of LTS defined by the following rules:

$$\begin{array}{ll}
\|true\|_{LTS} & = Q \\
\|\neg\varphi\|_{LTS} & = Q \setminus \|\varphi\|_{LTS} \\
\|\varphi_1 \wedge \varphi_2\|_{LTS} & = \|\varphi_1\|_{LTS} \cap \|\varphi_2\|_{LTS} \\
\|\langle\alpha\rangle\varphi\|_{LTS} & = \{q \in Q | \exists q' \in Q, (q, \alpha, q') \in \rightarrow\ \wedge q' \in \|\varphi\|_{LTS}\} \\
\|E[\varphi_1\ U\ \varphi_2]\|_{LTS} & = \{q \in Q | \exists \eta \in Paths(q) \text{ such that} \\
& \qquad \exists k \geq 0.(\forall 0 \leq j \leq k.\eta(j) \in \|\varphi_1\|_{LTS}) \wedge \eta(k) \in \|\varphi_2\|_{LTS}\} \\
\|A[\varphi_1\ U\ \varphi_2]\|_{LTS} & = \{q \in Q | \forall \eta \in Paths(q) \text{ such that} \\
& \qquad \exists k \geq 0.(\forall 0 \leq j \leq k.\eta(j) \in \|\varphi_1\|_{LTS}) \wedge \eta(k) \in \|\varphi_2\|_{LTS}\}
\end{array}$$

Definition 2.1.24. Given a labelled transition system $LTS = (Q, A, \rightarrow, I)$, we say that a state q of LTS satisfies a CTL formula φ, denoted by $q \models \varphi$, if $q \in \|\varphi\|_{LTS}$. We say that LTS satisfies a CTL formula φ, denoted by $LTS \models \varphi$, if $I \subseteq \|\varphi\|_{LTS}$

CTL model checking. The CTL model checking problem amounts to verifying for a given labelled transition system LTS and a CTL formula φ whether $LTS \models \varphi$. A basic algorithm of CTL model checking is rather simple: (i) compute recursively a set $\|\varphi\|$ of states satisfying φ; (ii) check whether $I \subseteq \|\varphi\|$. Alternatively, we can resort to the automata-based approach which is proposed in [73], which translates a CTL formula into an alternating tree automata and then reduces the problem to the non-emptiness problem of alternating tree automata.

$$
\begin{array}{llll}
\text{(Act)} & \pi.T \xrightarrow{\pi} T & \text{(Eq)} & \dfrac{T \xrightarrow{\pi} T'}{X \xrightarrow{\pi} T'} X = T \in \triangle \\
\text{(Choice}_1\text{)} & \dfrac{T_1 \xrightarrow{\pi} T_1'}{T_2 + T_1 \xrightarrow{\pi} T_2 + T_1'} & \text{(Choice}_2\text{)} & \dfrac{T_1 \xrightarrow{\pi} T_1'}{T_1 + T_2 \xrightarrow{\pi} T_1' + T_2} \\
\text{(Parallel}_1\text{)} & \dfrac{T_1 \xrightarrow{\pi} T_1'}{T_2 \parallel T_1 \xrightarrow{\pi} T_2 \parallel T_1'} & \text{(Parallel}_2\text{)} & \dfrac{T_1 \xrightarrow{\pi} T_1'}{T_1 \parallel T_2 \xrightarrow{\pi} T_1' \parallel T_2}
\end{array}
$$

Figure 2.2: The Operation Semantics of BPP processes.

2.1.5 Basic Parallel Processes

Basic Parallel Process (BPP) [45] is considered as one of the most simplest models of concurrency. It contains the prefix action ., the choice operator + and the merge operator $\parallel$.

Definition 2.1.25. We assume a set of process variables $\mathcal{PV}$, ranged over by X, Y, Z, a set A of actions, ranged over by α, and $\tau \notin A$ to denote an *internal* or *unobservable* action. BPP expressions are defined as follows

$$
\begin{array}{ll}
\text{(BBP expressions) } T, T' & ::= \mathbf{0} | X | \pi.T | T + T' | T \parallel T' \\
\text{(prefix actions) } \pi, \pi' & ::= \tau | \alpha
\end{array}
$$

The $\mathbf{0}$ is the empty process, i.e. the process that does nothing. The prefix action $\pi.T$ is a process that performs π and then behaves like T. The choice $T_1 + T_2$ represents a process that behaves either as T_1 or as T_2. The process $T_1 \parallel T_2$ is the parallel composition of T_1 and T_2 executing independently in parallel.

A BBP process is defined by the finite family $\triangle$ of recursive process equations:

$$\{X_i = T_i \| 1 \leq i \leq n\},$$

where the X_i are distinct and the T_i are BPP expressions at most containing the variables $\{X_1, \ldots, X_n\}$. The variable X_1 is singled out as the *leading variable* and $X_1 = T_1$ is called the *leading equation*. The set of BPP processes is denoted by $\mathcal{P}bpp$

A BPP expression T is called *guarded* if every variable occurrence in T is within the scope of action prefix sub-expression $\pi.T'$ of T.

A given finite family $\triangle$ of guarded BPP equations determines a labelled transition system. The operational semantics of BPP processes is defined through the least transition relation satisfying the rules in Fig. 2.2.

Decidability of model checking BPP processes against linear time logics. Despite of its simplicity, BPP lies at the border of decidability of many model checking problems [78]. In [54], it is showed that model checking BPP with most branching time logics is undecidable. This follows from the result that model checking BPP

BPP	**general**	**fixed formula**
reachability	NP-complete	$\in$ NP
EF	decidable, PSPACE-complete	$\in \sum_d^p$
EG	undecidable	undecidable
UB	undecidable	undecidable
CTL	undecidable	undecidable
alternation free modal μ calc	undecidable	undecidable
modal μ calc	undecidable	undecidable
LTL	decidable, EXPSPACE-hard	decidable
linear time μ calc	decidable, EXPSPACE-hard	decidable

Table 2.1: BPP decidability

with EG-fragment of CTL is undecidable. EF is the only decidable fragment of CTL for BPP. It has been showed that model checking BPP with EF-fragment is PSPACE complete. Model checking BPP for linear time logics is decidable and EXPSPACE hard. The table 2.1 shows the complexity of model checking BPP (taken from [54]).

2.2 The λ-Calculus

The λ-calculus is a formal language introduced by Church in the 1930s to investigate functions, function applications and recursion. In spite of its very simple syntax, the λ-calculus is strong enough to describe all mechanically computable functions.

2.2.1 Syntax

Definition 2.2.1. Given an infinite set of variables $\mathcal{V}$, ranged over by x, y, z. The set E of λ-expressions, ranged over by e, e', is defined by the following grammar:

$$\begin{array}{rlll} e, e' & ::= & & \textit{expressions} \\ & \mid & \mathbf{x} & \text{variable} \\ & \mid & \lambda x.\ e & \text{abstraction} \\ & \mid & (e_1\ e_2) & \text{application} \end{array}$$

The values v, v' of the calculus are variables and lambda abstractions. , i.e.

$$\begin{array}{rlll} v, v' & ::= & & \textit{values} \\ & \mid & \mathbf{x} & \text{variable} \\ & \mid & \lambda x.\ e & \text{abstraction} \end{array}$$

Definition 2.2.2. An occurrence of a variable x inside a term of the form $\lambda.x\ e$ is said to be *bound*. The corresponding λx is called a binder, and we say that the subterm e is the scope of the binder. A variable occurrence that is not bound is *free*. More generally, the set of free variables of a term e is denoted $\mathsf{fv}(e)$, and it is defined formally as follows

$$\begin{array}{ll} \mathsf{fv}(x) & = \{x\} \\ \mathsf{fv}(\lambda x.\ e) & = \mathsf{fv}(e) \setminus \{x\} \\ \mathsf{fv}((e_1\ e_2)) & = \mathsf{fv}(e_1) \cup FV(e_2) \end{array}$$

An expression e is called *closed* if $FV(e)$ is empty. We denote a set of closed λ-expressions as E_0.

Convention 2.2.3. We often write let $x = e_1$ in e_2 for $(\lambda x.\ e_2)\ e_1$, $\lambda\_\ e$ for $\lambda x.\ e$, where $x \notin fn(e)$ and $e_1; e_2$ for $(\lambda\_\ e_2)\ e_1$.

For any e, e' and x, *substitution* of e for x in e' is an operator for replacing every occurrence of x in e' by e with changing bound variables to avoid clashes. Formally, it is defined as follows.

Definition 2.2.4. A substitution of e for x in e', denoted by $e'\{e/x\}$, is defined as:

$$\begin{array}{lll} x\{e/x\} & = e & \\ (e_1\ e_2)\{e/x\} & = e_1\{e/x\}\ e_2\{e/x\} & \\ (\lambda x.\ e')\{e/x\} & = \lambda x.\ e' & \\ (\lambda y.\ e')\{e/x\} & = \lambda y.\ e' & \text{if } x \notin FV(e') \\ (\lambda y.\ e')\{e/x\} & = \lambda y.\ e'\{e/x\} & \text{if } x \in FV(e') \text{ and } y \notin FV(e) \\ (\lambda y.\ e')\{e/x\} & = \lambda z.\ e'\{z/y\}\{e/x\} & \text{if } x \in FV(e'),\ y \notin FV(e) \text{ and } z \notin FV(e) \cup FV(e') \end{array}$$

Definition 2.2.5. Let a λ-expression e contain an occurrence of $\lambda x.\ e'$, and let $y \notin e'$. We define α-conversion of e to e', denoted by $e \equiv_\alpha e'$, as a process that replaces a sub-expression $\lambda x\ e''$ of e by $\lambda y\ e''\{y/x\}$, where y does not occur at all in e.

Lemma 2.2.6. *α-conversion is an equivalence relation.*

Convention 2.2.7. From now on, unless stated otherwise, we identify λ-expressions up to α-conversion.

2.2.2 The operational semantics

The semantics of λ-expressions are defined through the concept of β-reduction, which captures the idea of function application. Formally, β-reduction is defined in terms of substitution.

Definition 2.2.8 (β-reduction)**.** Any expression of the form $(\lambda x.\ e)\ e'$ is called a β-redex and the corresponding expression $e\{e'/x\}$ is called reduct. We define a single

β-reduction to be the smallest relation $\rightarrow_\beta$ on λ-expressions satisfying:

$$\begin{array}{ll}
[\beta] & \lambda x.e\ e' \rightarrow_\beta e\{e'/x\} \\
[\text{cong}_1] & \dfrac{e_1 \rightarrow_\beta e_1'}{e_1\ e_2 \rightarrow_\beta e_1'\ e_2} \\
[\text{cong}_2] & \dfrac{e_2 \rightarrow_\beta e_2'}{e_1\ e_2 \rightarrow_\beta e_1\ e_2'}
\end{array}$$

Call-by-value semantics. There are several strategies of β-reduction, which are based on different order of evaluation. In this text, we consider only *call-by-value* strategy. Intuitively, call-by-value order is defined as "only outermost redexes are reduced", and "a redex is reduced only when its right-hand side has already been reduced to a value". For more details, we refer the reader to [92].

Definition 2.2.9. We define a call-by-value β-reduction to be the smallest relation $\rightarrow_{\beta cbv}$ on λ-expressions satisfying:

$$\begin{array}{ll}
[\text{cbv\_}\beta] & \lambda x.e\ e' \rightarrow_{\beta cbv} e\{e'/x\} \\
[\text{cbv\_cong}_1] & \dfrac{e_1 \rightarrow_{\beta cbv} e_1'}{e_1\ e_2 \rightarrow_{\beta cbv} e_1'\ e_2} \\
[\text{cbv\_cong}_2] & \dfrac{e_2 \rightarrow_{\beta cbv} e_2'}{v\ e_2 \rightarrow_{\beta cbv} v\ e_2'}
\end{array}$$

And, $e \rightarrow_{\beta cbv} e'$ if and only if e' is obtained from e by a single β-reduction. We write $\rightarrow$ for $\rightarrow_{\beta cbv}$, unless stated otherwise. Moreover, $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$.

Notice that in the rule [*cbv_cong*$_2$], the first term of application in the conclusion is always a value.

Definition 2.2.10. Applicative Bisimulation A binary relation R on E_0 is a *simulation* if whenever $e\ R\ d$ and $e \rightarrow^* \lambda x.e'$, then there exists d' such that $d \rightarrow^* \lambda x.d'$, $\lambda x.e'\ R\ \lambda x.d'$ and for any value v, $e'\{v/x\}\ R\ d'\{v/x\}$.

We write $\lesssim$ for the union of all simulations and call it similarity. A binary relation R on T_0 is a bisimulation if R and its conversion R^{-1}, i.e. $R^{-1} = \{(e_2, e_1) | (e_1, e_2) \in R\}$, are simulations. We write $\sim$ for the union of all bisimulations and call it bismilarity.

Definition 2.2.11. An equivalent relation R on T_0 is a congruence if it is preserved by the operations and substitutions.

In [61], Howe showed that bisimilarity is a congruence.

Theorem 2.2.12. *Bisimilarity $\sim$ is a congruence relation.*

2.2.3 Control Flow Analysis

Since introduced in the sixties [96], Control Flow Analysis (CFA) has proved its importance in the cycle of development of software. Basically, it provides a framework to compute which values or information can reach certain program points or can be assigned to a specific variable. Control Flow Analysis comes into many different formulations, as stated in [86], such as constraint-based, abstract interpretation-based and specification-based, and type-based. The simplest form of Control Flow Analysis is so-called 0-CFA, which does not take the context information into account when analysing programs. Intuitively, it does not distinguish instances of function calls, more precisely, different instances of program points and variables of the function at various call sites.

In this section, we consider specification-based approach to 0-CFA for a simply typed λ-calculus, based on [86]. To consider this approach, the syntax of the λ-calculus is extended with recursive variable, which are bound to function bodies.

Definition 2.2.13. Given an infinite set of variables $\mathcal{V}$, ranged over by x, y, z. The set E of λ-expressions, ranged over by e, e', is defined by the following grammar:

$$\begin{array}{rll} e, e' ::= & & \textit{expressions} \\ | & \mathbf{x} & \text{variable} \\ | & \lambda_z x.\ e & \text{abstraction} \\ | & (e_1\ e_2) & \text{application} \end{array}$$

The values v, v' of the calculus are variables and lambda abstractions. , i.e.

$$\begin{array}{rll} v, v' ::= & & \textit{values} \\ | & \mathbf{x} & \text{variable} \\ | & \lambda_z x.\ e & \text{abstraction} \end{array}$$

The set of free names and notion of substitution are defined as expected. The operational semantics of extended λ-expressions is similar to the one defined in the previous section, except that the rule [cbv_β] is defined as follows:

$$[\text{cbv\_}\beta] \quad \lambda_z x.e\ e' \rightarrow_{\beta cbv} e[\lambda_z x.e/z, e'/x],$$

where $e[\lambda_z x.e/z, e'/x]$ is a *simultaneous* substitution. The idea is that the recursive variable is replaced by the function body $\lambda_z x.e$ when reduction is applied.

Also, given λ-expression e, assume that all program points (i.e all sub-expressions of e) are labelled and, for simplicity, all labels are just integers. We denote the set of all labels by *Lab*, ranged over by l and the set of extended λ-expressions by E^*, ranged over by t. Formally, we define the following abstract syntax:

$$\begin{array}{rl} l \in \mathit{Lab} & \text{labels} \\ t ::= e^l,\ \text{where } e \in E & \text{terms} \end{array}$$

The result of 0-CFA analysis is a pair of (C, ρ) where C is a mapping from program points to a set of values, i.e. $C(l)$ contains the values, i.e. variables or lambda abstractions, that can reach the program point l and $\rho(x)$ is mapping from variables to a set of values, i.e. $\rho(x)$ contains the values that the variable x can be bound to. Formally, C is a mapping from labels to sets of values and ρ is a mapping from variables to sets of values.

Example 2.2.14. As a running example, we use the following λ-expression:

$$e = let\ g\ = (\lambda_f x.\ (f^1\ (\lambda y.\ y^2)^3)^4)^5\ in\ (g^6\ (\lambda z.\ z^7)^8)^9$$

The idea behind Control Flow Analysis is that based on control flow it defines rules for transferring all possible information from one program point to another. Usually, this set of rules forms a specification of analysis. The specification is defined in Tab. 2.2. To specify Control Flow Analysis, we firstly use the specification to generate a system of constraints and then find a solution of the system. It leads to the following definition.

Definition 2.2.15. A pair of (C, ρ) is an acceptable 0-CFA analysis of a λ-expression e if it satisfies the system of constraints generated by specification when analysing e, where constraints are defined as follows:

- $C(l) \subseteq \rho(x)$, that is, a set of values that reach the program point labelled by l is included in the set of values that are possibly bound to the variable x.
- $C(l) \subseteq C(l')$, that is, a set of values that reach the program point labelled by l is included in the set of values that reach the program point labelled by l'.
- $S \subseteq \rho(x)$, that is, a set S of values is included in the set of values that are possibly bound to the variable x.

$$\begin{array}{llll}
(abs) & (C,\rho) & \models & (\lambda_z x.\ e)^l \text{ iff } \{\lambda_z x.\ e\} \subseteq C(l) \wedge (C,\rho) \models e \\
(app) & (C,\rho) & \models & ((e_1)^{l_1}\ (e_2)^{l_2})^l \text{ iff } (C,\rho) \models (e_1)^{l_1} \wedge (C,\rho) \models (e_2)^{l_2} \\
 & & & \wedge\ (\forall (\lambda x.\ {e_0}^{l_0}) \in C(l_1)) : (C,\rho) \models {e_0}^{l_0}\ \wedge\ C(l_2) \subseteq \rho(x)\ \wedge\ C(l_0) \subseteq C(l) \\
 & & & \wedge\ (\forall (\lambda_z x.\ {e_0}^{l_0}) \in C(l_1)) : (C,\rho) \models {e_0}^{l_0}\ \wedge\ C(l_2) \subseteq \rho(x)\ \wedge\ C(l_0) \subseteq C(l) \\
 & & & \qquad\qquad\qquad\qquad \wedge\ \{\lambda_z x.\ {e_0}^{l_0}\} \subseteq \rho(z)
\end{array}$$

Table 2.2: Specification of 0-CFA

In the rule (abs), the analysis holds for the sub-expression e and the abstraction $\lambda_z x\ e$ is included in a set of values reaching the label l. The rule (app) not only analyse the sub-expressions e_1 and e_2, but also generates a set of constraints for each reachable abstraction-value $\lambda_z x.{t_0}^{l_0}$ at l_1, that is: i) a set of values reaching the label l_2 can be bound to x; ii) a set of values reaching the label l contains a set of values reaching the label l_0; iii) in the case of *recursive* abstraction-values, the value is bound to its recursive variable.

Example 2.2.16. We apply the CFA to our example. The system of constraints is generated as follows:

$$\begin{array}{ll}
(1) & (\forall(\lambda x.\ {e_0}^{l_0}) \in C(5)) : (C,\rho) \models {e_0}^{l_0} \ \wedge\ C(8) \subseteq \rho(x) \ \wedge\ C(l_0) \subseteq C(9) \\
(2) & (\forall(\lambda_z x.\ {e_0}^{l_0}) \in C(5)) : (C,\rho) \models {e_0}^{l_0} \ \wedge\ C(8) \subseteq \rho(x) \\
 & \qquad\qquad\qquad\qquad\qquad \wedge\ C(l_0) \subseteq C(9) \ \wedge\ \{\lambda_z x.\ {e_0}^{l_0}\} \\
(3) & (\forall(\lambda x.\ {e_0}^{l_0}) \in C(1)) : (C,\rho) \models {e_0}^{l_0} \ \wedge\ C(3) \subseteq \rho(x) \ \wedge\ C(l_0) \subseteq C(4) \\
(4) & (\forall(\lambda_z x.\ {e_0}^{l_0}) \in C(1)) : (C,\rho) \models {e_0}^{l_0} \ \wedge\ C(3) \subseteq \rho(x) \\
 & \qquad\qquad\qquad\qquad\qquad \wedge\ C(l_0) \subseteq C(4) \ \wedge\ \{\lambda_z x.\ {e_0}^{l_0}\} \\
(5) & \{f\} \subseteq C(5) \\
(6) & \rho(f) \subseteq C(1) \\
(7) & \{id_y\} \subseteq C(3) \\
(8) & \rho(y) \subseteq C(2) \\
(9) & \{id_x\} \subseteq C(8) \\
(10) & \rho(z) \subseteq C(7)
\end{array}$$

where $f = \lambda_f x.\ (f^1\ (\lambda y.\ y^2)^3)^4$, $id_y = \lambda y.\ y^2$ and $id_z = \lambda z.\ z^7$. In this system, the first four constraints are *conditional or implicit constraints*, which are generated when analysing the λ-expressions at the labels 9 and 5. More precisely, in (1) and (2), for each reachable abstraction-value $\lambda_z x.\ {e_0}^{l_0}$ at the label 5: i) all abstraction-values reaching at the label 8 are included in the set of values bound to x; ii) all abstraction-values reaching at the label l_0 are also reached at the label 9. iii) all *recursive* abstraction-values are bound to its recursive variable. Similar arguments can made for clauses (3) and (4). The remaining constraints from (4)-(18) are called *explicit* constraints, which relate values to their labels.

One of solutions of the above systems is the following:

$$\begin{array}{lllllllll}
C(1) & = & \{f\} & C(2) & = & \emptyset & \rho(f) & = & \{f\} \\
C(3) & = & \{id_y\} & C(4) & = & \emptyset & \rho(g) & = & \{f\} \\
C(5) & = & \{f\} & C(6) & = & \{f\} & \rho(x) & = & \{id_y, id_z\} \\
C(7) & = & \emptyset & C(8) & = & \{id_z\} & \rho(y) & = & \emptyset \\
C(9) & = & \emptyset & C(10) & = & \emptyset & \rho(z) & = & \emptyset
\end{array}$$

Theoretical properties. The proof of existence of acceptable 0-CFA analysis is given in literature, see [86] for details. Moreover, the solution is not unique. For example, by taking the above solution and adding more information to $C(5), C(6)$

as follows:

$$\begin{array}{rcl} C'(5) & = & \{f, id_y\} \\ C'(6) & = & \{f, id_z\} \\ C'(i) & = & C(i), \quad \text{where } i \neq 5, 6 \end{array}$$

We have another acceptable analysis (C', ρ), but less precise. Theoretically, a set of acceptable analyses for a specific term enjoys the *model intersection property*, i.e. whenever we take the intersection of a number of acceptable analyses we still get an acceptable analysis. As a consequence, we can obtain the *least* (most precise) solution from the set of acceptable analyses for the term.

Definition 2.2.17. The set of acceptable analyses can be partially ordered by setting $(\rho, C) \sqsubseteq (\rho', C')$ if and only if $\forall l \in Lab, C(l) \subseteq C'(l)$ and $\forall x \in \mathcal{V}, \rho(x) \subseteq \rho'(x)$. We write $(\rho, C) \sqcup (\rho', C')$ ($(\rho, C) \sqcap (\rho', C')$, resp.) for the binary least upper bound (the binary greatest lower bound, resp.) (defined point-wise). We use $\sqcup I$ ($\sqcap$)to denote the least upper bound (the greatest lower bound, resp.) for a set I of acceptable analyses.

The important feature of a Moore family set is that it enjoys the model intersection property.

Definition 2.2.18. The set of estimates is a Moore family if and only if it contains $\sqcap J$ for all $J \subseteq I$.

Theorem 2.2.19 (Existence of the least solution)**.** *Given a λ-expression e, a set of acceptable analyses of e, i.e. $\{(C, \rho) | (C, \rho) \models e\}$, is a Moore family.*

More importantly, the least acceptable analysis is correct with respect to the semantics, i.e. it ensures that the information from the analysis is indeed a safe description of what will happen during the execution of the program.

Theorem 2.2.20 (Subject Reduction)**.** *Given a λ-expression e, $(C, \rho) \models e$. If $e \to e'$, then $(C, \rho) \models e'$.*

Remark 2.2.21. We refer readers to [86] for details of he algorithm of computing the least solution of a system of constraints.

2.2.4 The $\lambda^{\Box}$-calculus

We present an extension of the simply typed lambda calculus, called lambda box ($\lambda^{\Box}$), with possible expressing resources and events. This extension is introduced in [11] to formalise a model of *history-based* access control. The idea behind it is to abstract the entire execution by means of sequences of events, called histories, and to specify access control for running code based on these histories generated by the code so far. By considering the entire execution, it improves the idea of e.g. *stack inspection* [57], which instead records a fragment of the whole execution, hence provides a better understanding of the behaviour of programs. Indeed, the previous analysis originally comes from the idea of history-based access control. It is simple to approximate all the possible runtime histories of the programs and then use some model checking to check whether the resulting approximations satisfy some policy based on the sequence of events.

The Syntax

Definition 2.2.22. (**Local Policies**) Let A be a set of actions, a policy φ is a regular safety property over A.

Definition 2.2.23. Let $\mathcal{V}$ be an infinite set of variables, ranged over by x, y, z, $\mathcal{R}$ be a set of resources, ranged over by r, r', and A be a set of monadic actions, ranged over by α, β. A set Ev of access events is defined as $\{\alpha(r) | \alpha \in A \text{ and } r \in \mathcal{R}\}$. We assume a set Φ of local policies, ranged over by φ, φ'. The set $E^{\square}$ of $\lambda^{\square}$-expressions, ranged over by e, e', is defined by the following grammar:

$$\begin{array}{rlll}
e, e' & ::= & & \textit{expressions} \\
 & | & \mathbf{x} & \text{variable} \\
 & | & \alpha(r) & \text{access event} \\
 & | & \text{if } b \text{ then } e_1 \text{ else } e_2 & \text{conditional} \\
 & | & \lambda_z\, x.\, e & \text{abstraction} \\
 & | & (e_1\ e_2) & \text{application} \\
 & | & \varphi[e] & \text{policy framing,}
\end{array}$$

The values v, v' of the calculus are variables and lambda abstractions, i.e.

$$\begin{array}{rlll}
v, v' & ::= & & \textit{values} \\
 & | & \mathbf{x} & \text{variable} \\
 & | & \lambda_z\, x.\, e & \text{abstraction}
\end{array}$$

We write $\mathbf{0}$ for a fixed, closed and event-free value. The definition of guards b in conditionals is irrelevant here, and so it is omitted.

The language $\lambda^{\square}$ extends the λ-calculus by adding two new language constructs. The first one is that of *access events* $\alpha(r)$, which describes the application of the action α on the target resource r at runtime. A finite sequence of access events is called a *history*. The second one is that of *policy framing* $\varphi[e]$ which indicates that the evaluation of e must obey the policy φ.

Usage Automata. Security policies $\varphi \in \Phi$ are modelled as regular safety properties of histories, i.e. properties whose set of bad prefixes are recognized by an automaton, as discussed above. A *policy framing* $\varphi[e]$ enforces regular property of histories during the execution of e. The policy φ can be represented by the set of its *bad* prefix, which in turn is recognized by usage automata introduced in [10]. Here

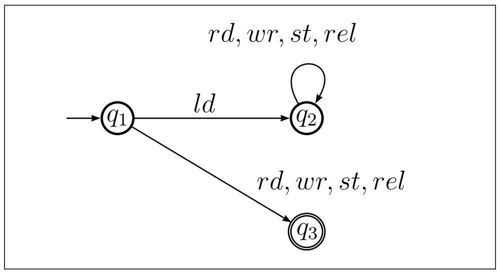

Figure 2.3: The usage automata for the policy of the low-bandwidth connection

we consider a simplified version of usage automata, where usage automata do not contain resource variables. We refers readers to [10] for more details. Informally, a usage automaton describes a policy defined by a regular safety property. Usage automata are considered as an extension of automata, where its final states are *offending*, entering into which is considered as a *policy violation*. We write $\eta \models \varphi$ if the history η does not lead to offending states in the corresponding usage automaton.

Example 2.2.24. Back to our reader mobile example, it could be required that for the low-bandwidth connection the tablets needs to load an e-book to local memory before any other actions. The usage automata given in Fig. 2.3 describes this policy. The set Q of states is $\{q_1, q_2, q_3\}$ and the set A of actions is $\{rd, wr, ld, st, rel\}$. The only final state q_3 is marked by a double circle, which indicates the *offending* state. At the beginning, from the initial state q_1 any action, except for ld, leads to the offending state q_3, which therefore describes the required policy.

The Operational Semantics. The behaviour of $\lambda^{\square}$-expressions, described in Fig. 2.4, is defined through a structural operational semantics. A transition $(\eta, e) \xrightarrow{\mu} (\eta', e')$, indicates that, starting from a state described by the history η, the expression e evolves to e', issuing an event μ, possibly extending the history to η'. Initial configurations have the form (ϵ, e), where ϵ denotes the empty history.

$$[\text{s\_event}] \quad \eta, \alpha(r) \to \eta\alpha(r), \mathbf{0} \qquad [\text{s\_app}_0] \quad \eta, (\lambda_z x.e)\ v \ \to\ \eta, e[\lambda_z x.e/z, v/x]$$

$$[\text{s\_cond}_0] \quad \frac{\mathcal{B}(b) = true}{\eta, \text{if } b \text{ then } e_1 \text{ else } e_2 \to \eta, e_1} \qquad [\text{s\_cond}_1] \quad \frac{\mathcal{B}(b) = false}{\eta, \text{if } b \text{ then } e_1 \text{ else } e_2 \to \eta, e_2}$$

$$[\text{s\_app}_1] \quad \frac{\eta, e_1 \to \eta, e_1'}{\eta, e_1\ e_2 \to \eta, e_1'\ e_2} \qquad [\text{s\_app}_2] \quad \frac{\eta, e_2 \to \eta, e_2'}{\eta, v\ e_2 \to \eta, v\ e_2'}$$

$$[\text{s\_pol}_0] \quad \frac{\eta, e \to \eta\alpha(r), e' \quad \eta\alpha(r) \models \varphi}{\eta, \varphi[e] \to \eta\alpha(r), \varphi[e']} \qquad [\text{s\_pol}_1] \quad \frac{\eta \models \varphi}{\eta, \varphi[v] \to \eta, v}$$

Figure 2.4: The Operational Semantics of the $\lambda^{\square}$

We now comment on the operational rules. The rule [s_event] describes the evaluation of an event $\alpha(r)$ that consists in extending the current history with the event itself, and producing the empty value $\mathbf{0}$. The rules [s_app$_0$], [s_app$_1$] and [s_app$_2$] are the standard rules of the call-by-value semantics of λ-calculus. Notice that the whole function body $\lambda_z x.e$ replaces the self variable z after the parameter substitution, so giving an explicit copy-rule semantics for recursive functions. We assume that $\mathcal{B}$ is a total function to evaluate guards b in conditionals. In rules [s_cond$_0$] and [s_cond$_1$], depending on the evaluation of the guard b, a choice among two branches is made. The policy framing $\varphi[e]$ enforces the policy φ on the expression e, meaning that the history must respect φ at each step of the evaluation of e and each event issued within e must be checked against φ as in [s_pol$_0$]. When e is just a value, the security policy is simply removed as in [s_pol$_1$].

2.2.5 Type system

Type systems are a formal tool for reasoning about programs. By associating a *type* to each computed value, it provides a tractable syntactic method for proving the absence of certain program behaviours. For instance, by examining the flow of values, therefore their types, it ensures that no boolean variable is assigned to an integer value. One of the possible approaches to type-based analysis is *type and effect systems*, which is considered below.

A simple type system. To better understand the type and effect system introduced in [11], we present a simplified type system without the effects that will be added in the second part of the section.

Definition 2.2.25. We use $\mathbf{1}$ to denote a type for the empty value $\mathbf{0}$. We use the following grammar to define types:

$$T \ ::= \mathbf{1} \mid T \to T$$

Furthermore, we assume that each access event $\alpha(r)$ has functional type of the form $\mathbf{1}$.

A type environment Γ is a finite mapping from variables to types. We write $\Gamma, x \to T$ when $x \notin \mathit{dom}(\Gamma)$, to denote the environment that maps the variable x to the type T, and the variables $y \neq x$ to $\Gamma(y)$, whenever $y \in \mathit{dom}(\Gamma)$. Furthermore, we write $\Gamma \models x : T$ if $x \in \mathit{dom}(\Gamma)$ and $\Gamma(x) = T$. Formally, Γ is given as below:

$$\Gamma \ ::= \epsilon \mid \Gamma, x \to T,$$

where ϵ denotes the empty type environment. The general form of a typing rule is given by:

$$\Gamma \vdash e : T$$

$$
\begin{array}{llll}
[\text{t\_empty}] & \Gamma \vdash \mathbf{0} : \mathbf{1} & [\text{t\_var}] & \dfrac{\Gamma \models x : T}{\Gamma \vdash x : T} \\[2ex]
[\text{t\_evt}] & \Gamma \vdash \alpha(r) : 1 & [\text{t\_abs}] & \dfrac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2} \\[2ex]
[\text{t\_app}] & \dfrac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2} & [\text{t\_cond}] & \dfrac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : T} \\[2ex]
[\text{t\_pol}] & \dfrac{\Gamma \vdash e : T}{\Gamma \vdash \varphi[e] : T} & &
\end{array}
$$

Figure 2.5: Typing rules

that says the program e has type T assuming that any free variable in e has a type given by Γ. The axioms and typing rules are defined in Fig. 2.5.

The rule [t_empty] associates the type $\mathbf{1}$ to the empty value $\mathbf{0}$. The rule [t_var] states that a variable has whichever type it is declared to have in the typing environment Γ. In the rule [t_evt], event accesses over resources are assumed to have the type $\mathbf{1}$. The rule [t_abs] states that an abstraction $\lambda_z x.e$ has a function type $T_1 \rightarrow T_2$, provided that e can be typed as T_2 in an environment where x has type T_1. The rule [t_app] associates a type T_2 to an application $e_1\ e_2$ whenever the argument e_1 has a function type $T_1 \rightarrow T_2$, and the argument e_2 has type T_1. In the rule [t_cond], the conditional expression has the same type as its branches. Finally, in the rule [t_pol], the policy framing $\varphi[e]$ has the same type as e.

Type and effect system. In type and effect systems, by annotating types with effects which describe which side effects a program may have, we could reason about and control the overall computational effect of the program.

To see the idea behind effect systems, we present the type and effect system introduced in [11] to analyse the behaviour of the program abstracted by sequences of access events resulting from the executions. To predict the histories generated by programs at runtime, history expressions are introduced. The grammar of history expressions is defined as follows:

$$H \ ::= \epsilon \mid h \mid \alpha(r) \mid H.H' \mid H + H' \mid \varphi[H] \mid \mu h.H$$

The history expressions include the empty history ϵ, events $\alpha(r)$, sequential composition $H.H'$, non-deterministic choice $H + H'$, policy framing $\varphi[H]$ and $\mu h.H$ recursion, in which μ binds occurrences of the recursive history variable h in H. Free variables and closed expressions are defined as expected. Furthermore, to explicitly represent policy framing events, special symbols $[_\varphi$ and $]_\varphi$ are used to denote opening and closing of the scope of the policy φ. We write Ev' for $Ev \cup \{[_\varphi,]_\varphi | \varphi \in \Phi\}$ and $\eta^\flat$ for the subsequence of η containing only events in Ev.

Let $\mathcal{H}$ range over the histories. $\mathcal{H}.\mathcal{H}'$ denotes the set of histories $\{\eta\eta' | \eta \in \mathcal{H}, \eta' \in \mathcal{H}'\}$, and $\varphi[\mathcal{H}]$ is the set $\{[_\varphi \eta]_\varphi\}$. The set $\mathsf{fv}()$ of free history variables of a history expression H is defined as follows:

$$\begin{array}{ll}
\mathsf{fv}(\epsilon) = \emptyset & \mathsf{fv}(h) = \{h\} \\
\mathsf{fv}(\alpha(r)) = \emptyset & \mathsf{fv}(H.H') = \mathsf{fv}(H) \cup \mathsf{fv}(H') \\
\mathsf{fv}(H + H') = \mathsf{fv}(H.H') = \mathsf{fv}(H) \cup \mathsf{fv}(H') & \mathsf{fv}(\varphi[H]) = \mathsf{fv}(H) \\
\mathsf{fv}(\mu h.H) = \mathsf{fv}(H) \setminus \{h\} &
\end{array}$$

A history expression is closed if it has no free history variables.

The denotational semantics of history expressions is defined over the complete lattice $(2^{Ev'}, \subseteq)$, where $2^{Ev'}$ denotes the set of all subsets of Ev'. The environment ρ used below maps variables to sets of (finite) histories. The denotation of history expressions is defined as follows:

$$\begin{array}{l}
[\![\epsilon]\!]_\rho = \{\epsilon\} \quad [\![\alpha(r)]\!]_\rho = \{\alpha(r)\} \quad [\![h]\!]_\rho = \rho(h) \\
[\![H.H']\!]_\rho = [\![H]\!]_\rho.[\![H']\!]_\rho \quad [\![H + H']\!]_\rho = [\![H]\!]_\rho \cup [\![H']\!]_\rho \quad [\![\varphi[H]]\!]_\rho = \varphi[[\![H]\!]_\rho] \\
[\![\mu h.H]\!]_\rho = \bigcup_{n \in \mathcal{N}} f^n(\emptyset),
\end{array}$$

where $\mathcal{N}$ is the set of natural numbers, $f(X) = [\![H]\!]_{\rho\{X/h\}}$, and $f(0) = \emptyset$,otherwise $f^n(X) = f^{n-1}(f(X))$. Furthermore, the relation $\sqsubseteq$ is used to denote sub-effects (by abuse of notation). Roughly, $H \sqsubseteq H'$ means that a set of histories represented by H are included in a set of histories represented by H'.

In the annotated types given below, the functional type $T_1 \xrightarrow{H} T_2$, is annotated with an effect H that describes the *latent* effect associated with an abstraction, i.e. one of the histories represented by H could be generated when such an abstraction is applied to a value. Formally, annotated types are defined by the following syntax:

$$T \ ::= \mathbf{1} \mid T \xrightarrow{H} T$$

The general form of type judgement has the following form

$$\Gamma \vdash e : T \rhd H$$

that says that the program e, which has type T assuming that any free variable in e has a type given by Γ, produces a history represented by H. The obtained type and effect system is defined as follows:

$$\begin{array}{llll}
[\text{te\_var}] & \dfrac{\Gamma(x) = T}{\Gamma \vdash x : T \rhd \epsilon} & & \\[2ex]
[\text{te\_evt}] & \vdash \alpha(r) : \mathbf{1} \rhd \alpha(r) & [\text{te\_abs}] & \dfrac{\Gamma, x : T_1 \vdash e : T_2 \rhd H}{\Gamma \vdash \lambda x.e : T_1 \xrightarrow{H} T_2 \rhd \epsilon} \\[2ex]
[\text{te\_app}] & \dfrac{\Gamma \vdash e_1 : T_1 \xrightarrow{H} T_2 \rhd H' \quad \Gamma \vdash e_2 : T_1 \rhd H''}{\Gamma \vdash e_1\ e_2 : T_2 \rhd H'.H.H''} & [\text{te\_pol}] & \dfrac{\Gamma \vdash e : T \rhd H}{\Gamma \vdash \varphi[e] : T \rhd \varphi[H]} \\[2ex]
[\text{te\_cond}] & \dfrac{\Gamma \vdash e_1 : T \rhd H_1 \qquad \Gamma \vdash e_2 : T \rhd H_2}{\Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : T \rhd H_1 + H_2} & [\text{te\_sub}] & \dfrac{\Gamma \vdash e : T \rhd H \quad H \sqsubseteq H'}{\Gamma \vdash e : T \rhd H'}
\end{array}$$

The rule [te_evt] records the event $\alpha(r)$ in the history component. In the rule [te_abs], the type of an abstraction $\lambda_z x.e$ is annotated with a history H, provided that the expression e has H as an effect when type checking. The rule [te_app] concatenates three histories in the following order:

i) the history H' generated when typing e_1;

ii) the history H that is a latent effect due to the type $T_1 \xrightarrow{H} T_2$ of e_1;

iii) the history H'' generated when typing e_2. The rule [te_cond] describes the non-determination between two effects H_1 and H_2 from two branches of the conditional.

In the rule [te_pol], the effect H of e is put in a policy framing. Finally, the rule [te_sub] describes a *weakening* of the effect in the sense that the history H can be enlarged.

Type Safety. Based a computational effect H, one can verify regular properties of resource usages, exhibited by a program, by using model checking techniques. The basic idea is to translate H into BPA processes, which are basically BPP processes without the parallel operator, then apply decidable model checking techniques of BPA processes, as described in [53]. We refer readers to [16]. We will omit all technical details and state only the main property of the type and effect system.

Definition 2.2.26 (Validity of History Expressions)**.** A history expression H under ρ is valid for a policy φ if for all $\eta \in [\![H]\!]_\rho$, $\eta^\flat$ satisfies φ.

Theorem 2.2.27 (Type Safety)**.** *If $\Gamma \vdash e : T, H$, with e is closed and H under the empty environment ρ is valid for all policies in e, then e can not go wrong.*

2.3 Calculus of Communicating Systems.

Calculus of Communicating Systems (CCS) is process calculus introduced by Milner in [81]. CCS processes can synchronise on their actions, rename and hide their actions as well. In this text, we consider a variant of CCS with guarded replication. The idea of CCS over BPP is able to capture a notion of communication. Intuitively, an action in CCS can synchronize with its *co-action.*

Definition 2.3.1. We assume a set A of names, ranged over by a, b, c and a set $\bar{A}$ of co-names , ranged over by $\bar{a}$ and an internal or unobservable action, denoted by τ. CCS processes are defined as follows:

$$\begin{array}{lll} \text{(CCS processes) } P, P' & ::= \mathbf{0} \mid \pi.P \mid P + P' \mid P \parallel P' \mid P \setminus a \mid P[b/a] \mid !\pi.P \\ \text{(prefix actions) } \pi, \pi' & ::= \tau | a | \overline{a} \end{array}$$

The $\mathbf{0}$ is the empty process, i.e. a process that does nothing. The prefix action $\pi.P$ is a process that can perform π, then behaves like P. The choice $P_1 + P_2$ represents a process that behaves either as P_1 or as P_2. The operator $\parallel$ denotes the parallel composition of processes $P_1 \parallel P_2$. The restriction $P \setminus a$ hides the action a in P. The relabelling $P[b/a]$ represents the process P where all actions are named a renamed as b. A guarded replication process $!\pi.P$ represents an unlimited number of instances of $\pi.P$ in parallel.

$$\begin{array}{llll}
[\text{c\_act}] & \pi.P \xrightarrow{\pi} P & [\text{c\_rep}] & !\pi.P \xrightarrow{\pi} !\pi.P \parallel P \\
[\text{c\_choice}_1] & \dfrac{P_1 \xrightarrow{\pi} P_1'}{P_2 + P_1 \xrightarrow{\pi} P_2 + P_1'} & [\text{c\_choice}_2] & \dfrac{P_1 \xrightarrow{\pi} P_1'}{P_1 + P_2 \xrightarrow{\pi} P_1' + P_2} \\
[\text{c\_parallel}_1] & \dfrac{P_1 \xrightarrow{\pi} P_1'}{P_2 \parallel P_1 \xrightarrow{\pi} P_2 \parallel P_1'} & [\text{c\_parallel}_2] & \dfrac{P_1 \xrightarrow{\pi} P_1'}{P_1 \parallel P_2 \xrightarrow{\pi} P_1' \parallel P_2} \\
[\text{c\_comm}_1] & \dfrac{P_1 \xrightarrow{a} P_1' \quad P_2 \xrightarrow{\overline{a}} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\tau} P_1' \parallel P_2'} & [\text{c\_comm}_2] & \dfrac{P_1 \xrightarrow{\overline{a}} P_1' \quad P_2 \xrightarrow{a} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\tau} P_1' \parallel P_2'} \\
[\text{c\_res}] & \dfrac{P \xrightarrow{\pi} P'}{P \setminus a \xrightarrow{\pi} P \setminus a} \pi \neq a & [\text{c\_rel}] & \dfrac{P \xrightarrow{\pi} P'}{P[b/a] \xrightarrow{\pi\{b/a\}} P'[b/a]}
\end{array}$$

Figure 2.6: The operational semantics of CCS processes.

The operational semantics of CCS processes is given by the transition relation defined in Fig. 2.6. The labels π, π' of the transitions are τ, a and $\overline{a}$. We use $c\{b/a\}$ to denote b if $c = a$, otherwise c. The rule [c_act] describes actions of processes. A process $\pi.P$ performs an action π, then behaves like P. The rules [c_par$_1$] and [c_par$_2$] express the parallel computation of processes, while the rules [c_choice$_1$] and [c_choice$_2$] represent a choice among alternatives. The rules [c_com$_1$] and [c_com$_2$] are used to represent the synchronisation on a certain free name. The rule [c_res] ensures that an action π of P is also an action of $P \setminus a$, if the action is not restricted by a, i.e $\pi \neq a$. The rule [c_rep] instantiates an instance of $!\pi.P$ by performing an action π. The result is a parallel composition of P and $!\pi.P$. Finally, in the rule [c_rel], transition labels are renamed by a substitution of a with b.

2.4 The π-Calculus

In this section, we briefly recall the π-calculus [97]. The π-calculus can be thought as an extension of CCS. The mobility of processes and names is modelled in the π-calculus. More precisely, processes not only synchronise on a name, but also *send/receive* a value over a name. A name may have its own *scope* and *extrude* it.

2.4.1 Syntax

Definition 2.4.1. Given a set $\mathcal{N}$ of channel names, ranged over by x, y, z, a, b. A set $\mathcal{P}$ of processes is defined by the following grammar:

$$\begin{array}{lll}
P, P' ::= & & \textit{processes} \\
& \mathbf{0} & \text{empty process} \\
| & \pi.P & \text{prefix action} \\
| & (\nu x)\ P & \text{restriction} \\
| & P + P' & \text{choice} \\
| & P \parallel P' & \text{parallel composition} \\
| & !P & \text{replication}
\end{array}
\qquad
\begin{array}{lll}
\pi, \pi' ::= & & \textit{action prefixes} \\
& \tau & \text{internal action} \\
| & a(y) & \text{free input} \\
| & \bar{a}b & \text{free output}
\end{array}$$

The empty process $\mathbf{0}$ denotes an idle process, i.e the process that does nothing. In the prefix actions, the process $\pi.P$ performs the action π and behaves like P. In the process prefixed by a free input $a(y).P$, y is a bound variable with a scope bound by P and P can receive a name along a, which will substitutes y. In the process prefixed by a free output $\bar{a}y.P$, the name y is sent over the channel a. The action τ describes some *internal* activity of process and is not observed from outside. The choice operator denotes non-deterministic choice among processes. The parallel operator describes parallel composition of processes. In the restriction $(\nu x)P$, (νx) denotes a binder for the name x with the scope P. Intuitively, the name x is different from all *external names*. A replication P denotes an infinite number of copies of P running in parallel.

Convention 2.4.2. From now on, for the sake of simplicity, we often omit the trailing $\mathbf{0}$.

In $a(x).P$ and $(\nu x)P$, x is called a *bound* name. The set of free names $\mathsf{fn}(P)$ of a process P is inductively defined as follows:

$$\begin{array}{ll ll}
\mathsf{fn}(\mathbf{0}) & = \emptyset & \mathsf{fn}(\tau.P) & = \mathsf{fn}(P) \\
\mathsf{fn}(a(y).P) & = \{a\} \cup \mathsf{fn}(P) \setminus \{y\} & \mathsf{fn}(\nu x)\ P) & = \mathsf{fn}(P) \setminus \{x\} \\
\mathsf{fn}(\bar{a}b.P) & = \{a, b\} \cup \mathsf{fn}(P) & \mathsf{fn}(P + P') & = \mathsf{fn}(P) \cup \mathsf{fn}(P') \\
\mathsf{fn}(P \parallel P') & = \mathsf{fn}(P) \cup \mathsf{fn}(P') & \mathsf{fn}(!a(y).P) & = \mathsf{fn}(a(y).P)
\end{array}$$

Similarly, the set of bound names $\mathsf{bn}(P)$ of process P is inductively defined as follows:

$$\begin{array}{ll ll}
\mathsf{bn}(\mathbf{0}) & = \emptyset & \mathsf{bn}(\tau.P) & = \mathsf{bn}(P) \\
\mathsf{bn}(a(y).P) & = \{y\} \cup \mathsf{bn}(P) & \mathsf{bn}(\nu x)\ P) & = \{x\} \cup \mathsf{bn}(P) \\
\mathsf{bn}(\bar{a}b.P) & = \mathsf{bn}(P) & \mathsf{bn}(P + P') & = \mathsf{bn}(P) \cup \mathsf{bn}(P') \\
\mathsf{bn}(P \parallel P') & = \mathsf{bn}(P) \cup \mathsf{bn}(P') & \mathsf{bn}(!a(y).P) & = \{y\} \cup \mathsf{bn}(P)
\end{array}$$

The set of names $\mathsf{n}(P)$ of a process P is defined as the union of $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$.

$$\begin{array}{ll}
P \equiv Q \text{ if } P \text{ is } \alpha\text{-equivalent of } Q & \\
(P+Q)+R \equiv P+(Q+R) & (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \\
P+Q \equiv Q+P & P \parallel Q \equiv Q \parallel P \\
P+\mathbf{0} \equiv P & P \parallel \mathbf{0} \equiv P \\
(\nu x)\mathbf{0} \equiv \mathbf{0} & (\nu x)P \parallel Q \equiv (\nu x)(P \parallel Q)\ x \notin \mathsf{fn}(Q), \\
!P \equiv P \parallel !P &
\end{array}$$

Figure 2.7: Structural Congruence.

Definition 2.4.3. If the name b does not occur in the process P, then $P\{b/a\}$ is the process obtained by replacing each free occurrence of a in P by b. A change of bound names in a process P is the replacement of a sub-process $(\nu x)Q$ of P by $(\nu y)Q\{y/x\}$, or the replacement of $a(x).Q$ of P by $a(y).Q\{y/x\}$, where in each case y does not occur in Q. Two processes P and Q are called α-equivalent of if P can be obtained from Q by a finite number of changes of bound names.

We assume a notion of structural congruence on processes and we denote it by $\equiv$. The structural congruence on processes is defined as the least congruence satisfying the clauses in Fig. 2.7.

Definition 2.4.4 (substitution)**.** A substitution, ranged over by σ, is a partial map from names to names. A process $P\sigma$ is P where all free names a are replaced by $\sigma(a)$, where α-equivalent is applied, when needed, to avoid name captures.

2.4.2 Operational Semantics

The operational semantics of the π-calculus is defined by the transition relation given in Fig. 2.8. The labels μ, μ' of transitions are tau action τ, free input $a(y)$, free output $\bar{a}b$, bound output $\bar{a}(b)$. The effect of bound output $\bar{a}(b)$ is to extrude the sent name b from the initial scope to the external environment. The sets of free names $fn(\mu)$, bound names $\mathsf{bn}(\mu)$ in labels are defined as follows:

$$\begin{array}{llll}
\mathsf{fn}(\tau) & = \emptyset & \mathsf{bn}(\tau) & = \emptyset \\
\mathsf{fn}(a(y)) & = \{a\} & \mathsf{bn}(a(y)) & = \{a\} \\
\mathsf{fn}(\bar{a}b) & = \{a,b\} & \mathsf{bn}(\bar{a}b) & = \emptyset \\
\mathsf{fn}(\bar{a}(y)) & = \{a\} & \mathsf{bn}(\bar{a}(y)) & = \{y\}
\end{array}$$

As usual, we write $\mathsf{n}(\mu)$ for $\mathsf{fn}(\mu) \cup \mathsf{bn}(\mu)$.

We consider the late semantics for π-calculus. It essentially differs from the early semantics in the time in which a bound name is bound to a free name. In late semantics, a bound name in free input is instantiated when communication occurs.

The rule [p_act] describes actions of processes, e.g. the silent action, free input and free output. Concretely, $\bar{a}b.P$ sends the name b along the channel a and then behaves like P, while $a(y).P$ receives a name via the channel a, to which y is bound,

$$\begin{array}{llll}
\text{[p\_act]} & \pi.P \xrightarrow{\pi} P & \text{[p\_cong]} & \dfrac{P_1 \equiv P_1' \quad P_1' \xrightarrow{\mu} P_2' \quad P_2' \equiv P_2}{P_1 \xrightarrow{\mu} P_2} \\
\text{[p\_par]} & \dfrac{P_1 \xrightarrow{\mu} P_1' \quad \mathsf{bn}(\mu) \cap \mathsf{fn}(P_2) = \emptyset}{P_1 \parallel P_2 \xrightarrow{\mu} P_1' \parallel P_2} & \text{[p\_choice]} & \dfrac{P_1 \xrightarrow{\mu} P_1'}{P_1 + P_2 \xrightarrow{\mu} P_1'} \\
\text{[p\_res]} & \dfrac{P \xrightarrow{\mu} P' \quad x \notin \mathsf{n}(\mu)}{(\nu x)P \xrightarrow{\mu} (\nu x)P'} & \text{[p\_open]} & \dfrac{P \xrightarrow{\bar{a}x} P'}{(\nu x)P \xrightarrow{\bar{a}(x)} P'}\ x \neq a \\
\text{[p\_comm]} & \dfrac{P_1 \xrightarrow{\bar{a}y} P_1' \quad P_2 \xrightarrow{a(z)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{y/z\}} & \text{[p\_close]} & \dfrac{P_1 \xrightarrow{\bar{a}(y)} P_1' \quad P_2 \xrightarrow{a(z)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\tau} (\nu y)(P_1' \parallel P_2'\{y/z\})}
\end{array}$$

Figure 2.8: The Operational Semantics of Processes.

and then behaves like P. Note that the considered semantics is the late one, e.g. w is actually bound to a value when a communication occurs. Finally, $\tau.P$ performs the silent action τ and then behaves like P. In the rule [p_cong], structurally congruent processes behave the same.

The rule [p_par] expresses the parallel behaviour of processes, while the rule [p_choice] represents a choice among alternatives. The side condition in the rule [p_par] ensures that the bound name in the transition label is fresh to avoid clash. The rule [p_comm] is used to communicate free names, while the rule [p_close] is used to communicate bound names. The rules [p_res] and [p_open] are rules for restriction. The first ensures that an action of P is also an action of $(\nu x)P$, provided that the restricted name x is not in the action. In the case of x in the action, the rule [p_open] transforms a free output action $\bar{a}x$ into a bound output action $\bar{a}(x)$, which basically expresses opening scope of a bound name. The rule [p_close] describes communication of bound names, which also closes the scope of the bound name in communication x.

2.4.3 Control Flow Analysis

In this section, we present a Control Flow Analysis following [26, 87] for the π-calculus that statically predicts how names are bound to actual names at run-time. More precisely, it computes an over-approximation of the set of names bound to a given name and the set of names possibly sent along a given name. These approximations give us *estimates*. Further, there exists the least estimate among them in sense that all other estimates *contains* the least one.

To simplify the definition of Control Flow Analysis, a discipline in the choice of fresh names, and therefore to α-equivalent, is imposed. Indeed, the result of analysing a process P must still hold for all its derivative processes Q, including all the processes obtained from Q by α-equivalent. In particular, the CFA uses the names and the variables occurring in P. If they were changed by the dynamic

$(\rho, \kappa) \models \mathbf{0}$	iff true
$(\rho, \kappa) \models \tau.P$	iff $(\rho, \kappa) \models P$
$(\rho, \kappa) \models \bar{a}y.P$	iff $\forall b \in \rho(a) : \rho(y) \subseteq \kappa(b) \ \wedge \ (\rho, \kappa) \models P$
$(\rho, \kappa) \models a(y).P$	iff $\forall b \in \rho(a) : \kappa(b) \subseteq \rho(y) \ \wedge \ (\rho, \kappa) \models P$
$(\rho, \kappa) \models P_1 + P_2$	iff $(\rho, \kappa) \models P_1 \wedge (\rho, \kappa) \models P_2$
$(\rho, \kappa) \models P_1 \parallel P_2$	iff $(\rho, \kappa) \models P_1 \wedge (\rho, \kappa) \models P_2$
$(\rho, \kappa) \models (\nu x)P$	iff $(\rho, \kappa) \models P \wedge x \in \rho(x)$
$(\rho, \kappa) \models !P$	iff $(\rho, \kappa) \models P$

Table 2.3: CFA Equational Laws

evolution, the analysis values would become a sort of dangling references, no more connected with the actual values. To statically maintain the identity of values and variables, all the names used by a process are partitioned into finitely many equivalence classes. $\lfloor a \rfloor$, that is called *canonical name* of a, is used to denote the equivalence class of the name a. We simply write a for $\lfloor a \rfloor$, when unambiguous. Two names can be α-renamed only when they have the same canonical name.

The result of analysing a process P is a tuple (ρ, κ). The first component gives information about a set of names to which given names can be bound; the second component contains information about a set of names which can be sent on given names. Formally, the analysis keeps track of the following information:

- An approximation $\rho : \mathcal{N} \cup \mathcal{R} \to \wp(\mathcal{N} \cup \mathcal{R})$ of names bindings. If $a \in \rho(x)$ then the channel variable x can assume the channel value a.
- An approximation $\kappa : \mathcal{N} \to \wp(\mathcal{N} \cup \mathcal{R})$ of the values that can be sent on each channel. If $b \in \kappa(a)$, then b can be sent on the channel a.

To validate the correctness of a given estimate (ρ, κ), a set of clauses that operate upon judgements in the form $(\rho, \kappa) \models P$ are stated. The analysis correctly captures the behaviour of P, i.e. the estimate (ρ, κ) is valid for all the derivatives P' of P.

The judgement rules of CFA is given in Tab 2.3. All the clauses dealing with a compound process check that the analysis also holds for its immediate sub-processes. In particular, the analysis of $(\nu x)P$ and of $!P$ is equal to the one of P. This is an obvious source of imprecision (in the sense of over-approximation).

Besides the validation of the continuation process P, the rule for output, requires that the set of names that can be communicated along each element of $\rho(a)$ includes the names to which y can evaluate. Symmetrically, the rules for input demands that the set of names that can pass along a is included in the set of names to which y can evaluate. Intuitively, the estimate components take into account the possible dynamics of the process under consideration. The clauses' checks mimic the semantic evolution, by modelling the semantic preconditions and the consequences of the possible synchronisations. In the rule for input, e.g. CFA checks whether the precondition of a synchronisation is satisfied, i.e. whether there is a corresponding output possibly sending a value that can be received by the analysed input. The conclusion imposes the additional requirements on the estimate components, necessary to give a valid prediction of the analysed synchronisation action, mainly that the variable y can be bound to that value.

Existence of Estimates and Correctness. An estimate for a given process P always exists. Moreover, the *least* estimate exists as well. Informally, the least estimate is contained in all other estimates. This relation, denoted by $\sqsubseteq$, is formally defined in the following.

Definition 2.4.5. The set of estimates can be partially ordered by setting $(\rho, \kappa) \sqsubseteq (\rho', \kappa')$ if and only if $\forall a \in \mathcal{N}$: $\rho(a) \subseteq \rho'(a)$ and $\kappa(a) \subseteq \kappa'(a)$.

Definition 2.4.6. A set $\mathcal{I}$ of proposed estimates is a Moore family if and only if it contains $\sqcap \mathcal{J}$ for all $\mathcal{J} \subseteq \mathcal{I}$, where $\sqcap \mathcal{J}$ denote the greatest lower bound of $\mathcal{J}$. Note that the least element of $\mathcal{I}$ is $\sqcap \mathcal{I}$.

Theorem 2.4.7 (Existence of the least estimate)**.** *For all P, the set of estimates*

$$\{(\rho, \kappa) | (\rho, \kappa) \models P\}$$

is a Moore family.

Correctness of the least estimate with respect to the semantics is proved by the following theorem.

Theorem 2.4.8 (Subject Reduction)**.** *If $(\rho, \kappa) \models P$ and $P \xrightarrow{\mu}^{*} P'$ then $(\rho, \kappa) \models P'$.*

2.4.4 The Behavioural Type System

In this section, we present a type system developed for the π-calculus, where resources are channel names. We follow the *local* version of the type system introduced in [4]. The main characterisations of this approach is to maintain the structure of processes in types. This allows to check various properties of processes be inspecting their corresponding types. Properties are defined in term of the Shallow Logic, which can be regarded as a fragment of Caires and Cardelli's Spatial Logic [40]. The logic allows one to specify the dynamics as well as the "shallow" spatial structure of processes and types. If a process is well-typed, then properties with which restricted names are annotated can be guaranteed by the type systems. We will omit all technical details of properties and its verification since their developments are not relevant in the next chapters. In the following, we present a *local* version of type systems. We refer readers to [4] for full details.

The syntax of the π-calculus. The development of type system requires a minor extension of the π-calculus. Restrictions are annotated by types (see below). Moreover, properties of restrictions, defined as a formula of Shallow Logic, are also introduced (see below).

To keep track of the structure of processes in types, types of restricted names are explicitly given. For sake of simplicity, we consider the monadic version of the π-calculus. The syntax is defined as follows.

Definition 2.4.9. Let $\mathcal{N}$ ve a set of channel names, ranged over by x, y, z, a, b and Φ be a set of all Shallow Logic formulae, ranged over by ϕ. The set $\mathcal{P}^*$ of processes is defined by the following grammar, where t is a channel type (see below).

$$\begin{array}{llll}
P, P' ::= & & \textit{processes} \\
& \mathbf{0} & \text{empty process} \\
| & \pi.P & \text{prefix action} \\
| & (\nu x : t)\ P & \text{restriction} \\
| & P + P' & \text{choice} \\
| & P \parallel P' & \text{parallel composition} \\
| & !a(y).P & \text{guarded replication}
\end{array}
\qquad
\begin{array}{llll}
\pi, \pi' ::= & & \textit{action prefixes} \\
& \tau & \text{internal action} \\
| & a(y) & \text{free input} \\
| & \bar{a}b & \text{free output}
\end{array}$$

Convention 2.4.10. Often, we write $\tilde{a}$ for a finite set of names and $\nu\tilde{x}$ for a finite sequence of restrictions.

Type annotations should guarantee that the correspondence between the spatial structure of processes and types is preserved the scope extrusion. To this end, it requires that the set of free names of a process includes the set of free names in types of annotated restrictions. Formally, the set of free names fn and bound names bn of a process are defined as before, except for $\mathsf{fn}((\nu x : t)\ P) = (\mathsf{fn}(P) \cup \mathsf{fn}(t)) \setminus \{x\}$, where $\mathsf{fn}(t)$ is defined in the next section. The free names of a formula ϕ, written $\mathsf{fn}(\phi)$, are defined as expected. The set of logical operators includes spatial $(a, a, |, H)$ as well as dynamic $(\langle a \rangle, \langle \tilde{a} \rangle, \langle -\tilde{a} \rangle)$ connectives, beside the usual boolean connectives, including a constant true for"true".

The structural congruence $\equiv$ are defined as usual, except that we drop two rules for restrictions $(\nu x : t)(\nu y : t')P \equiv (\nu y : t')(\nu x : t)P$ and $(\nu x : t)\mathbf{0} \equiv \mathbf{0}$ and the rule for replication $!a(y).P \equiv a(y).P \parallel !a(y).P$. The operational semantics is similar to the one previously defined as in Section 2.4.2, except for the rules for communications and the rule for replication. We use the labels $\langle a \rangle$ for communications in the rules [p_comm] and [p_close] as described below.

$$\text{[p\_comm]} \quad \frac{P_1 \xrightarrow{\bar{a}y} P_1' \quad P_2 \xrightarrow{a(z)} P_2'}{P_1 \parallel P_2 \xrightarrow{\langle a \rangle} P_1' \parallel P_2'\{y/z\}}$$

$$\text{[p\_close]} \quad \frac{P_1 \xrightarrow{\bar{a}(x)} P_1' \quad P_2 \xrightarrow{a(z)} P_2'}{P_1 \parallel P_2 \xrightarrow{\langle a \rangle} (\nu x)(P_1' \parallel P_2'\{x/z\})}$$

The labels of communications allow us to show the corresponding behaviours of processes and behavioural types (introduced in the next section). Replication is not ruled by structural congruence, but by the operational semantics as follows:

$$\text{[p\_rep]} \quad !a(y).P \xrightarrow{a(y)} !a(y).P \parallel P$$

Intuitively, an instance of $!a(y).P$ is instantiated by performing $a(y)$. The resulting process is the parallel composition of P and $!a(y).P$.

The syntax of behavioural types.

Definition 2.4.11. Let $\mathcal{N}$ be a set of channel names, ranged over by a, b , a set $\mathcal{R}$ of resource names, ranged over by r, r', and a set $\mathcal{V}_T$ of type variables, ranged over by X, Y. The set $\mathcal{T}$ of behavioural types, ranged over by T, T', are defined by the following grammar:

$$\begin{array}{lll}
\text{(process types) } T, T' & ::= & \mathbf{0} \mid \pi.T \mid (\nu x : t)\ T \mid T + T' \mid T \parallel T' \mid !a(t).T \\
\text{(prefix actions) } \pi, \pi' & ::= & \tau \mid a(t) \mid \overline{a} \\
\text{(channel types) } t, t' & ::= & (x : t)T \mid ()T,
\end{array}$$

A type T is called *closed* if it does not contain any type variable.

The set of free names $\mathsf{fn}(T)$ and bound names $\mathsf{bn}(T)$ of a given closed type T is defined as expected, i.e. :

$$\begin{array}{llll}
\mathsf{fn}(\mathbf{0}) & = \emptyset & \mathsf{fn}(a(t).T) & = \{a\} \cup \mathsf{fn}(t) \cup \mathsf{fn}(T) \\
\mathsf{fn}(\bar{a}.T) & = \{a\} \cup \mathsf{fn}(T) & \mathsf{fn}(\tau.T) & = \mathsf{fn}(T) \\
\mathsf{fn}(T + T') & = \mathsf{fn}(T) \cup \mathsf{fn}(T') & \mathsf{fn}(T \parallel T') & = \mathsf{fn}(T) \cup \mathsf{fn}(T') \\
\mathsf{fn}((\nu x : t)T) & = \mathsf{fn}(t) \cup \mathsf{fn}(T) \setminus \{x\} & \mathsf{fn}((x : t)T) & = \mathsf{fn}(t) \cup \mathsf{fn}(T) \setminus \{x\} \\
\mathsf{fn}(!a(t).T) & = \mathsf{fn}(a(t).T) & &
\end{array}$$

$$\begin{array}{llll}
\mathsf{bn}(\mathbf{0}) & = \emptyset & \mathsf{bn}(a(t).T) & = \mathsf{bn}(T) \\
\mathsf{bn}(\bar{a}.T) & = \mathsf{bn}(T) & \mathsf{bn}(\tau.T) & = \mathsf{bn}(T) \\
\mathsf{bn}(T + T') & = \mathsf{bn}(T) \cup \mathsf{bn}(T') & \mathsf{bn}(T \parallel T') & = \mathsf{bn}(T) \cup \mathsf{bn}(T') \\
\mathsf{bn}((\nu x : t)T) & = \mathsf{bn}(T) \cup \{x\} & \mathsf{bn}(!a(t).T) & = \mathsf{bn}(a(t).T)
\end{array}$$

Note that t in a process type $a(t).T$ contribute free names to a set of free names of that type and the same is for a channel type $(x : t)T$. We write $\mathsf{n}(T)$ for $\mathsf{fn}(T) \cup \mathsf{bn}(T)$.

Definition 2.4.12. If the name b does not occur in the process T, then $T\{b/a\}$ is the process obtained by replacing each free occurrence of a in T by b. A change of bound names in a process T is the replacement of a sub-process $(\nu x)T'$ of T by $(\nu y)T'\{y/x\}$, or the replacement of $a(x).T'$ of T by $a(y).T'\{y/x\}$, where in each case y does not occur in T'. Two types T and T' are called α-equivalent of T' if T can be obtained from T' by a finite number of changes of bound names.

Definition 2.4.13 (substitution)**.** A substitution, ranged over by σ, is a partial map from names to names. A type T, $T\sigma$ is T where all free names $a \in \mathsf{fn}(T)$ are replaced by $\sigma(a)$, where changes of restricted names are applied when needed, to avoid name captures. .

There are two kinds of types: process (or behavioural) types and channel types. In a channel type $(x : t)T$, x is a binder with the scope T, e.g. a process type T pre-describes a usage of x of the type t with dependencies on other channel names. In a special case, where an input is ignore by a channel, its type is denoted by $()T$. We assume that $x \in \mathsf{fn}(T)$. Note that the input prefix $a(t)$ in a process type $a(t).T$ describes the type t of the objects that can be received by T, while a prefix action $\bar{a}$ reflects an output action on the channel name a. For the sake of simplicity, we write $y\#T$ for $y \notin \mathsf{fn}(T)$. Similar notations are defined for $\tilde{a}\#\phi$.

The operational semantics of types.

Definition 2.4.14. The structural congruence on types is the relation defined below

$$\begin{array}{ll}
T \equiv T' \text{ if } T' \text{ is } \alpha\text{-equivalent of } T & \\
T + \mathbf{0} \equiv \mathbf{0} + T \equiv T & T \parallel \mathbf{0} \equiv \mathbf{0} \parallel T \equiv T \\
T_1 + T_2 \equiv T_2 + T_1 & T_1 \parallel T_2 \equiv T_2 \parallel T_1 \\
(T_1 + T_2) + T_3 \equiv T_1 + (T_2 + T_3) & (T_1 \parallel T_2) \parallel T_3 \equiv T_1 \parallel (T_2 \parallel T_3) \\
(\nu x)T \parallel T' \equiv (\nu x)(T \parallel T') \;\text{ if } x \notin \mathsf{fn}(T') &
\end{array}$$

The operational semantics of types, defined in Tab. 2.4. Labels μ, μ' for transitions are τ for silent actions, $a, \bar{a}$ for abstract input/output, and $\langle a\rangle$ for communications. The rule [bt_act] describes actions of types. A type $\pi.T$ performs an action π, then behaves like P. In the rule [bt_cong], congruent types can perform the same action. The rules [bt_par$_1$] and [bt_par$_2$] express the parallel behaviour of types, while the rules [c_choice$_1$] and [bt_choice$_2$] represent a choice among alternatives.

The rule [bt_comm$_1$] and [bt_comm$_2$] are used to synchronize a free name. The rule [bt_res] manages restrictions. The rule [bt_res] ensures that an action μ of T is also an action of $(\nu x)T$, if the name of the action is not restricted by x, i.e $\mu \neq x(t)$ and $\mu \neq \overline{x}$. Finally, the rule [bt_rep] instantiates an instance of $!a(t).T$ by performing $a(t)$. The result is the parallel composition of T and $!a(t).T$.

Basic properties of processes. First, we need some auxiliary definitions. In the following, we use A, B to range over $\mathcal{U} = \mathcal{P}^* \cup \mathcal{T}$. Elements of $\mathcal{U}$ will be generally

[bt_act]	$\pi.T \xrightarrow{\pi} T$	[bt_cong]	$\dfrac{T_1 \equiv T_1' \quad P_1' \xrightarrow{\mu} T_2' \quad T_2' \equiv T_2}{T_1 \xrightarrow{\mu} T_2}$
[bt_par$_1$]	$\dfrac{T_1 \xrightarrow{\mu} T_1'}{T_1 \parallel T_2 \xrightarrow{\mu} T_1' \parallel T_2}$	[bt_par$_2$]	$\dfrac{T_2 \xrightarrow{\mu} T_2'}{T_1 \parallel T_2 \xrightarrow{\mu} T_1 \parallel T_2'}$
[bt_choice$_1$]	$\dfrac{T_1 \xrightarrow{\mu} T_1'}{T_1 + T_2 \xrightarrow{\mu} T_1'}$	[bt_choice$_2$]	$\dfrac{T_2 \xrightarrow{\mu} T_2'}{T_1 + T_2 \xrightarrow{\mu} T_2'}$
[bt_comm$_1$]	$\dfrac{T_1 \xrightarrow{\bar{a}} T_1' \quad T_2 \xrightarrow{a(t)} T_2'}{T_1 \parallel \ T_2 \xrightarrow{\langle a \rangle} T_1' \parallel T_2'}$	[bt_comm$_2$]	$\dfrac{T_1 \xrightarrow{a(t)} T_1' \quad T_2 \xrightarrow{\bar{a}} T_2'}{T_1 \parallel \ T_2 \xrightarrow{\langle a \rangle} T_1' \parallel T_2'}$
[bt_res]	$\dfrac{T \xrightarrow{\mu} T'}{(\nu x)T \xrightarrow{\mu} (\nu x)T'} \mu \neq \overline{x}, x(t)$	[bt_rep]	$!a(t).T \xrightarrow{a(t)} !a(t).T \parallel T$

Table 2.4: The Operational Semantics of Types

referred to as *terms*. A property set, P-set in brief, is a set of terms closed under structural congruence and having a finite support: the latter intuitively means that the set of names that are relevant for the property is finite (somewhat analogous to the notion of free names for syntactic terms). In the following, we let $\{a \leftrightarrow b\}$ denote the *transposition* of a and b, that is, the substitution that assigns a to b and b to a, and leaves the other names unchanged. For $\Phi \subset \mathcal{U}$, we let $A \models \Phi$ mean that $A \in \Phi$, and $\Phi\{a \leftrightarrow b\}$ denote the set $\{A\{a \leftrightarrow b\} | A \models \Phi\}$.

Definition 2.4.15 (Support, P-set, least support)**.** Let $\Phi \subset \mathcal{U}$ and $N \subset \mathcal{N}$.

1. A set N of names is a *support* of Φ if for each $a, b \notin N$, it holds that $\Phi\{a \leftrightarrow b\} = \Phi$.

2. A *property set* (P-set) is a set of terms $\Phi \subset \mathcal{U}$ that is closed under $\equiv$ and has a finite support.

3. The least support of Φ is defined as $\mathsf{supp}(\Phi) = \bigcap_N \text{ is support of }_\Phi N$.

In other words, N is a support of Φ if renaming names *outside* N with fresh names does not affect Φ. P-sets have a finite support, and since countable intersection of supports is still a support, they also have a least support. Furthermore, we introduce a notion of μ-derivative of a P-set (to capture changes of properties through reductions), describing the set of terms reachable via μ-reductions from terms in Φ.

$$\Phi_\mu \triangleq \{B | \exists A \text{ s.t. } A \models \Phi \text{ and } A \xrightarrow{\mu} B\}$$

The following property ensures that a μ-derivative of a P-set is a P-set, provided μ involves a name in the support of Φ.

Lemma 2.4.16. *Let Φ be a P-set. If $\mu = \langle a \rangle$ with $a \in \mathsf{supp}(\Phi)$ then Φ_μ is a P-set and $\mathsf{supp}(\Phi_\mu) \subseteq \mathsf{supp}(\Phi)$.*

The Ok() predicate introduced below identifies P-sets that enjoy certain desirable conditions: (1) requires a P-set to be closed under parallel composition with terms not containing free names; (2) demands a P-set to be invariant under reductions that do not involve names in its support; finally, (3) requires preservation of (1) and (2) under derivatives. These requirements will be essential for guaranteeing the subject reduction property of type systems, introduced in the next section. Note the co-inductive form of the definition.

Definition 2.4.17 ($Ok(.)$ predicate)**.** $Ok(.)$ predicate is defined as the largest predicate on P-sets such that whenever $Ok(\Phi)$ then:

1. for any $A, B \in \mathcal{P}$ s.t. $\mathsf{fn}(B) = \emptyset$ if and only if $A|B \models \Phi$; similarly for $A, B \in \mathcal{T}$.
2. if $\mu = \tau$ or $\mu = \langle b \rangle$ with $b \notin \mathsf{supp}(\Phi)$ then $\Phi_\mu = \Phi$.
3. for each μ, $Ok(\Phi_\mu)$ holds

Definition 2.4.18. The set $\mathcal{F}$ of *Shallow Logic* formulae is given by the following syntax:

$$\phi, \phi' \ ::= \text{true} \mid \phi \vee \phi' \mid \neg\phi \mid \langle a \rangle \phi \mid \langle \tilde{a} \rangle^* \phi \mid \langle -\tilde{a} \rangle^* \phi \mid a \mid \bar{a} \mid \phi|\phi' \mid H^*\phi$$

The interpretation of the Shallow formulae over the set of processes and types is given in Tab. 2.5. Note that a process has a *barb* a (resp. $\bar{a}$), written $P \searrow_a$ (resp. $P \searrow_{\bar{a}}$ whenever $P \equiv (\nu\tilde{x})(P' + a(x).Q \parallel R)$ or $P \equiv (\nu\tilde{x})(P' + \bar{a}\langle b \rangle.Q \parallel R)$, with $a \notin \tilde{x}$. Similar notations are defined for types. Connectives are interpreted in the standard manner. In particular, concerning spatial modalities, the barb atom a (resp. $\bar{a}$) requires that A has an input (resp. output) barb on a; $\phi|\phi'$ requires that A can be split into two independent threads satisfying ϕ and ϕ'; $H^*\phi$ requires that A satisfies ϕ, up to some top level restrictions. Concerning the dynamic part, formula $\langle a \rangle \phi$ requires an interaction with subject a may lead A to a state where ϕ is satisfied; $\langle \tilde{a} \rangle^* \phi$ requires any number, including zero, of reductions with subject in $\tilde{a}$ may lead A to a state where ϕ is satisfied; $\langle -\tilde{a} \rangle^* \phi$ is similar, but it requires that the subjects of the reductions leading to such a state are not in $\tilde{a}$. We write $A \models \phi$ if $A \in \phi$. Interpretations of formulae are P-sets, as stated below.

Lemma 2.4.19. *Let $\phi \in \mathcal{F}$. Then $\|\phi\|$ is a P-set and $\mathsf{fn}(\phi) \supseteq \mathsf{supp}(\|\phi\|)$.*

The Behavioural Type System. The type works on annotated processes, where each restriction introduces a property, defined in term of an Ok P-set, that depends on the restricted names and is expected to be satisfied by the process in the scope of the restriction. For annotated processes, the clause for restriction is

$$P ::= \cdots |(\nu\tilde{x} : \tilde{t}; \Phi)P \text{ with } \tilde{x} \supseteq \mathsf{supp}(\Phi) \text{ and } OK(\Phi).$$

$$
\begin{array}{llll}
\|\text{true}\| & = \mathcal{U} & \|H^*\phi\| & = \{A | \exists \tilde{a}, B : A \equiv (\nu\tilde{x})B, \tilde{x}\#\phi, B \in \|\phi\|\} \\
\|\phi \vee \phi'\| & = \|\phi\| \cup \|\phi'\| & \|\phi | \phi'\| & = \{A | \exists B, B' : A \equiv B \parallel B', B \in \|\phi\|, B' \in \|\phi'\|\} \\
\|\neg\phi\| & = \mathcal{U} \setminus \|\phi\| & \|\langle a\rangle\phi\| & = \{A | \exists B : A \overset{\langle a\rangle}{\to} B, B \in \|\phi\|\} \\
\|a\| & = \{A | A \searrow_a\} & \|\langle\tilde{a}\rangle^*\phi\| & = \{A | \exists \mu, B : A \overset{\mu}{\to} B, \mu \in \{\langle b\rangle | b \in \tilde{a}\}, B \in \|\phi\|\} \\
\|\bar{a}\| & = \{A | A \searrow_{\bar{a}}\} & \|\langle\tilde{a}\rangle^*\phi\| & = \{A | \exists \mu, B : A \overset{\mu}{\to} B, \mu\#\tilde{a}, B \in \|\phi\|\}
\end{array}
$$

Table 2.5: The interpretation of formulae over terms

The reduction rule for restriction of annotated processes takes into account the μ-derivative of Φ in the continuation process. Hence, the rule for restriction on annotated processes is

$$
[\text{p\_res}] \quad \frac{P \overset{\mu}{\to} P' \qquad x \notin \mathsf{n}(\mu)}{(\nu x : t; \Phi)P \overset{\mu}{\to} (\nu x : t; \Phi_\mu)P'}
$$

The judgements of the type system have the form $\Gamma \vdash P : T$, where Γ is a context, $P \in \mathcal{P}$, $T \in \mathcal{T}$. Intuitively, T is an abstract behaviour of P under the context Γ. A context Γ is a map from channel names to channel types. We write $\Gamma \vdash a : t$ if $a \in \mathit{dom}(\Gamma)$ and $\Gamma(a) = t$. A context Γ is well-formed if whenever $\Gamma \vdash a : (x : t)T$ then $\mathsf{fn}(t, T) \subseteq \{x\} \cup \mathit{dom}(\Gamma)$. From now on, we only consider well-formed context. In the type system, we make use of a "hiding" operation on types, $T_{\downarrow\tilde{a}}$, which masks the use of names not in $\tilde{a}$ (as usual, in the definition we assume that all bound names in T and t are distinct from each other and disjoint from the set of free names and from $\tilde{a}$). $T_{\downarrow\tilde{a}}$ is formally defined in Tab. 2.6.

$$
\begin{array}{llll}
\mathbf{0}_{\downarrow\tilde{a}} & = \mathbf{0} & s((\nu\tilde{x} : \tilde{t})T)_{\downarrow\tilde{a}} & = (\nu\tilde{x} : \tilde{t}_{\downarrow\tilde{a},\tilde{x}})T_{\downarrow\tilde{a},\tilde{x}} \\
(\bar{b}.T)_{\downarrow\tilde{a}} & = \begin{cases} \tau.T_{\downarrow\tilde{x}} & \text{if} \;\; \text{if } b \notin \tilde{a} \\ \bar{a}.T_{\downarrow\tilde{a}} & \text{if} \;\; \text{otherwise} \end{cases} & (b(t).T)_{\downarrow\tilde{a}} & = \begin{cases} \tau(t_{\downarrow\tilde{a}}).T_{\downarrow\tilde{x}} & \text{if} \;\; \text{if } b \notin \tilde{a} \\ b(t_{\downarrow\tilde{a}}).T_{\downarrow\tilde{a}} & \text{if} \;\; \text{otherwise} \end{cases} \\
T_1 \parallel T_{2\downarrow\tilde{a}} & = T_{1\downarrow\tilde{a}} \parallel T_{2\downarrow\tilde{a}} & T_1 + T_{2\downarrow\tilde{a}} & = T_{1\downarrow\tilde{a}} + T_{2\downarrow\tilde{a}} \\
\tau.T_{\downarrow\tilde{a}} & = \tau.T_{\downarrow\tilde{a}} & (!b(t).T)_{\downarrow\tilde{a}} & = !b(t)T_{\downarrow\tilde{a}}
\end{array}
$$

Table 2.6: The "hiding" operator on types

We are now ready to comment typing rules. The rules of the type system are given in Tab. 2.7. The key rules are rules for input prefix [ts_input], output prefix [ts_output] and [ts_eq]. In the rule [ts_input], the type of P in $a(x).P$ is split into two components T_1 and T_2. The condition $x\#T_1$ guarantees that all type information depending on x is collected in T_2. Furthermore, T_2 must match exactly T_a, a type that describes the usage of argument x on channel a.

$$[\text{ts\_empty}] \quad \Gamma \vdash \mathbf{0} : \mathbf{0} \qquad\qquad [\text{ts\_rep}] \quad \frac{\Gamma \vdash P : T}{\Gamma \vdash !P : !T}$$

$$[\text{ts\_output}] \quad \frac{\begin{array}{c}\Gamma \vdash P : T_1 \\ \Gamma \vdash a : (y : t)T_2 \quad \Gamma \vdash b : t\end{array}}{\Gamma \vdash \bar{a}b.P : \bar{a}.(T_1 \parallel T_2\{b/y\})} \qquad [\text{ts\_input}] \quad \frac{\begin{array}{c}\Gamma \vdash a : (y : t)T_a \\ \Gamma, y : t \vdash P : T_1 \parallel T_2 \quad T_2 = T_a \;\; y\#T_1\end{array}}{\Gamma \vdash a(y).P : a.T_1}$$

$$[\text{ts\_choice}] \quad \frac{\Gamma \vdash P_1 : T_1 \; \Gamma \vdash P_2 : T_2}{\Gamma \vdash P_1 + P_2 : T_1 + T_2} \qquad [\text{ts\_par}] \quad \frac{\Gamma \vdash P_1 : T_1 \; \Gamma \vdash P_2 : T_2}{\Gamma \vdash P_1 \parallel P_2 : T_1 \parallel T_2}$$

$$[\text{ts\_res}] \quad \frac{\Gamma, \tilde{x} : \tilde{t} \vdash P : T \quad T_{\downarrow\tilde{x}}}{\Gamma \vdash (\nu\tilde{x} : \tilde{t}; \Phi)P : (\nu\tilde{x} : \tilde{t})T} \qquad [\text{ts\_tau}] \quad \frac{\Gamma \vdash P : T}{\Gamma \vdash \tau.P : \tau.T}$$

$$[\text{ts\_eq}] \quad \frac{\Gamma \vdash P : T \quad T \equiv T'}{\Gamma \vdash P : T'}$$

Table 2.7: Typing rules

In the rule [ts_output], the type $\bar{a}b.P$ is the parallel composition of the type of P and a continuation $T_a\{b/x\}$ with the actual argument b, provided that a has a type $(x : t)T_a$. Rules [ts_input] and [ts_output] are asymmetric in the sense that when type checking receiver $a(x).P$, the type information of P depending on the input parameters x is moved to the sender side. The rule [ts_eq] is related to sub-typing. The idea of using the structural congruence instead of a preorder on types is to maintain the spatial structure of processes in types.

In the rules [ts_par] and [ts_choice], the resulting type is obtained as composition of types of the components. In the rules [ts_act], [ts_tau], the resulting types extend the types in the premise to reflect the structure of processes. In the rule of restriction [ts_res], the abstraction T obtained for P is used to check that P's usage of names $\tilde{x}$ fulfils the property Φ ($T_{\downarrow\tilde{x}} \models \Phi$): in practical cases, Φ is a shallow logic formula and this is actually spatial model checking. Note that $T \models \Phi$ might be undecidable, however significant decidable fragments will be identified in the following.

Basic Properties As said before, types reflect the structure of processes, which is stated by the following lemmas. A *normal* derivation of $\Gamma \vdash P : T$, denoted by $\Gamma \vdash_N P : T$, is a derivation where rule [ts_eq] can only be found immediately above rule [ts_input]. The following lemma states that every derivation of $\Gamma \vdash P : T, E$ has its normal derivation.

Lemma 2.4.20 (Normal Derivation). *If* $\Gamma \vdash P : T$ *then* $\Gamma \vdash_N P : T'$, $T \equiv T'$.

Normal derivations are syntax-directed, that is, processes and their types share the same shallow structure. The type inversion lemmas given below show that we can obtain the *shallow* structure of a process reflected in its type, and vice versa. This correspondence also reflects in the semantics of processes and types (see below). This is formally stated by the two lemmas below.

Lemma 2.4.21 (Type Inversion)**.** *Given $\Gamma \vdash_N P : T, E$, $\Gamma \vdash_N b : t$ and $\Gamma \vdash a : (x : t)U_a$. Then for any Q, Q_1, Q_2 and $\pi.Q$, it holds that:*

- *1. If $P = a(x).Q.$then $T = a.(S \parallel U)$ for some S such that $\Gamma \vdash_N Q : S \parallel U$, $x \# S$ and $U = U_a$.*
- *2. If $P = \bar{a}b.Q$ then $T = \bar{a}.(S \parallel S')$ for some S such that $\Gamma \vdash_N Q : S$ and $S' = U_a\{b/x\}$.*
- *3. If $P = \tau.Q$ then $T = \tau.S$ for some S such that $\Gamma \vdash_N Q : S$.*
- *4. If $P = (\nu x)Q$ then $T = (\nu x)S$ for some S such that $\Gamma, x : t' \vdash_N Q : S$.*
- *5. If $P = Q_1 \parallel Q_2$ then $T = S_1 \parallel S_2$ for some S_1 and S_2 such that $\Gamma \vdash_N Q_1 : S_1$ and $\Gamma \vdash_N Q_2 : S_2$.*
- *6. If $P = Q_1 + Q_2$ then $T = S_1 + S_2$ for some S_1 and S_2 such that $\Gamma \vdash_N Q_1 : S_1$ and $\Gamma \vdash_N Q_2 : S_2$.*
- *7. If $P = !a(x).Q$ then $T = !a.S$ for some S such that $\Gamma \vdash_N a(x)Q : a.S$.*

Lemma 2.4.22. ***Process Inversion*** *Given $\Gamma \vdash_N P : T, E$, $\Gamma \vdash_N b : t$ and $\Gamma \vdash a : (x : t)U_a$. Then for any Q, Q_1, Q_2 and $\pi.Q$, it holds that:*

- *1. If $T = a((x : t)U_a).S$ then $P \equiv a(x).Q$ for some Q such that $\Gamma \vdash_N Q : S \parallel U_a$, $x \# S$.*
- *2. If $T = \bar{a}.S$ then $P \equiv \bar{a}b.Q$ for some S' such that $\Gamma \vdash_N Q : S'$ and $S = U_a\{b/x\} \parallel S'$.*
- *3. If $T = \tau.S$ then $P \equiv \tau.Q$ for some Q such that $\Gamma \vdash_N Q : S$.*
- *4. If $T = (\nu x)S$ then $P \equiv (\nu x)Q$ for some Q such that $\Gamma, x : t' \vdash_N Q : S$.*
- *5. If $T = S_1 \parallel S_2$ then $P \equiv Q_1 \parallel Q_2$ for some Q_1 and Q_2 such that $\Gamma \vdash_N Q_1 : S_1$ and $\Gamma \vdash_N Q_2 : S_2$.*
- *6. If $T = S_1 + S_2$ then $P \equiv Q_1 + Q_2$ for some Q_1 and Q_2 such that $\Gamma \vdash_N Q_1 : S_1$ and $\Gamma \vdash_N Q_2 : S_2$.*
- *7. If $T = !a((x : t)U_a).S$ then $P \equiv !a(x).Q$ for some Q such that $\Gamma \vdash_N a(x)Q : a.S$.*

The subject reduction property holds in the type system. Intuitively, if a process P is typed under Γ, i.e. there exists T such that $\Gamma \vdash P : T$, then all derivatives from P are also typed under Γ. Furthermore, its "inverse" version, i.e. type subject reduction, also holds.

Theorem 2.4.23. ***Subject Reduction*** *If* $\Gamma \vdash P : T$ *and* $P \xrightarrow{\mu} P'$, *where* μ *is* $\langle a \rangle$ *or* τ, *then there exists a* T' *such that* $T \xrightarrow{\mu} T'$ *and* $\Gamma \vdash P' : T'$.

Theorem 2.4.24. ***Type Subject Reduction*** *If* $\Gamma \vdash P : T$ *and* $T \xrightarrow{\mu} T'$, *where* μ *is* $\langle a \rangle$ *or* τ, *then there exists a* P' *such that* $P \xrightarrow{\mu} P'$ *and* $\Gamma \vdash P' : T'$.

Type soundness First, we present classes of properties for which well-typed-ness implies well-annotated-ness. In principle, model checking on processes against these classes of properties can be reduced to a type checking problem whose solution requires only a (local) use of model checking on types.

Definition 2.4.25 (locally checkable properties)**.** We let Lc be the largest predicate on P-sets such that whenever $Lc(\Phi)$ then $Ok(\Phi)$ and:

1. whenever $\Gamma \vdash P : T$ and $\tilde{a} \supseteq \mathsf{supp}(\Phi)$ and $T_{\downarrow \tilde{a}} \models \Phi$ then $P \models \Phi$.
2. $Lc(\Phi_\mu)$ holds for each μ.

Definition 2.4.26 (well-annotated processes)**.** A process $P \in \mathcal{P}$ is well-annotated if whenever $P \equiv (\nu \tilde{x})(\nu \tilde{y} : \Phi)Q$ then $Q \models \Phi$

Theorem 2.4.27. ***Type Soundness*** *Suppose* $\Gamma \vdash P : T$ *and* P *is decorated with locally checkable* P*-sets only. Then* P *is well-annotated.*

Theorem 2.4.28. ***Run-time soundness*** *Suppose* $\Gamma \vdash P : T$ *and* P *is decorated with locally checkable* P*-sets only. Then* $P \xrightarrow{\mu_1} P_1 \xrightarrow{\mu_2} \ldots P'$ *implies that* P' *is well-annotated.*

Part I

A Model of Cloud Systems

Chapter 3

Lambda in Clouds

3.1 Introduction

In the old times, people used to exploit the bakery's oven for their home-made bread. Similarly, people utilised the public mill to obtain flour from their wheat. In both cases, people did not own the physical infrastructure to process their products, neither they invested on it. We refer to [**?**] for a general description of Cloud Computing. Cloud Computing customers do not invest on hardware, software or services, but they just pay providers to use them, either on a utility or a subscription basis. Cloud services over the Internet are therefore used on demand and with a certain degree of flexibility. Usually, these services rely on (a farm of) servers, often virtual ones and are fully managed by their providers. As a consequence, old and new security problems may arise, because if security is related to trust, as Schneier [99] wrote "[cloud computing] moves the trust boundary one step further ... You have to trust your outsourcer completely. You not only have to trust the outsourcer's security, but its reliability, its availability and its business continuity". Therefore, Cloud computing borrows from well-established technologies and models, in particular from the Service Oriented Computing (SOC) ones. Nevertheless, the possibility to offer resources, on demand, in a multi-tenant environment, and in a scalable way, makes cloud computing emerge as a new combination that deserves to be modelled and investigated on its own. Our understanding drives us to propose a model in which the loosely coupled nature of services is compensated by a coarser view of resource usages. Furthermore, while SOC applications are mainly concerned with business logic, cloud applications must conciliate the business logic with the operation logic [22]. To summarise, our model for clouds supports service orientation, integrated by the operation logic of resources.

In this chapter, we propose and outline a formal framework (borrowed by [15, 12]) for specifying and reasoning about cloud computing systems. We build on the functional model offered by a concurrent version of λ-calculus, enriched with primitives for handling resources, for describing and assembling services, and for managing security properties. In the following, we informally present the main features of our approach.

Cloud resources as functions with side effects. We view a cloud server as being composed of several computational components interacting through well-defined interfaces. Components may assume a variety of forms, e.g. virtual machines, databases, resource schedulers and so on. We argue that it is effective and useful to abstractly view cloud components as functions with a side effect, modelling changes of the resource state. For instance, let us consider a simple database service, offered by a cloud, that gets a string query from the user and accordingly queries the database. The following functional interface describes the database service outlined above.

```
Table fun Q(Query q):  Effect e
```

The invocation of the service `Q` with the actual query will yield a table value as result. The side effect `e` provides the abstract representation of the database changes such as updates of tables. By applying the typing techniques developed in [15, 16], we describe the interface of the service `Q`, through an annotated functional type, of the form $\texttt{Query} \xrightarrow{e} \texttt{Table}$. When supplied with an argument of type `Query`, the service evaluates to an object of type `Table`. The annotation `e` is the *side effect* of service evaluation that abstractly describes the possible run-time traces of service executions.

The main benefit of the idea of considering cloud components as functions with side effects is that it provides a *high-level* notion to model cloud resources, their composition and interactions, by abstracting from low level implementation details. This choice has also the methodological spin-off that each cloud computational component has to expose a well defined interface containing both supported operations and an abstract behavioural description.

Security Policies. In spite of its undeniable advantages and cost-savings, cloud computing makes data processing inherently risky, as data and computation reside not under the user's control, as effectively said by Diffie in an interview [51]: "... The effect of the growing dependence on cloud computing is similar to that of our dependence on public transportation ... which forces us to trust organizations over which we have no control, ... and subjects us to rules and schedules that we wouldn't apply ... On the other hand, it is so much more economical that we don't realistically have any alternative. ... [Concerning safety] from the view of a broad class of potential users it is very much like trusting the telephone company ... to keep your communications private".

Classical security concerns are therefore more crucial, and also assuming that the underlying networking infrastructure manages the more basic factors, design flaws can arise and make cloud services *unsafe*. Often security problems do not depend on weird attacks, but simply on the application of careless policies or insufficient policy enforcement. Consequently, it is essential that safety is addressed when designing a cloud.

Our programming model focuses on application-based security by considering *security policies* as first class programming constructs. We provide explicit constructs to declare and enforce the security policies governing the behaviour of cloud components, in the style of [11, 13]. In our framework, a security policy regulates how cloud components are granted to and used. For instance, let us consider the database service example introduced above. The `Q` service may be unsafe although the code normally runs in most of the cases. An attacker can indeed taint the query string by injecting a command in front; consequently the service would issue dangerous commands such as deleting a file before executing the safe query. This is called an *SQL injection bug*.

Sequences of resource accesses in executions are called *histories*. A *security policy* φ is a regular property of histories. Policies are expressed as languages accepted by an extension of finite state automata, since automata recognize those words that violate the desired property. We refer to the general case of policies as regular safety properties [14, 16]. While evaluating a program fragment e protected by a policy φ, written as $\varphi[e]$, the histories must respect φ.

From a methodological perspective, the awareness of security issues from the very beginning of the development process facilitates the design of secure clouds: security is faced in advance, without sweeping it under the carpet (read it as security patches added later). The database service example above can be moved into a more secure land, by wrapping it inside a suitable security policy φ_{DB}. For instance, the policy can impose that no update operations on the database (i.e. system commands) can be issued during service executions, i.e. the only operations allowed are those in which the database content can only be read. Adding the specified policy to the query interface results in:

```
Table fun Q(Query q):  Effect e ensuring φ_DB
```

meaning that each step of service execution must obey the security policy φ_{DB}. Operationally, the run-time structures will enforce the security policy φ_{DB} by monitoring service execution and by catching the occurrences of possible bad actions, i.e. the actions that violate the policies. Actually, the run-time enforcement mechanism depends on a suitable abstraction of the execution of all the pieces of code (possibly partially) executed so far. This implies that the mechanism enforcing the security policies can make decisions, based on all previous changes of shared resources affected by different user requests. This approach, known under the name of *history-based security*, has been receiving major attention, at both levels of foundations [9, 56, 101] and of language design/implementation [1, 52].

Cloud server. Abstractly a cloud server can be seen as a pool of components and resources running over a variety of virtual machines. We keep the vision of

Software as a service, where each service exposes over the network certain functional behaviour and it is invoked via request/response communication protocols (e.g. SOAP). In our programming model, a cloud server is a triple consisting of (i) the history representing the global cloud state (that represents the dependencies among services and resources, as well as virtual machine configurations), (ii) the set of active processes, and (iii) a service environment that associates each service name with the script, or the service code, required to load the virtual machine and the resources needed to run the service.

For example, let us consider a cloud server, whose service environment provides facilities to convert files from one format to other formats. The initial configuration of the cloud server only includes the cloud services. Notice that these services are idle: they are activated by service invocation. We do not model here how clients operate. Clients interactions are asynchronously observed, by means of the server operations required to activate VM as well as the resource needed to operate. For instance, in our example, the initial configuration includes an empty history ϵ, an empty set of active processes $\mathbf{0}$ and a service environment with two possible services $F2F_1$, $F2F_2$ available to convert files having a certain source format:

$$(\epsilon, \mathbf{0}, \{F2F_1 \to p_1, F2F_2 \to p_2\})$$

These two services could be characterized by the following types that declare both information about the virtual machines attached to the services and about the costs of service invocations.

```
Format$_1$ fun F2F$_1$(Format file):  Effect ActivateVM$_1$; c$_1$
Format$_2$ fun F2F$_2$(Format file):  Effect ActivateVM$_2$; c$_2$
```

Our cloud server may activate a translation service by the following transition:

$$(\epsilon, \mathbf{0}, \{F2F_1 \to p_1, F2F_2 \to p_2\}) \xrightarrow{invoke_{F2F_1}} (invoke_{F2F_1}, p_1, \{FI2F_1 \to p_1, F2F_2 \to p_2\})$$

This rule spawns the service code in an asynchronous manner. Moreover the server state records the operation that activates the service. Finally, the services are persistent.

3.2 The Lambda Clouds

We consider a concurrent version of the call-by-value λ-calculus, called $\lambda^{\{\}}$ (*lambda clouds*), enriched with primitives for accessing resources, for declaring and enforcing security policies, and for installing services and managing their invocation. It is considered as an extension of $\lambda^{\square}$ introduced in [11] for concurrent systems. For simplicity, we assume that resources are objects already available in the cloud environment (i.e. resources cannot be dynamically created).

3.2.1 Syntax

Definition 3.2.1 (Syntax)**.** Let $\mathcal{V}$ be an infinite set of variables , ranged over by x, y, z, $\mathcal{R}$ be a finite set of resources, ranged over by r, r', A be a finite set of monadic actions, ranged over by α, β and Π be a finite repository of public service names, ranged over by π. A set Ev of access events is defined as $\{\alpha(r) | \alpha \in A$ and $r \in \mathcal{R}\}$. We define $Ev^* = Ev \cup \{link_\pi, invoke_\pi | \pi \in \Pi\}$. We assume a set Φ of local policies, ranged over by φ, φ', and local policies are defined as a regular safety properties over Ev^*. We call $\eta, \eta' \in Ev^*$ *histories*, i.e. finite sequences of events $\alpha_1(r_1)\alpha_2(r_2)\ldots\alpha_n(r_n)$. The syntax of $\lambda^{\{\}}$-expressions, ranged over by e, e', is defined by the following grammar:

$$\begin{array}{lcll}
e, e' & ::= & & \textit{expressions} \\
 & | & x & \text{variable} \\
 & | & \alpha(r) & \text{access event} \\
 & | & \lambda_z x.\ e & \text{abstraction} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & e_1\ \|\ e_2 & \text{parallel composition} \\
 & | & \text{link}\ \ \text{e} & \text{link component (service constructor)} \\
 & | & \varphi[e] & \text{policy framing}
\end{array}$$

The values v, v' of the calculus are variables and lambda abstractions, i.e.

$$\begin{array}{lcll}
v, v' & ::= & & \textit{values} \\
 & | & \mathbf{x} & \text{variable} \\
 & | & \lambda_z\ x.\ e & \text{abstraction}
\end{array}$$

Write $\mathbf{0}$ for a fixed, closed and event-free value. We assume all the standard notions and definitions of λ-calculus. Free and bound variables are defined in the standard way. An expression is closed if it contains no free variable. We denote the set of closed expressions as T_0 and the set of all expressions as T.

As in $\lambda^{[]}$, an access event $\alpha(r) \in Ev$ describes the application of the action α on the target resource r. A policy framing $\varphi[e]$ defines the scope of the policy φ to be enforced during the evaluation of e. Security policies φ are modelled as regular safety properties of event histories, i.e. properties that are recognizable by a usage automaton. The idea of usage automata is to describe *bad* usages. A history η satisfies a security policy φ, written $\eta \models \varphi$, if η does not lead the corresponding usage automaton to the offending states. The parallel composition $\|$ allows us to handle concurrency. The constructor *link e* is a new construct, introduced here in order to model the dynamic publication of a service whose code is e. Note that resources are not dynamically created in our approach.

Example 3.2.2. To prevent users from performing *bad commands* by exploiting SQL bugs in the database service `Q`, as introduced above, we require that the service

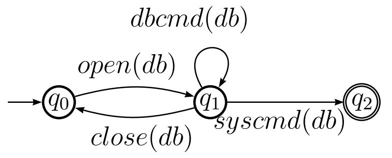

Figure 3.1: Usage automaton of the service Q

.

runs within the scope of the policy ϕ_{DB}. The corresponding automaton is described in Fig. 3.1. Intuitively, the policy constrains the database service to behave as follows. Firstly, the database service can open a connection to database by issuing the event $open(db)$. Then, it can performs any number of database commands, modelled by the event $dbcmd(db)$. Finally, it closes the connection to database by the event $close(db)$. In addition to these events, we use the event $syscmd(db)$ to model actions that perform system commands. To detect possible violations, we use a single offending final state (marked by a double circle), into which automaton can be driven by the event $syscmd(db)$.

3.2.2 Operational Semantics

We model a cloud server as a pool of services and computational resources running over a variety of virtual machines. Formally, a cloud server and its corresponding configuration are a triple of the form

$$(\eta, e, \sigma) \quad \text{where}$$

• $\eta \in Ev^*$ is the history representing the global cloud state, that details the dependencies among services and resources, as well as virtual machine configurations,
• $e \in \lambda^{\{\}}\text{-}Terms$ is the expression that describes the set of active processes, and
• $\sigma \in \Sigma$ is a mapping, also called the service environment, that maps each service name $\pi_i \in \Pi$ to the expression (script) used to load the virtual machine and the resources required to run the service $e \in T$.

The behaviour of $\lambda^{\{\}}$-expressions, described in Fig. 3.2, is defined through a small step operational semantics, called *cloud semantics*. A transition $(\eta, e, \sigma) \xrightarrow{\mu} (\eta', e', \sigma')$, indicates that, starting from a state described by the history η, the expression e evolves to e', issuing an event labelled by μ, possibly extending the history to η', and the service environment to σ'. Initial configurations have the form (ϵ, e, σ), where ϵ denotes the empty history. Transition labels μ are τ, access events $\alpha(r)$, service creations $link_\pi$ and service invocations $invoke_\pi$.

As in $\lambda^{[]}$, The rule [event] describes the evaluation of an event $\alpha(r)$ that consists in extending the current history with the event itself, and producing the empty value $\mathbf{0}$. Rules [app$_0$], [app$_1$] and [app$_2$] are standard rules of the call-by-value semantics of λ-calculus. Notice that the whole function body $\lambda_z x.e$ replaces the self variable z after the parameter substitution, so giving an explicit copy-rule semantics for recursive functions. The policy framing $\varphi[e]$ enforces the policy φ on the expression e, meaning that the history must respect φ at each step of the evaluation of e and each event issued within e must be checked against φ. When e is just a value, the security policy is simply removed as in [pol$_1$].

The rule [link] requires, from the repository, a free service name π to bind to the code e. The result of the evaluation of this transition is the empty value. Moreover, the event $link_\pi$ is issued and appended to the current history η, thus modelling the binding of the service. Notice that the side condition on the service repository Π ensures the uniqueness of the binding: the same service name cannot bind different service codes. By "π available", we mean that the service name π has not been used, i.e. $\pi \notin \Pi \cap dom(\sigma)$. Alternatively, one could define a notion of well-formed expressions requiring constrains on semantic of expressions in order to avoid captures of names. Also, note that our binding construct does not require the introduction of alpha-conversion. The addition of a new service $\{\pi \to e\}$ into the service environment σ, written by $\sigma[\pi \to e]$, where $\pi \notin dom(\sigma)$, is a further side effect of the rule. Formally, $\sigma[\pi \to e](\pi) = e$ and $\sigma[\pi \to e](\pi') = \sigma(\pi')$ if $\pi' \neq \pi$. Back to the service for converting formats, where the conversion service is activated by the transition:

$$(\epsilon, \mathbf{0}, \{F2F_1 \to p_1, F2F_2 \to p_2\}) \xrightarrow{invoke_{F2F_1}} (invoke_{F2F_1}, p_1, \{FI2F_1 \to p_1, F2F_2 \to p_2\})$$

Note that the initial empty process $\mathbf{0}$ indicates that the system waits for the user calling the service. After the call, the system performs an invocation action $invoke\ F2F_1$ and the conversion starts.

The rule [inv] deals with asynchronous interactions of clients and describes the evaluation of service requests. To manage a service invocation through the service name π, the cloud server spawns the code e associated with π. Moreover, the event $invoke_\pi$ is appended to the current history η. In our framework, rule [inv] allows us to indirectly model client invocation, through the occurrences of asynchronous events on the server side. Our treatment of service invocation, that is an original feature of our approach, has the consequent benefit to manage a variety of clients, by abstracting from the specific interaction protocols established with the cloud. Our semantic framework handles services as *persistent* entities. Services are not consumed by an invocation: they remain in the service environment. Alternatively, one could have introduced a volatile variant, in which the service is removed from the service environment, after service invocation, by modifying the rule [inv]. Furthermore, it is possible to encode volatile services by wrapping their invocations within a security policy preventing duplicate of the invocation event of each service.

$$\text{[event]} \qquad (\eta, \alpha(r), \sigma) \xrightarrow{\alpha(r)} (\eta.\alpha(r), \mathbf{0}, \sigma)$$

$$\text{[app}_0\text{]} \qquad (\eta, (\lambda_z x.e\ v), \sigma) \xrightarrow{\tau} (\eta, e[\lambda_z x.e/z, v/x], \sigma)$$

$$\text{[app}_1\text{]} \qquad \frac{(\eta, e_1, \sigma) \xrightarrow{\mu} (\eta', e_1', \sigma')}{(\eta, e_1\ e_2, \sigma) \xrightarrow{\mu} (\eta', e_1'\ e_2, \sigma')}$$

$$\text{[app}_2\text{]} \qquad \frac{(\eta, e_2, \sigma) \xrightarrow{\mu} (\eta', e_2', \sigma')}{(\eta, v\ e_2, \sigma) \xrightarrow{\mu} (\eta', v\ e_2', \sigma')}$$

$$\text{[par}_0\text{]} \qquad \frac{(\eta, e_0, \sigma) \xrightarrow{\mu} (\eta', e_0', \sigma')}{(\eta, e_0 \parallel e_1, \sigma) \xrightarrow{\mu} (\eta', e_0' \parallel e_1, \sigma')}$$

$$\text{[par}_1\text{]} \qquad \frac{(\eta, e_1, \sigma) \xrightarrow{\mu} (\eta', e_1', \sigma')}{(\eta, e_0 \parallel e_1, \sigma) \xrightarrow{\mu} (\eta', e_0 \parallel e_1', \sigma')}$$

$$\text{[pol}_0\text{]} \qquad \frac{(\eta, e, \sigma) \xrightarrow{\mu} (\eta', e', \sigma') \ \wedge\ \eta, \eta' \models \varphi}{(\eta, \varphi[e], \sigma) \xrightarrow{\mu} (\eta', \varphi[e'], \sigma')}$$

$$\text{[pol}_1\text{]} \qquad \frac{\eta \models \varphi}{(\eta, \varphi[v], \sigma) \xrightarrow{\tau} (\eta, v, \sigma)}$$

$$\text{[link]} \qquad \frac{\pi \in \Pi, \pi \text{ available}}{(\eta, \mathit{link}\ e, \sigma) \xrightarrow{\mathit{link}_\pi} (\eta.\mathit{link}_\pi, \mathbf{0}, \sigma[\pi \to e])}$$

$$\text{[inv]} \qquad \frac{\pi \in \mathit{domain}(\sigma)\ \mathit{and}\ \sigma(\pi) = e'}{(\eta, e, \sigma) \xrightarrow{\mathit{invoke}_\pi} (\eta.\mathit{invoke}_\pi, e \parallel e', \sigma)}$$

Figure 3.2: Cloud Semantics

3.3 Abstract Semantics for Clouds

In this section, we introduce the notion of abstract semantics for our framework, by resorting to the notion of bisimilarity.

Applicative bisimulation [3] provides the suitable abstract machinery for semantic reasoning, but not sufficient to deal with the peculiar features of our framework. Basically, the idea behind applicative bisimulation is that in order to reason on the equivalence of two functions, we need to know whether their behaviours are the same with all possible closed values. As a consequence, applicative bisimulation relies on the output generated by functions, hence it does not capture the events issued by our functions. To clarify this point with an example, let us consider the following cloud servers.

$$(\eta, \alpha; \lambda x.x, \sigma)$$
$$(\eta, \lambda x.x, \sigma)$$

For ease of writing, we often use $e; e'$ to denote $(\lambda_z x.e')e$, where x is not a free name in e' and α to denote $\alpha(r)$. It is easy to see that the two services in the clouds yield the same output. However, while the former service, during its execution, issues an event α and changes its history, the latter does not.

The management of events is crucial in our framework. By definition, service behaviour is indeed *history-dependent*, i.e. an expression may be executed differently when plugged within different cloud states.

Now, let us consider the cloud servers: $(\eta, \beta; \alpha; \varphi[\gamma], \sigma)$ and $(\eta, \alpha; \beta; \varphi[\gamma], \sigma)$, where η contains neither α nor β, and the policy φ states that the sequence $\alpha\beta$ is *not* allowed. After two transitions, the first configuration can make a transition that issues γ, while the second cannot, as illustrated below:

$$\begin{array}{l}(\eta, \beta; \alpha; \varphi[\gamma], \sigma) \xrightarrow{\beta} (\eta.\beta; \alpha; \varphi[\gamma], \sigma) \xrightarrow{\alpha} (\eta.\beta.\alpha; \varphi[\gamma], \sigma) \xrightarrow{\gamma} (\eta.\beta.\alpha.\gamma; \varphi[\mathbf{0}], \sigma) \\ (\eta, \alpha; \beta; \varphi[\gamma], \sigma) \xrightarrow{\alpha} (\eta.\alpha; \beta; \varphi[\gamma], \sigma) \not\rightarrow\end{array}$$

We need to take into account this feature when extending the applicative bisimulation notion. Furthermore, we need to understand when two configurations can be considered equivalent according to their service environments. Let us consider two configurations. A naive solution could be that the two configurations contain the same service environment. Nevertheless, this definition would prevent us from reasoning about processes of updating and maintaining services, which is a key feature in cloud computing. It is desirable that if upon a client request one server activates a service code in its service environment and makes a transition, the other server should make the same transition, producing the same side effect and reaching an equivalent process. To obtain this, the two service environments must contain equivalent service codes. In addition, to make sure that two configurations can make the same transitions, they must agree on service names.

We are now ready for defining cloud bisimulation. The definition of applicative bisimulation is originally introduced due to Abramsky [3] Here we adopted the variation of applicative bisimulation introduced by Sangiorgi [98]. In the following, we denote by $\eta\uparrow = \{\eta.\eta' | \eta' \in Ev^*\}$ the upward-closure of the history η.

Definition 3.3.1 (Cloud Simulation). A relation on R_H over $T_0 \times \Sigma$ is a *cloud simulation* w.r.t. a set of histories $H = \eta_o \uparrow$ for some η_o if whenever $(e, \sigma) R_H (d, \varsigma)$ then for every history $\eta \in H$,

(1) if $e = \mathbf{0}$ then $(\eta, d, \varsigma) \xrightarrow{\tau}^* (\eta, \mathbf{0}, \varsigma)$,

(2) if $e = \lambda_z x.e'$, then $(\eta, d, \varsigma) \xrightarrow{\tau}^* (\eta, \lambda_z x.d', \varsigma)$ s.t. $(\lambda_z x.e', \sigma) R_H (\lambda_z x.d', \varsigma)$ and for any value v, $(e'[\lambda_z x.e/z, v/x], \sigma) R_H (d'[\lambda_z x.d'/z, v/x], \varsigma)$,

(3) if $(\eta, e, \sigma) \xrightarrow{\tau} (\eta, e', \sigma)$ then $(e', \sigma) R_H (d, \varsigma)$,

(4) if $(\eta, e, \sigma) \xrightarrow{\alpha(r)} (\eta.\alpha(r), e', \sigma)$, where $\alpha \notin \{link_\pi \cup invoke_\pi | \pi \in \Pi\}$, then there exists d' s.t. $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha(r)} (\eta.\alpha, d', \varsigma)$ and $(e', \sigma) R_{H'} (d', \varsigma)$, where $H' = \eta.\alpha(r) \uparrow$,

(5) if $e = link\ e'$ and $(\eta, e, \sigma) \xrightarrow{link_\pi} (\eta.link_\pi, \mathbf{0}, \sigma[\pi \to e'])$, then there exist d', d'' such that $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} \xrightarrow{link_\pi} (\eta.link_\pi, d'', \varsigma[\pi \to d'])$, and $(\mathbf{0}, \sigma[\pi \to e'])$ $R_{H'} (d'', \varsigma[\pi \to d'])$, where $H' = \eta.link_\pi \uparrow$,

(6) if $(\eta, e, \sigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, e \parallel \sigma(\pi), \sigma)$ then we have $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} \xrightarrow{invoke_\pi}$ $(\eta.invoke_\pi, d \parallel \varsigma(\pi), \varsigma)$ and $((e \parallel \sigma(\pi)), \sigma) R_{H'} ((d \parallel \varsigma(\pi)), \varsigma)$, where $H' = (\eta.invoke_\pi) \uparrow$,

(7) $dom(\sigma) = dom(\varsigma)$ and $\forall \pi \in dom(\sigma)$, $(\sigma(\pi), \sigma) R_H (\varsigma(\pi), \varsigma)$.

where $\xrightarrow{\tau}{}^{*}$ means zero or more τ transitions. We write $\lesssim_H$ for the union of all cloud simulations w.r.t. to a set of histories H. If H is the set of all histories, then we call it cloud similarity and simply write $\lesssim$ for it.

Remark 3.3.2. Note that events of service invocation and creation do not involve resources, therefore the clauses (5) and (6) are handled without mentioning the resource in the transition label. In the contrary, resource names in transition labels of the clause (4) are essential since they are needed to define operations over resources, i.e. access events.

The first clause (1) ensures that if e can produce an empty value, then d can do the same, while the second clause (2) is a variant of applicative simulation. In the third clause (3), e evolves to e' by performing an internal transition τ, which does not change the history and after that e' remains equivalent to d. The forth clause (4) states that d can generate whatever e can, i.e. if e performs an action α that possibly changes the history into $\eta.\alpha$, then d can perform the same action α after zero or more internal transitions τ^* and generate the same history $\eta.\alpha$.

The clauses (5) and (6), for creating and invoking a service, basically guarantee the equivalence of two service environments at runtime. Finally, the clause (7) ensures that two service environments contain *equivalent* services.

Note that our definition of bisimilarity requires to check the conditions for all η in the upward closure of η_0. This amounts to an infinite number of checks. Therefore our notion can make it difficult to develop effective verification techniques. We plan to address this issue by adopting symbolic techniques like the ones developed in the field of software model checking.

It is easy to prove that the relation with respect to a set of histories H is included in the one obtained with respect to one of its subsets $H' = \eta \uparrow$, where $\eta \in H$.

Lemma 3.3.3. *Let $e, d \in T_0$ and $\sigma, \varsigma \in \Sigma$. If $H' = \eta \uparrow$, where $\eta \in H$, and $(e, \sigma) R_H (d, \varsigma)$, then $(e, \sigma) R_{H'} (d, \varsigma)$.*

Definition 3.3.4. Let $e, d \in T_0$ and $\sigma, \varsigma \in \Sigma$. We say that (d, ς) *cloud-simulates* (e, σ) w.r.t. a set of histories $H = \eta_o \uparrow$ for some η_o if there exists a cloud simulation R_H s.t. $(e, \sigma) R_H (d, \varsigma)$.

Definition 3.3.5 (Cloud Bisimulation)**.** A binary relation R_H on $T_0 \times \Sigma$, where $H = \eta_o \uparrow$ for some η_o, is *cloud bisimulation* if both R_H and its converse R^{-1} are cloud simulations w.r.t. a set of histories H. We write $\sim_H$ for the union of all cloud bisimulations w.r.t. a set of histories H. If H is the set of all histories, then we call it cloud bisimilarity and simply write $\sim$ for it.

Now we show that bisimulation is a congruence relation using Howe's method [61]. The idea is based on the construction of an auxiliary relation called the *precongruence candidate* $\hat{R}$ in terms of the preorder R which we need to prove a precongueence. It is possible to prove indeed that the preorder is a precongruence if and only if it coincides with the precongruence candidate. The precongruence candidate $\hat{R}$ is a precongruence that contains R, and that is preserved by language constructors. A key property of $\hat{R}$ is that if R is a bisimulation then $\hat{R}$ is a bisimulation, as well. Consequently, if we can show that the precongruence candidate of bisimilarity (union of all bisimulations) is a bisimulation, then bisimilarity and its precongruence candidate coincide, and due to the congruence of the precongruence candidate, bisimilarity is a congruence.
In our setting, we need to show that the precongruence candidate of cloud bisimulation is also a bisimulation. To prove that $\widehat{\sim}$ is indeed a bisimulation, we need to show that $\widehat{\sim}$ is preserved by computation, i.e. it is preserved under substitution and by tau actions, event actions and abstractions.

In the following, we present the definition and the lemma that we need for presenting our precongruence candidate. We first extend relations on closed terms to open terms, by substituting closed terms for variables.

Definition 3.3.6. Let R be a binary relation over $T_0 \times \Sigma$. The binary relation R^o over $T \times \Sigma$ is the extension of R to open expressions in $T \times \Sigma$, called *open* extension, is defined as follows: $(e, \sigma) R^o (e', \sigma')$ if $(\gamma(e), \sigma) R (\gamma(e'), \sigma')$ for every closing substitution γ.

Definition 3.3.7 (Precongruence Candidate)**.** Given a preorder R over $T_0 \times \Sigma$, we define the *precongruence candidate* $\widehat{R}$ over $T \times \Sigma$, denoted by $(e, \sigma) \widehat{R} (e', \varsigma)$, for $e, e' \in T$, by induction on the size of e.

- for each variable x, if $(x, \sigma) R^o (e, \varsigma)$ then $(x, \sigma) \widehat{R} (e, \varsigma)$;
- for each resource r, if $(r, \sigma) R^o (e, \varsigma)$ then $(r, \sigma) \widehat{R} (e, \varsigma)$;

- for each event α, if $(\alpha, \sigma)R^o(e, \varsigma)$ then $(\alpha, \sigma)\widehat{R}(e, \varsigma)$;
- for the empty value $\mathbf{0}$, if $(\mathbf{0}, \sigma)R^o(e, \varsigma)$ then $(\mathbf{0}, \sigma)\widehat{R}(e, \varsigma)$;
- for $e_1, e_2, e_1', e_2' \in T$, if $(e_1, \sigma)\widehat{R}(e_1', \varsigma)$, $(e_2, \sigma)\widehat{R}(e_2', \varsigma)$ and $((e_1'\ e_2'), \sigma)R^o(e, \varsigma)$, then $((e_1\ e_2), \sigma)\widehat{R}(e, \varsigma)$.
- for $e_1, e_1' \in T$, if $(e_1, \sigma)\widehat{R}(e_1', \varsigma)$ and $((\lambda x\ e_1'), \sigma)R^o(e, \varsigma)$, then $((\lambda x\ e_1), \sigma)\widehat{R}(e, \varsigma)$.

Now we prove some properties of the candidate precongruence of a given preorder R over $T_0 \times \Sigma$, needed to provide, in turn, that it is a bisimulation. First, we show that the candidate precongruence $\widehat{R}$ is reflexive, i.e. $(e, \sigma)\widehat{R}(e, \sigma)$, for all $(e, \sigma) \in T \times \Sigma$. Another important property of $\widehat{R}$ is *constructor respecting*, that is, $\widehat{R}$ is preserved by language constructors. The next property says that $\widehat{R}$ includes the open extension R^o of R with open expressions. The last property shows how to relate elements of $\widehat{R}$ by using relations R^o.

Lemma 3.3.8. *Let R be a preorder over $T_0 \times \Sigma$, then the following hold:*

1. *$\widehat{R}$ is reflexive.*
2. *$\widehat{R}$ is constructor respecting, i.e.*
   - *if $(e, \sigma)\widehat{R}(e', \varsigma)$, then $((\lambda x\ e), \sigma)\widehat{R}((\lambda x\ e'), \varsigma)$.*
   - *if $(e_1, \sigma)\widehat{R}(e_1', \varsigma)$ and $(e_2, \sigma)\widehat{R}(e_2', \varsigma)$, then $((e_1\ e_2), \sigma)\widehat{R}((e_1'\ e_2'), \varsigma)$.*
   - *$(e, \sigma)\widehat{R}(e', \varsigma)$, then $(link\ e, \sigma)\widehat{R}(link\ e', \varsigma)$.*
   - *$(e, \sigma)\widehat{R}(e', \varsigma)$, then $(\varphi[e], \sigma)\widehat{R}(\varphi[e'], \varsigma)$.*
3. *$R^o \subseteq \widehat{R}$.*
4. *If we have $(e, \sigma)\widehat{R}(e', \sigma')$ and $(e', \sigma')R^o(e'', \sigma'')$, then $(e, \sigma)\widehat{R}(e'', \sigma'')$.*

Proof. 1. By induction on term size, by definition of $\widehat{R}$ and reflexivity of R.

2. Suppose that $(e_1, \sigma)\widehat{R}(e_1', \varsigma)$ and $(e_2, \sigma)\widehat{R}(e_2', \varsigma)$. By reflexivity of R, we have $((\lambda x\ e_1), \varsigma)R^o((\lambda x\ e_2'), \varsigma)$, $((e_1\ e_2), \varsigma)R^o((e_1'\ e_2')), \varsigma)$, $(link\ e_1, \varsigma)R^o(link\ e_2', \varsigma)$. Then by definition of $\widehat{R}$, the property follows immediately.

3. We show that if $(e, \sigma)R^o(e', \sigma')$, then $(e, \sigma)\widehat{R}(e', \sigma')$, by induction on term e,
   Case: e is a variable, an event, resource or empty process: it holds by definition of $\widehat{R}$.
   Case: e is a term of form $(\lambda x\ e_1)$: by (1), we have $(e_1, sigma)\widehat{R}(e_1, \sigma)$. By definition of $\widehat{R}$, $(e, \sigma)R^o(e', \sigma')$.
   Case: e is a term of form $(e_1\ e_2)$: by (1), we have $(e_1, \sigma)\widehat{R}(e_1, \sigma)$ and $(e_2, \sigma)\widehat{R}e_2, \sigma)$

. By definition of $\widehat{R}$, $(e,\sigma)R^o(e',\sigma')$.
Case: e is a term of form $(\mathit{link}\ e_1)$: by (1), we have $(e_1,\sigma)\widehat{R}(e_1,\sigma)$. By definition of $\widehat{R}$, $(e,\sigma)R^o(e',\sigma')$.
Case: e is a term of form $(\varphi[e_1])$: by (1), we have $(e_1,\sigma)\widehat{R}(e_1,\sigma)$. By definition of $\widehat{R}$, $(e,\sigma)R^o(e',\sigma')$.

4. We proceed by induction on term e and transitivity of R:
Case: e is a variable x: by definition of $\widehat{R}$, $(x,\sigma)R^o(e',\sigma')$. By transitivity of R, we have $(x,\sigma)R^o(e'',\sigma'')$. The result follows immediately .
Case: e is an event, resource or empty process: similarly.
Case: e is a term $(\lambda x\ e_1)$: there exists (e_1',σ') such that $(e_1,\sigma)\widehat{R}(e_1',\sigma')$ and $((\lambda x\ e_1'),\sigma')R^o((\lambda x\ e_1'),\sigma')$. By transitivity of R, $((\lambda x\ e_1'),\sigma')R^o(e'',\sigma'')$. The result follows by definition of $\widehat{R}$.
Case: e is a term $(e_1\ e_2)$: there exist $(e_1',\sigma'),(e_2',sigma')$ such that $(e_1,\sigma)\widehat{R}(e_1',\sigma')$, $(e_2,\sigma)\widehat{R}(e_2',\sigma')$ and $(e_1'\ e_2',\sigma')R^o(e',\sigma')$. By transitivity of R, $(e_1'\ e_2',\sigma')R^o(e'',\sigma'')$. The result follows by definition of $\widehat{R}$.
Case: e is a term $\mathit{link}\ e_1$: there exists (e_1',σ') such that $(e_1,\sigma)\widehat{R}(e_1',\sigma')$ and $(\mathit{link}\ e_1',\sigma')R^o(\mathit{link}\ e_1',\sigma')$. By transitivity of R, $(\mathit{link}\ e_1',\sigma')R^o(e'',\sigma'')$. The result follows by definition of $\widehat{R}$.
Case: e is a term $\varphi[e_1]$: there exists (e_1',σ') such that $(e_1,\sigma)\widehat{R}(e_1',\sigma')$ and $(\varphi[e_1'],\sigma')R^o(\varphi[e_1'],\sigma')$. By transitivity of R, $(\varphi[e_1'],\sigma')R^o(e'',\sigma'')$. The result follows by definition of $\widehat{R}$.

□

Lemma 3.3.9. *Let $e,d\in T_0$ and $\sigma,\varsigma\in\Sigma$. If $H'\subseteq H$ and $(e,\sigma)R_H(d,\varsigma)$, then $(e,\sigma)R_{H'}(d,\varsigma)$.*

Proof. Straightforward. □

We now state the congruence theorem with the main auxiliary lemmata. First, we prove that the service environments play a minor role in the open extension of the cloud bisimulation. Intuitively, service environments contain *equivalent* service codes.

Lemma 3.3.10. *Let $e,e',d\in T$ and $\sigma,\sigma'\in\Sigma$.*

- *If $(e,\sigma)\sim_H^o(e',\sigma')$ then $(e,\sigma)\sim_H^o(e',\sigma)$.*
- *If $(e,\sigma)\widehat{\sim}_H(e',\sigma')$ then $(e,\sigma)\widehat{\sim}_H(e',\sigma)$.*

Proof. Straightforward by induction on term size. □

Recall that to prove that $\widehat{\sim}$ is a bisimulation, we need to show that $\widehat{\sim}$ is preserved by computation. As computation in the λ-Cloud calculus typically involves substitutions, it requires that $\widehat{\sim}$ of $\sim$ is preserved under substitutions. This is formulation of the below substitution lemma. The remaining lemma shows that $\widehat{\sim}$ is preserved by computation, that is, by τ actions, event actions, applications, service creations and service invocations. Recall that notation $\sim_H$ is used to denote bisimilarity with respect to a set H of histories, while $\sim$ denotes the case with respect to all sets of histories.

Lemma 3.3.11 (Substitution). *Let $e_1, e'_1, e_2, e'_2 \in T$ and $\sigma, \sigma' \in \Sigma$. If $(e_1, \sigma)\widehat{\sim}_H(e'_1, \sigma')$ and $(e_2, \sigma)\widehat{\sim}_H(e'_2, \sigma')$ then we have $(e_2[e_1/x], \sigma)\widehat{\sim}_H\ (e'_2[e'_1/x], \sigma')$.*

Proof. By induction on the size of term e_2
Case: e_2 is a variable x: By the lemma 3.3.8, we have $\sim^o_H \subseteq \widehat{\sim}_H$. The fact that x is a variable and $(x, \sigma)\widehat{\sim}_H(e'_2, \sigma')$ imply that $(x, \sigma)\sim^o_H(e'_2, \sigma')$, and therefore

$$(e'_1, \sigma)\sim^o_H(e'_2[e'_1/x], \sigma')$$

by definition of $\sim^o_H$.

By previous lemma, we have $(e_1, \sigma)\widehat{\sim}_H(e'_1, \sigma)$. By property (4):

$$\begin{array}{l}(e_1, \sigma)\widehat{\sim}_H(e'_1, \sigma)\\ (e'_1, \sigma)\sim^o_H(e'_2[e'_1/x], \sigma'),\end{array}$$

hence we have $(x[e_1/x], \sigma)\sim^o_H(e'_2[e'_1/x], \sigma')$.
Case: e_2 is a variable $y \neq x$: similarly
Case: e_2 is an event or empty process: similarly

Case: e_2 is a term $(\lambda x\ e)$: since $((\lambda x\ e), \sigma)\widehat{\sim}_H(e'_2, \sigma')$, for there exists e' s.t $(e, \sigma)\widehat{\sim}_H(e', \sigma')$ and $((\lambda x\ e'), \sigma')\sim_H{}^o(e'_2, \sigma')$. By induction hypothesis,

$$\begin{array}{l}(e[e_1/x], \sigma)\widehat{\sim}_H(e'[e'_1/x], \sigma')\\ ((\lambda x\ e')[e'_1/x], \sigma')\sim^o_H(e'_2[e'_1/x], \sigma')\end{array}$$

so $(e_2[e_1/x], \sigma)\widehat{\sim}_H(e'_2[e'_1/x], \sigma')$

Case: e_2 is a term $(e_3\ e_4)$: since $((e_3\ e_4), \sigma)\widehat{\sim}_H(e'_2, \sigma')$, there exist e'_3, e'_4 s.t $(e_3, \sigma)\widehat{\sim}_H(e'_3, \sigma')$, $(e_4, \sigma)\widehat{\sim}_H(e'_4, \sigma')$ and $((e'_3\ e'_4), \sigma')\sim_H{}^o(e'_2, \sigma')$. By induction hypothesis,

$$\begin{array}{l}(e_3[e_1/x], \sigma)\widehat{\sim}_H(e'_3[e'_1/x], \sigma')\\ (e_4[e_1/x], \sigma)\widehat{\sim}_H(e'_4[e'_1/x], \sigma')\\ ((e'_3\ e'_4)[e'_1/x], \sigma')\sim^o_H(e'_2[e'_1/x], \sigma')\end{array}$$

so $(e_2[e_1/x],\sigma)\widehat{\sim}_H(e_2'[e_1'/x],\sigma')$

Case: e_2 is a term $(\mathit{link}\ e)$: since $((\mathit{link}\ e),\sigma)\widehat{\sim}_H(e_2',\sigma')$, for there exists e' s.t $(e,\sigma)\widehat{\sim}_H(e',\sigma')$ and $((\mathit{link}\ e'),\sigma')\sim_H{}^o(e_2',\sigma')$. By induction hypothesis,

$$\begin{array}{l}(e[e_1/x],\sigma)\widehat{\sim}_H(e'[e_1'/x],\sigma')\\((\mathit{link}\ e')[e_1'/x],\sigma')\sim_H^o(e_2'[e_1'/x],\sigma')\end{array}$$

so $(e_2[e_1/x],\sigma)\widehat{\sim}_H(e_2'[e_1'/x],\sigma')$

Case: e_2 is a term $\varphi[e]$: since $(\varphi[e],\sigma)\widehat{\sim}_H(e_2',\sigma')$, for there exists e' s.t $(e,\sigma)\widehat{\sim}_H(e',\sigma')$ and $(\varphi[e'],\sigma')\sim_H{}^o(e_2',\sigma')$. By induction hypothesis,

$$\begin{array}{l}(e[e_1/x],\sigma)\widehat{\sim}_H(e'[e_1'/x],\sigma')\\(\varphi[e'][e_1'/x],\sigma')\sim_H^o(e_2'[e_1'/x],\sigma')\end{array}$$

so $(e_2[e_1/x],\sigma)\widehat{\sim}_H(e_2'[e_1'/x],\sigma')$

□

Lemma 3.3.12 (Tau actions). *Let $e,d\in T_0$, $\sigma,\varsigma\in\Sigma$ and $(e,\sigma)\widehat{\sim}_H(d,,\varsigma)$. For every history η if $(\eta,e,\sigma)\xrightarrow{\tau}(\eta,e',\sigma)$, then $(e',\sigma)\widehat{\sim}_H(d,\varsigma)$.*

Proof. By induction on derivation of $(\eta,e,\sigma)\xrightarrow{\tau}(\eta,e',\sigma)$:
Case of [APP$_0$] rule: $e=(\lambda_z x.e_1)e_2:(\eta,(\lambda_z x.e_1)e_2)\xrightarrow{\tau}(\eta,e_1[\lambda_z x.e_1/z,e_2/x])$, where e_2 is a value. We need to show that $(e_1[\lambda_z x.e_1/z,e_2/x],\sigma)\widehat{\sim}_H(d,\varsigma)$.
Since $e\widehat{\sim}_H d$, w.l.o.g. there exist $d_1,d_2,\lambda_z x.e_1'\in T_0$ such that

$$\begin{array}{l}(\lambda_z x.e_1,\sigma)\widehat{\sim}_H(d_1,\varsigma)\\(e_2,\sigma)\widehat{\sim}_H(d_2,\varsigma)\\(e_1,\sigma)\widehat{\sim}_H(e_1',\varsigma)\\(\lambda x.e_1',\varsigma)\sim_H(d_1,\varsigma)\\(d_1\ d_2,\varsigma)\sim_H(d,\varsigma)\\d_2\text{ is a value.}\end{array}$$

Otherwise, by choosing a closing substitution for d_1,d_2 and e_1' and the definition of $(e_2,\sigma)\widehat{\sim}_H(d_2,\varsigma)$, we can obtain the desired result.
By Lemma 3.3.11, we have

$$(e_1[\lambda_z x.e_1/z,e_2/x],\sigma)\widehat{\sim}_H(e_1'[\lambda_z x.e_1'/z,d_2/x],\varsigma)\ (1)$$

By definition of $\sim$, we have $(\eta,d_1,\varsigma)\xrightarrow{\tau}{}^*(\eta,\lambda_z x.d_1',\varsigma)$ such that $(\lambda_z x.e_1',\sigma)\sim_H(\lambda_z x.d_1',\varsigma)$ and $(e_1'[\lambda_z x.e_1'/z,d_2/x],\sigma)\sim_H(d_1'[\lambda_z x.d_1'/z,d_2/x],\varsigma)$.

Since $(d_1\ d_2, \varsigma) \sim_H (d, \varsigma)$, we have

$$(d_1'[\lambda_z x.d_1'/z, d_2/x], \varsigma) \sim_H (d, \varsigma)\ (2)\ .$$

(1) and (2) implies that $(e_1[\lambda_z x.e_1/z, e_2/x], \sigma)\widehat{\sim}_H(d, \varsigma)$.

Other cases: we will consider [APP$_1$] rule. The proofs of the other rules are similar.

We have $e = e_1\ e_2$ and $(\eta, e_1, \sigma) \xrightarrow{\tau} (\eta, e_1', \sigma)$. Since $(e, \sigma)\widehat{\sim}_H(d, \varsigma)$, w.l.o.g. there exist $d_1, d_2 \in T_0$ such that

$$\begin{array}{l}(e_1, \sigma)\widehat{\sim}_H(d_1, \varsigma)\\ e_2, \sigma)\widehat{\sim}_H(d_2, \varsigma)\\ (d_1 d_2, \varsigma) \sim_H (d, \varsigma).\end{array}$$

By induction hypothesis, $(e_1', \sigma)\widehat{\sim}_H(d_1, \varsigma)$. Since $\widehat{\sim}_H$ is operator respecting, we have that

$$(e_1' e_2, \sigma)\widehat{\sim}_H(d_1 d_2, \varsigma).$$

This implies that $(e_1' e_2, \sigma)\widehat{\sim}_H(d, \varsigma)$. □

Lemma 3.3.13 (Applicative lemma). *Let $\lambda_z x.e, d \in T_0$, $\sigma, \varsigma \in \Sigma$ and $(\lambda_z x.e, \sigma)\widehat{\sim}_H (d, \varsigma)$. Then, for every history $\eta \in H$, there exists d' such that $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} (\eta, \lambda_z x.d', \varsigma)$, $(\lambda_z x.e, \sigma)\widehat{\sim}_H(\lambda_z x.d', \varsigma)$ and for any value v, $(e[\lambda_z x.d/z, v/x], \sigma)\widehat{\sim}_H (d'[\lambda_z x.d'/z, v/x], \varsigma)$.*

Proof. Let $\eta \in H$. By definition of $\widehat{\sim}_H$, w.l.o.g. there exists $\lambda_z x.c \in T_0$ such that

$$\begin{array}{l}(e, \sigma)\widehat{\sim}_H(c, \varsigma)\\ (\lambda_z x.c, \varsigma) \sim_H (d, \varsigma).\end{array}$$

By Lemma 3.3.11, we have that

$$e[\lambda_z x.e/z, v/x], \sigma)\widehat{\sim}_H(c[\lambda_z x.c/z, v/x], \varsigma)\ (1)$$

By definition of $\sim_H$, there exist d' such that

$$\begin{array}{l}(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} (\eta, \lambda_z x.d', \varsigma) \text{ for any value } v\\ (c[\lambda_z x.c/z, v/x], \varsigma) \sim_H (d'[\lambda_z x.d'/z, v/x], \varsigma)\ (2)\end{array}$$

By property 4, (1),(2) imply that $(e[\lambda_z x.d/z, v/x], \sigma)\ \widehat{\sim}_H(d'[\lambda_z x.d'/z, v/x], \varsigma)$. □

Lemma 3.3.14 (Event action). *Let $e, d \in T_0$, $\sigma, \varsigma \in \Sigma$ and $(e, \sigma)\widehat{\sim}_H(d, \varsigma)$. For every history $\eta \in H$ if $(\eta, e, \sigma) \xrightarrow{\alpha} (\eta.\alpha, e', \sigma')$, where $\alpha \notin \{link_\pi \cup invoke_\pi | \pi \in \Pi\}$, then there exists d' such that $(\eta, d, \varsigma) \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} (\eta.\alpha, d', \varsigma)$, $(e', \sigma)\widehat{\sim}_{H'}(d', \varsigma)$, where $H' = \eta.\alpha \uparrow$.*

Proof. By induction on derivation of $(\eta, e, \sigma) \xrightarrow{\alpha} (\eta', e', \sigma)$:
Case of [EVENT] rule: $e = \alpha$ and $(\eta, \alpha, \sigma) \xrightarrow{\alpha} (\eta, \mathbf{0}, \sigma)$
Since $(\alpha, \sigma)\widehat{\sim}_H(d, \varsigma)$, we have

$$(\alpha, \sigma) \sim_H (d, \varsigma)$$

By definition of $\sim_H$ if $(\eta, \alpha, \sigma) \xrightarrow{\alpha} (\eta.\alpha, \mathbf{0}, \sigma)$, then there exists $d' \in T_0$ such that

$$\begin{array}{l}(\eta, d, \varsigma) \xrightarrow{\alpha} (\eta.\alpha, d', \varsigma)\\ (\mathbf{0}, \sigma) \sim_{H'} (d', \varsigma) \text{ where } H' = \eta.\alpha \uparrow\end{array}$$

This implies that $(\mathbf{0}, \sigma)\widehat{\sim}_{H'}(d', \varsigma)$.

Other cases: we will consider [APP$_1$] rule. The proofs of the other rules are similar.

We have $e = e_1\ e_2$ and $(\eta, e_1, \sigma) \xrightarrow{\alpha} (\eta', e'_1, \sigma)$. Since $(e, \sigma)\widehat{\sim}_H(d, \varsigma)$, w.l.o.g. there exist $d_1, d_2 \in T_0$ such that

$$\begin{array}{l}(e_1, \sigma)\widehat{\sim}_H(d_1, \varsigma)\\ (e_2, \sigma)\widehat{\sim}_H(d_2, \varsigma)\\ (d_1 d_2, \varsigma) \sim_H (d, \varsigma)\end{array}$$

By induction hypothesis, there exists d'_1 such that

$$\begin{array}{l}(\eta, d_1, \varsigma) \xrightarrow{\alpha} (\eta', d'_1, \varsigma)\\ (e'_1, \sigma)\widehat{\sim}_{H'}(d'_1, \varsigma) \text{ where } H' = \eta' \uparrow\end{array}$$

Since

$$\begin{array}{l}\widehat{\sim}_{H'} \text{ is operator respecting and}\\ (e_2, \sigma)\widehat{\sim}_H(d_2, \varsigma) \text{ implies } (e_2, \sigma)\widehat{\sim}_{H'}(d_2, \varsigma),\end{array}$$

we have $(e'_1 e_2, \sigma)\widehat{\sim}_{H'}(d'_1 d_2, \varsigma)$.

Since $(\eta, d_1 d_2, \varsigma) \xrightarrow{\alpha} (\eta', d'_1 d_2, \varsigma)$, so there exists d' such that

$$\begin{array}{l}(\eta, d, \varsigma) \xrightarrow{\alpha} (\eta', d', \varsigma)\\ (d'_1 d_2, \varsigma) \sim_{H'} (d', \varsigma).\end{array}$$

It implies that $(e'_1 e_2, \sigma)\widehat{\sim}_{H'}(d', \varsigma)$. □

Lemma 3.3.15 (Resources and Empty-value)**.** *Let $r, d \in T_0$ and $\sigma, \varsigma \in \Sigma$. For every history $\eta \in H$:*

- *If $(r, \sigma)\widehat{\sim}_H(d, \varsigma)$, then for any $\eta \in H$, $(\eta, d, \varsigma) \xrightarrow{\tau}^{*} (\eta, r, \varsigma)$.*
- *If $(\boldsymbol{0}, \sigma)\widehat{\sim}_H(d, \varsigma)$, for any $\eta \in H$, $(\eta, d, \varsigma) \xrightarrow{\tau}^{*} (\eta, \boldsymbol{0}, \varsigma)$.*

Proof. Straightforward. □

The following lemma shows that $\widehat{\sim}$ is preserved by service creations.

Lemma 3.3.16 (Service Creation)**.** *Let link $e, d \in T_0$, $\sigma, \varsigma \in \Sigma$ and $(link\ e, \sigma)\widehat{\sim}_H$ (d, ς). For every history $\eta \in H$, $(\eta, link\ e, \sigma) \xrightarrow{link_\pi} (\eta.link_\pi, \boldsymbol{0}, \sigma[\pi \to e])$, then there exist d', d'' such that $(\eta, d, \varsigma) \overset{\tau}{\to}^* \xrightarrow{link_\pi} (\eta.link_\pi, d'', \varsigma[\pi \to d'])$, $(\boldsymbol{0}, \sigma[\pi \to e])R_{H'}$ $(d'', \varsigma[\pi \to d'])$, where $H' = \eta.link_\pi \uparrow$.*

Proof. Straightforward. □

Lemma 3.3.17 (environment)**.** *Let $e, d \in T_0$ $\sigma, \varsigma \in \Sigma$. If $(e, \sigma)\widehat{\sim}_H(d, \varsigma)$, then $dom(\sigma) = dom(\varsigma)$ and $\forall \pi \in dom(\sigma) : (\sigma(\pi), \sigma)\widehat{\sim}_H(\varsigma(\pi), \varsigma)$.*

Proof. By induction of term e.
Case e is a variable: we have $(e, \sigma) \sim_H (d, \varsigma)$. By the definition of $\sim_H$, it follows that $(\sigma(\pi), \sigma) \sim_H (\varsigma(\pi), \varsigma)$, hence $(\sigma(\pi), \sigma)\widehat{\sim}_H(\varsigma(\pi), \varsigma)$.
Case: e is an event, resource or empty process: similarly.
Case: e is a term of form $(\lambda x\ e')$: by the definition of $\widehat{\sim}_H$, there exist d' s.t. $(e', \sigma)\widehat{\sim}_H(d', \varsigma)$ and $((\lambda x\ d'), \sigma){\sim_H}^o(d, \varsigma)$. It follows that $(\sigma(\pi), \sigma){\sim_H}^o(\varsigma(\pi), \varsigma)$.
Case: e is a term of form $(e_1\ e_2)$: by the definition of $\widehat{\sim}_H$, there exist d_1, d_2 s.t. $(e_1, \sigma)\widehat{\sim}_H(d_1, \varsigma)$, $(e_2, \sigma)\widehat{\sim}_H(d_2, \varsigma)$, and $((d_1\ d_2), \sigma){\sim_H}^o(d, \varsigma)$. It follows that $(\sigma(\pi), \sigma){\sim_H}^o(\varsigma(\pi), \varsigma)$
Case: e is a term of form $(link e')$: by the definition of $\widehat{\sim}_H$, there exist d' s.t. $(e', \sigma)\widehat{\sim}_H(d', \varsigma)$ and $((link\ d'), \sigma){\sim_H}^o(d, \varsigma)$. It follows that $(\sigma(\pi), \sigma){\sim_H}^o(\varsigma(\pi), \varsigma)$.
Case: e is a term of form $\varphi[e']$: by the definition of $\widehat{\sim}_H$, there exist d' s.t. $(e', \sigma)\widehat{\sim}_H(d', \varsigma)$ and $(\varphi[d'], \sigma){\sim_H}^o(d, \varsigma)$. It follows that $(\sigma(\pi), \sigma){\sim_H}^o(\varsigma(\pi), \varsigma)$.
□

Lemma 3.3.18 (Invocation)**.** *Let $e, d \in T_0$ $\sigma, \varsigma \in \Sigma$ and $(e, \sigma)\widehat{\sim}_H(d, \varsigma)$. For every history $\eta \in H$, $(\eta, e, \sigma) \xrightarrow{invoke_\pi} (\eta.invoke_\pi, e \parallel \sigma(\pi), \sigma)$ then $(\eta, d, \varsigma) \overset{\tau}{\to}^* \xrightarrow{invoke_\pi}$ $(\eta.invoke_\pi, d \parallel \varsigma(\pi), \varsigma)$ and $(e \parallel \varsigma(\pi), \sigma)R_{H'}$ $(d \parallel \varsigma(\pi), \varsigma)$, with $H' = (\eta.invoke_\pi) \uparrow$.*

Proof. Straightforward. □

The following theorem is a direct consequence of the above lemmata.

Theorem 3.3.19 (Congruence)**.** *Cloud bisimulation $\sim$ is a congruence.*

Proof. immediate from the above lemmata. □

Lemma 3.3.20. $\forall e, \forall \varphi, \forall \sigma,\ \forall \sigma,\ (\varphi[e], \sigma) \lesssim (e, \sigma)$.

Proof. Consider a relation S:

$$S = \{((\varphi[e], \sigma), (e, \sigma)) | \forall e \in T_0 \wedge \forall \varphi\} \cup I_T,$$

where I_T is an identity relation on T_0. We need to show that S is a simulation. By induction on evaluation derivation of e. $\square$

This lemma is particularly useful because it ensures that instrumenting a program with a policy framing does not add new behavior of the program. As a consequence, any behavior that violates the policy on demand is prevented to occur. Our ultimate goal is to obtain a semantics-based methodology of safety refinement process in cycle of software development.

Corollary 3.3.21. *$\forall e, \forall \varphi, \forall \sigma$, $(\lambda_z x.\varphi[e], \sigma) \lesssim (\lambda_z x.e, \sigma)$.*

Example 3.3.22. Back to the storage service `Q` presented in the Introduction, we can specify `Q` as follows:

$$\begin{aligned} &e_{form} = \lambda x.\ e_{process}\ x \\ &e_{process} = \lambda y.\ open(db); (query\ db\ y); close(db), \end{aligned}$$

where e_{form} is the cloud service interface wrapping inside the database. The service gets a query string $\langle strquery \rangle$ from a user, then feeds it to $e_{process}$. In turn, the function $e_{process}$ takes the query y as a parameter, connects to the database db, makes a query to db, by exploiting an auxiliary function *query* with the database and the query string as parameters, then closes the database connection. We abstract from the details of the code of the *query* function here, we just assume that it may perform some internal activities and then issues the database command *dbcmd* and returns a value v. In the following, for ease of writing, we write (η, e) to denote (η, e, σ). The evolution of the service, starting from the initial state η is as follows:

$$\begin{aligned} &(\eta, e_{form}\ \langle strquery \rangle) \\ &\xrightarrow{\tau}\ (\eta, e_{process}\ \langle strquery \rangle) \\ &\xrightarrow{\tau}\ (\eta, open(db); ((query\ db\ strquery); close(db)) \\ &\xrightarrow{open(db)}\ (\eta.open(db), (query\ db\ strquery); close(db)) \\ &\xrightarrow{\tau}{}^{*} \xrightarrow{dbcmd} \xrightarrow{\tau}{}^{*}\ (\eta.open(db).dbcmd, v; close(db)) \\ &\xrightarrow{\tau}{}^{*} \xrightarrow{close(db)}\ (\eta.open(db).dbcmd.close(db), \mathbf{0})) \end{aligned}$$

As previously discussed, the service above is unsafe because it may contain a SQL injection bug: an attacker can try to inject a command in front of query string, e.g. $\langle syscmd; strquery \rangle$, and therefore can execute any dangerous command such as deleting a file, as illustrated by the following trace:

$$
\begin{aligned}
&(\eta, e_{form}\ \langle syscmd; strquery\rangle)\\
&\xrightarrow{\tau}\ (\eta, e_{process}\ \langle syscmd; strquery\rangle)\\
&\xrightarrow{\tau}\ (\eta, open(db); (query\ db\ (syscmd; strquery)); close(db))\\
&\xrightarrow{open(db)}\ (\eta.open(db), (query\ db\ (syscmd; strquery)); close(db))\\
&\xrightarrow{\mathbf{syscmd}} (\eta.open(db).syscmd, (query\ db\ (strquery)); close(db))
\end{aligned}
$$

To prevent system commands from being executed, we can instrument e_{form} by framing it with a security policy φ_{DB}, which does not allow execution of any *system command.* The corresponding usage automaton is depicted in Fig. 3.1, where *syscmd* denotes the generic system command. By applying our technical results, we can state that $\lambda x.\varphi_{DB}[e_{process}\ x] \prec \lambda x.e_{process}\ x$. The presence of φ_{DB} in $\lambda x.\varphi_{DB}[e_{process}\ x]$ excludes all generated sequences that contain system commands.

3.4 Related Works

Our work takes the approach presented in [11, 13] as starting point. In particular, we adopt their idea of history-based security and of resource usage. In this approach, security polices are defined in term of safety properties, which specify desirable behaviour of services. We make a step forward to introduce the parallel operator. In result, histories record a sequences of actions over resources generated by a set of services rather than a single services as in [13]. This is motivated by considering properties concerning global resource usage of the cloud systems rather than of individual services. The work in [18] uses the parallel operator to introduce a network of located services. However, each service has its own history and the focus of the work is on service orchestration.

Here we comment on related approaches exploiting extension of the λ-calculus to handle resource management. The concurrent λ-calculus, introduced in [22], is used to represent and compose services and virtual machines that run on them. This calculus presents indeed some similarities with ours. They also introduce a type system for avoiding troublesome configuration errors, that could be exploited for handling resources in the cloud. The work in [59] proposed a calculus, called **V**, with primitives to model basic virtualisation operations such as to start and stop Virtual Machines (VMs), and to read and write data in a hierarchical store. It is based on π-calculus rather than λ-calculus. The main idea of the calculus is to provide a formalism which enables programming and static analysing scripts that control virtual clusters or applications in modern distributed systems such as cloud systems. This approach has been originated from the idea of operation logic [23], whose characterisation is to specify properties of management scripts in data center. In our approach, side effects of functions enabling those scripts are captured, and they are monitored by usage polices. That is, by observing those side effects, policy structure ensures that *bad* behaviour never happens at runtime.

The authors in [50] introduced another security-based language. It exploits assertions that govern the boundaries between software building blocks such as procedures, classes or modules. A special kind of assertion, called *behavioural software contract*, which monitors the flow of values across component boundaries. In our approach, properties can be understood as assertions across states, whereas the assertions in this paper are a property of a single state. The work uses concurrency only for checking purposes, whereas ours use concurrency for programming purpose. The focus of the paper is mainly on implementation to reduce the run-time overhead. The work presented in [85] proposed an extension of the simply-typed λ-calculus with constructs for thread creation and monitor primitives with synchronised expressions. In this work, a strict access control is described in terms of automata, which is similar to ours. However, by equipping synchronisation mechanism, threads have mutually exclusive access to resources, where in our approach resources have the shared semantics. The focus of our work is on *global resource usages* on multi-tenant environment such as Cloud systems. Another work based on the λ-calculus is in [90], where arguments of lambda functions are considered as *linear resource* quantitative properties of resource usages are showed there. On the contrary, we focus on qualitative properties of resource, based on access events that are issued by cloud services.

Future Work To ensure the correctness of resource usages, static analysis techniques are desirable to be developed for our approach. Our preliminary result shows that type and effect systems in style of [16] can be used to construct resource behaviour of cloud services in form of BPP processes, which offer many decidable results in model checking technique [78]. The main issue in developing type and effect systems is how to handle recursion and concurrency introduced when type checking service invocations. Another possible extension, concerning resource variables, is to consider access events of the form $\alpha(\xi)$, where ξ is a resource variable, similar to the work in [16]. It could be useful in cloud systems that cloud services can take resource names as their arguments. For example, the service $(\lambda x\ \alpha(x))$ can perform the action α over different resources, depending on which resource is bound to x. We address this issue in the concurrent setting (see Chapter 4 and 5).

Part II

Static Analysis for Distributed Resources

Chapter 4

The G-Local π-Calculus

In this chapter, we introduce an extension of π-calculus, called G-Local, with explicit primitives for the distributed ownerships of resources. The distinguished features of our approach are described below.

Resource-awareness. Modern programming paradigms for distributed systems radically transformed the way computational resources are integrated into applications. Resources are usually geographically distributed and have their own states, costs and access mechanisms. Moreover, resources are not created nor destroyed by applications, but directly acquired on-the-fly when needed from suitable resource rental services. Clearly, resource acquisition is subject to availability and requires the agreement between client requirements and service guarantees (Service Level Agreement – SLA). The dynamic acquisition of resources increases the complexity of software since the capability of adapting behaviour strictly depends on resource availability. *Ubiquitous computing* [2] and *Cloud computing* [37, 110, 6] provide illustrative examples of a new generation of applications where resource awareness is a major concern.

Since we build on top of the π-calculus, name-passing is the basic communication mechanism among processes. Beyond exchanging channel names, processes can pass resource names as well. Resource acquisition is instead based on a different abstraction. In order to acquire the ownership of a certain resource, a process issues a suitable request. Such request is routed in the network environment to the resource. The resource is granted only if it is available. In other words the process-resource interaction paradigm adheres to the *publish-subscribe* model: resources act as publishers while processes act as subscribers. Indeed, the publish-subscribe paradigm is not only a natural choice to represent distributed resources, but also emphasises the fact that resources have to be published by external parties and therefore have to be available to everyone through appropriate requests. Notice that processes issue their requests without being aware of the availability of the resources. When they have completed their task on the acquired resource, they release it and make it available for new requests. The two-stage nature of the publish-subscribe paradigm relaxes the inter-dependencies among computational components thus achieving a high degree of loose coupling among processes and resources. In this sense our model also resembles tuple-based systems [58]. Consequently, our model seems to be particularly suitable to manage distributed systems where the set of published resources is subject to frequent changes and dynamic reconfigurations.

Usage Policies. The design of suitable mechanisms to control the distributed acquisition and ownership of computational resources is a primary concern in our approach. Central to this is the abstract notion of resource. In our model, resources are *stateful* entities available in the network environment where processes live. In other words, all resource modifications are kept in the resource states, and therefore guaranteeing the persistence of the resource states. Specifically, a resource is described through the declaration of its interaction endpoint (the resource name), its *local* state and its *global* properties. Global properties establish and enforce the usage policies to be satisfied by any interaction that the resource engages with its client process. Global interaction properties can be expressed by means of *regular* linear time properties. The interplay between local and global information occurring in the process-resource interactions motivates the adjective *G-Local* given to our extension of the π-calculus. A distinguished feature of our approach is that the reconfiguration steps updating the structure of the available resources are not under the control of client processes. This means that the deployed resources can be dynamically *reconfigured* to deal with resource upgrade, resource un-availability, security intrusion and failures.

Reasoning techniques. To verify correct usages of resources, we sort to two static analysis techniques, namely *Control Flow Analysis* (CFA) and *Typing System*. In this chapter, we present CFA, while Typing System will be introduced in the next section. The results of these analysis are safe approximations of resource usages. Hence, they can be used to statically check whether or not the global properties of resources usages are respected by process interactions. In particular, in this way we can detect *bad usages* of resources, due to policy violations.

4.1 The G-Local π-Calculus

4.1.1 Syntax

We consider the monadic version of π-calculus [97] extended with suitable primitives to declare, access and dispose resources.

Remark 4.1.1. To handle resource management, here, we extend the π-calculus with specific constructs to acquire and release resources. These constructs are inspired by the event-notification paradigm (EN). In the EN approach, we have a collection of publishers and a collection of subscribers and the linkage between publishers and subscribers is loosely coupled. We argue that loosely coupling mechanisms are required to manage resources in the distributed setting. The emphasis on EN paradigm characterises our proposal with respect to other approaches based on the name-passing features of the π-calculus, e.g. passing private names of resources via scope extrusion. Notice that resources can be transmitted as well but the scoping of resources is dynamic and it is based on explicit acquisition.

We reuse part of the notation introduced in Chapter 3 for resources and their access actions. For the sake of clarity, we shall define them again in the concurrent setting.

Definition 4.1.2. Assume that $\mathcal{N}$ is a set of channel names (ranged over by a, b, x, y, z), $\mathcal{R}$ is a set of resource names (ranged over by r, s, t), $\mathcal{A}$ is a set of actions (ranged over by α, β) for accessing resources, and Φ is a set of policies (ranged over by φ, φ'). A special action $rel \notin \mathcal{A}$ is also assumed for releasing resources. We use w, w' to range over channel and resource names. We assume that these sets are pairwise disjoint. The set $\mathcal{P}_{gl}$ of processes is defined by the following grammar.

$$
\begin{array}{lcll}
P, P' & ::= & & \textit{processes} \\
 & & \mathbf{0} & \text{empty process} \\
 & | & \pi.P & \text{prefix action} \\
 & | & P + P' & \text{choice} \\
 & | & P \parallel P' & \text{parallel composition} \\
 & | & (\nu x)\ P & \text{restriction} \\
 & | & !P & \text{replication} \\
 & | & (r, \varphi, \eta)\{P\} & \text{resource joint point} \\
 & | & req(r)\{P\} & \text{resource request point} \\
\pi, \pi' & ::= & & \textit{action prefixes} \\
 & & a(w) & \text{free input} \\
 & | & \bar{a}w & \text{free output} \\
 & | & \tau & \text{internal action} \\
 & | & \alpha(r) & \text{event action} \\
 & | & rel(r) & \text{event action}
\end{array}
$$

The input prefix $a(w).P$ binds the name w (either a channel or a resource) within the process P, while the output prefix $\bar{a}w.P$ sends the name w along channel a and then continues as P. Note that resource names can be communicated, however they cannot be used as private names and used as channels. As usual, input prefixes and restrictions act as binders. The meaning of the remaining operators is standard. The notions of names $\mathsf{n}()$, free names $\mathsf{fn}()$, bound names $\mathsf{bn}()$ and substitution $\{-/-\}$ are defined as expected.

Our extension introduces resource-aware constructs in the π-calculus. The access prefix $\alpha(r)$ models the invocation of the action $\alpha \in \mathcal{A}$ over the resource bound to the variable r. Access labels specify the kind of access operation. An action α accessing a resource can be seen as an event observed by resource monitors and the operation represents a basic resource usage in the calculus. Traces, denoted by $\eta, \eta' \in \mathcal{A}^*$, are finite sequences of actions over A. This allows us to define a usage policy expressing properties of traces. The special action rel denotes releasing the ownership of the acquired resources.

In our programming model, resources are viewed as stateful entities, equipped with policies constraining their usages. More precisely, a resource is a triple (r, φ, η), where $r \in \mathcal{R}$ is a resource name, $\varphi \in \Phi$ is the associated usage policy and $\eta \in \mathcal{A}^*$ is a state (ϵ denotes the empty state). Policies specify the required properties on resource usages. Policies are defined by means of a resource-aware logic (see [16, 17, 39]). For instance, in [16], the policies are expressed in terms of automata over an infinite alphabet, where automata steps correspond to actions on resources and final states indicate policy violations.

To cope with resource-awareness, we introduce two primitives managing resource boundaries: resource joint point $(r, \varphi, \eta)\{P\}$ and resource request point $req(r)\{P\}$. Instead of considering resources as an extension of private names with a set of traces of access events like in [66], the construct (r, φ, η) is introduced to represent a notion of resources with explicit boundaries. Unlike the dynamic scope of private names, where processes can concurrently access "private names" (i.e. resources), resource boundaries in our approach guarantee exclusive access to processes that are located inside their corresponding resource boundaries. More precisely, a process $(r, \varphi, \eta)\{P\}$ is a process that behaves like P, in the resource boundary (r, φ, η), where r can be accessed according to the policy φ. The state η is updated at each action $\alpha(r)$, issued by P, according to the required policy φ. Processes of the form $(r, \varphi, \eta)\{\mathbf{0}\}$ represent *available resources*. These processes are idle: they cannot perform any operation. In other words, resources can only react to requests. A resource request point $req(r)\{P\}$ represents a process asking for the resource r. Only if the request is fulfilled, i.e. the required resource is available, the process can enter the required resource boundary and can use the resource r, provided that the policy is satisfied.

Example 4.1.3. We consider a small example consisting of a green cloud computing environment, where the energy cost is taken into consideration in order to save energy. Each access $\alpha(r)$ to a resource r has an integer cost c_α and the policy φ of each resource r is a threshold value v_ϕ that cannot be passed. Suppose to have a couple of resources r_1 and r_2, with the policies φ_1 and φ_2. Users receive resources on channels x_i (for $i = 1,\ 2,\ 3$) and y. Policies are respected whether the sum of the costs of accesses do not pass the value fixed for each of the policy.

The initial configuration is given below. Resources (r_1 and r_2) have empty traces and have v_{ϕ_1} and v_{ϕ_2} as threshold values. The access action α comes associated with the cost c_α, and the action β with the cost c_β. Suppose for instance that c_α is quite expensive and that the threshold of r_2 is not very high, while that of r_1 is. Suppose, in particular, that $3 \cdot c_\alpha < v_{\phi_1}$ and that $c_\alpha + c_\beta > v_{\phi_2}$.

$$
\begin{aligned}
\mathit{Res} &::= (r_1, \varphi_1, \epsilon)\{\mathbf{0}\} \parallel (r_1, \varphi_1, \epsilon)\{\mathbf{0}\} \parallel (r_2, \varphi_2, \epsilon)\{\mathbf{0}\} \\
\mathit{Users} &::= x_1(s_1).\mathit{req}(s_1)\{\alpha(s_1).\mathit{rel}(s_1)\} \parallel x_2(s_2).\mathit{req}(s_2)\{\alpha(s_2).\mathit{rel}(s_2)\} \parallel \\
&\qquad x_3(s_3).\mathit{req}(s_3)\{\alpha(s_3).\mathit{rel}(s_3)\} \parallel y(t).\mathit{req}(t)\{\beta(t).\mathit{rel}(t)\} \\
\mathit{Plan} &::= \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0} \\
\mathit{System} &::= \mathit{Res} \parallel \mathit{Users} \parallel \mathit{Plan}
\end{aligned}
$$

4.1.2 Operational semantics

The operational semantics of the G-Local π-calculus is defined by the transition relation given in Fig. 4.2. Labels μ, μ' for transitions are τ for silent actions, $x(w)$ for free input, $\bar{x}v$ for free output, $\bar{x}(v)$ for bound output, $\alpha(r)$, $\alpha?r$ and $\overline{\alpha(r)}$ ($\mathit{rel}(r)$, $\mathit{rel}?r$ and $\overline{\mathit{rel}(r)}$, resp.) for closed, open, and faulty access or release actions over resource r. As in the standard π-calculus, the effect of bound output is to extrude the sent name from the initial scope to the external environment.

To simplify the definition of our Control Flow Analysis, we impose a discipline in the choice of fresh names, and therefore to α-equivalent. Indeed, the result of analysing a process P, must still hold for all its derivative processes Q, including all the processes obtained from Q by α-equivalent. In particular, the CFA uses the names and the variables occurring in P. If they were changed by the dynamic evolution, the analysis values would become a sort of dangling references, no more connected with the actual values. To statically maintain the identity of values and variables, we partition all the names used by a process into finitely many equivalence classes. We denote with $\lfloor n \rfloor$ the equivalence class of the name n, that is called *canonical name* of n. Not to further overload our notation, we simply write n for $\lfloor n \rfloor$, when unambiguous. We further demand that two names can be alpha-renamed only when they have the same canonical name.

In addition, we introduce the specific laws for managing the resource-aware constructs, reported in Fig. 4.1. If two processes P_1 and P_2 are equivalent, then also P_1 and P_2 when plugged inside the same resource boundaries are. Resource request and resource joint points can be swapped with the restriction boundary since restriction is not applied to resource names but only to channel names. The last law is crucial for managing the discharge of resources. This law allows rearrangements of available resources, e.g. an available resource is allowed to enter or escape within a resource boundary.

The rules [Act], [Cong], [Par], [Choice], [Res], [Open], [Comm] and [Close] are the standard π-calculus ones (see Fig. 2.8 in Chapter 2). We now comment on the semantic rules corresponding to the treatment of resources. The rule [Act$_{\mathrm{R}}$] models a process that tries to perform an action α (rel, resp.) on the resource r. This attempt

$$
\begin{array}{ll}
P \equiv Q \text{ if } P \text{ and } Q \text{ are } \alpha\text{-equivalent} & \text{(as explained below)} \\
(P + Q) + R \equiv P + (Q + R) & (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \\
P + Q \equiv Q + P & P \parallel Q \equiv Q \parallel P \\
P + \mathbf{0} \equiv P & P \parallel \mathbf{0} \equiv P \\
(\nu x)\mathbf{0} \equiv \mathbf{0} & (\nu x)P \parallel Q \equiv (\nu x)(P \parallel Q) \ x \notin \mathsf{fn}(Q), \\
!P \equiv P \parallel !P &
\end{array}
$$

$$
\begin{array}{l}
(r, \varphi, \eta)\{P_1\} \equiv (r, \varphi, \eta)\{P_2\} \text{ if } P_1 \equiv P_2 \\
req(r)\{P_1\} \equiv req(r)\{P_2\} \text{ if } P_1 \equiv P_2 \\
(\nu x)(r, \varphi, \eta)\{P\} \equiv (r, \varphi, \eta)\{(\nu x)P\} \\
(\nu x)req(r)\{P\} \equiv req(r)\{(\nu x)P\} \\
(r_2, \varphi_2, \eta_2)\{\mathbf{0}\} \parallel (r_1, \varphi_1, \eta_1)\{P\} \equiv (r_1, \varphi_1, \eta_1)\{(r_2, \varphi_2, \eta_2)\{\mathbf{0}\} \parallel P\}
\end{array}
$$

Figure 4.1: Structural congruence.

is seen as an *open action*, denoted by the label $\alpha?r$ ($rel?r$, resp.). We introduce the rule [Comm$_R$] to model the communication of resource names between processes.

When a resource r is available, then it can be acquired by a process P that enters the corresponding resource boundary (r, φ, η), as stated by the rule [Acquire]. Symmetrically, according to the rule [Release], the process P can release an acquired resource r and update the state of its resources by appending rel to η. In the resulting process, the process P escapes the resource boundary. Furthermore, the resource becomes available, i.e. it encloses the empty process $\mathbf{0}$. Intuitively, an access is successful only if the process is inside the boundary of r, and the action α satisfies the policy for r (see [Policy$_1$] and [Policy$_2$]). Similarly, releasing succeeds only if the process is inside the boundary of r (see [Release]).

The rules [Policy$_1$], [Policy$_2$] check whether the execution of the action α on the resource r obeys the policy φ, i.e. whether the updated state $\eta.\alpha$, obtained by appending α to the current state η, is consistent w.r.t. φ. If the policy is obeyed, then the updated state $\eta.\alpha$ is stored in the resource state according to the rule [Policy$_1$] and the action becomes *closed* and if not, then the resource is forcedly released according to the rule [Policy$_2$] and a *faulty* action $\overline{\alpha(r)}$ is fired. In the continuation P' of the process, all the actions over the the resource r remain open and without effect on the resource.

The rules [Local$_1$] and [Local$_2$] express that actions can bypass resource boundaries for r only if they do not involve the resource r.

Remark 4.1.4. The rule [Acquire] is not inductively given in the SOS style. We

$$[\text{Act}] \quad \pi.P \xrightarrow{\pi} P \quad \pi \neq \alpha(r), rel(r)$$

$$[\text{Cong}] \quad \frac{P_1 \equiv P_1' \quad P_1' \xrightarrow{\mu} P_2' \quad P_2' \equiv P_2}{P_1 \xrightarrow{\mu} P_2}$$

$$[\text{Par}] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \parallel P_2 \xrightarrow{\mu} P_1' \parallel P_2} \ \mathsf{bn}(\mu) \cap \mathsf{fn}(P_2) = \emptyset$$

$$[\text{Choice}] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 + P_2 \xrightarrow{\mu} P_1'}$$

$$[\text{Res}] \quad \frac{P \xrightarrow{\mu} P'}{(\nu x)P \xrightarrow{\mu} (\nu x)P'} \ x \notin \mathsf{n}(\mu)$$

$$[\text{Open}] \quad \frac{P \xrightarrow{\bar{a}x} P'}{(\nu x)P \xrightarrow{\bar{a}(x)} P'} \ x \neq a$$

$$[\text{Comm}] \quad \frac{P_1 \xrightarrow{\bar{a}y} P_1' \quad P_2 \xrightarrow{a(z)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{y/z\}}$$

$$[\text{Close}] \quad \frac{P_1 \xrightarrow{a(x)} P_1' \quad P_2 \xrightarrow{\bar{a}(y)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\tau} (\nu y)(P_1' \parallel P_2'\{y/x\})}$$

$$[\text{Act}_\text{R}] \quad \begin{array}{l} \alpha(r).P \xrightarrow{\alpha?r} P \\ rel(r).P \xrightarrow{rel?r} P \end{array}$$

$$[\text{Comm}_\text{R}] \quad \frac{P_1 \xrightarrow{\bar{x}r} P_1' \quad P_2 \xrightarrow{x(s)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\tau} P_1' \parallel P_2'\{r/s\}}$$

$$[\text{Acquire}] \quad req(r)\{P\} \parallel (r, \varphi, \eta)\{\mathbf{0}\} \xrightarrow{\tau} (r, \varphi, \eta)\{P\}$$

$$[\text{Release}] \quad \frac{P \xrightarrow{rel?r} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{rel(r)} (r, \varphi, \eta.rel)\{\mathbf{0}\} \parallel P'}$$

$$[\text{Policy}_1] \quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \models \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\alpha(r)} (r, \varphi, \eta.\alpha)\{P'\}}$$

$$[\text{Policy}_2] \quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \not\models \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\overline{\alpha(r)}} (r, \varphi, \eta)\{\mathbf{0}\} \parallel P'}$$

$$[\text{Local}_1] \quad \frac{P \xrightarrow{\mu} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{\mu} (r, \varphi, \eta)\{P'\}} \ r \notin \mathsf{n}(\mu)$$

$$[\text{Local}_2] \quad \frac{P \xrightarrow{\mu} P'}{req(r)\{P\} \xrightarrow{\mu} req(r)\{P'\}} \ r \notin \mathsf{n}(\mu)$$

Figure 4.2: Operational Semantics of G-Local π processes.

could rephrase it in the SOS style by the following rules:

$$\begin{array}{ll}
[\text{Resreq}] & req(r)\{P\} \xrightarrow{(r,\varphi,\eta)} (r,\varphi,\eta)\{P\} \\
[\text{Resjoin}] & (r,\varphi,\eta)\{\mathbf{0}\} \xrightarrow{\overline{(r,\varphi,\eta)}} \mathbf{0} \\
[\text{Comm}'_{\text{R}}] & \dfrac{P \xrightarrow{(r,\varphi,\eta)} P' \quad Q \xrightarrow{\overline{(r,\varphi,\eta)}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}
\end{array}$$

In alternative, we could define the rule [Acquire] as an axiom:

$$req(r)\{P\} \parallel (r,\varphi,\eta)\{\mathbf{0}\} \equiv (r,\varphi,\eta)\{P\}$$

We choose to orient the congruence rule in order to emphasise the acquisition of resources.

Remark 4.1.5. Note that resource entities could be dynamically reconfigured via resource movements. Besides the structural rules, we here could include the following transition rules [Appear] and [Disappear]:

$$\begin{array}{ll}
[\text{Appear}] & P \xrightarrow{\tau} P \parallel (r,\varphi,\eta)\{\mathbf{0}\} \\
[\text{Disappear}] & (r,\varphi,\eta)\{P\} \xrightarrow{\tau} \mathbf{0}
\end{array}$$

These rules describe the abstract behaviour of the resource manager performing asynchronous resource reconfigurations. In other words, in this proposal, resource configuration is not under the control of processes. Resources are created and destroyed by external entities and processes can only observe their presence/absence. Dynamic reconfiguration would offer a high degree of loose coupling among processes and resources and would be a feature not present in other proposals, such as the one in [66] as well as the ones in [70, 104]. This will be explored in future work.

Example 4.1.6. To explain the operational semantics, we come back to our running example. The following possible dynamic computation illustrates how the system works and a possible policy violation. At the beginning, $Users$ instantiates a new user (a resource request point) when receiving a resource name, e.g. r_1:

$$\begin{array}{l}
System \\
\equiv Res \parallel Users' \parallel x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\} \parallel \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.Plans' \\
\xrightarrow{\tau} Res \parallel Users' \parallel Plan' \parallel req(r_1)\{\alpha(r_1).rel(r_1)\},
\end{array}$$

where $Users' ::= x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\} \parallel x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\} \parallel y(t).req(t)\{\beta(t).rel(t)\}$ and $Plan' ::= \bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0}$.
At this point, the new user can acquire the resource and other resource names are also available (on the channel $x_2,\ x_3,\ y$). In the following, for the sake of simplicity, we only show the sub-processes that involve computation. Assume that the new user takes r_1, then we have the following transitions:

$$\begin{aligned}
&req(r_1)\{\alpha(r_1).rel(r_1)\} \parallel (r_1, \varphi_1, \epsilon)\{\mathbf{0}\}\\
&\xrightarrow{\tau} (r_1, \varphi_1, \epsilon)\{\alpha(r_1).rel(r_1)\}\\
&\xrightarrow{\alpha(r_1)} (r_1, \varphi_1, \alpha)\{rel(r_1)\}\\
&\xrightarrow{rel(r_1)} (r_1, \varphi_1, \alpha.rel)\{\mathbf{0}\}
\end{aligned}$$

Now, the remaining three further users are similarly instantiated.

$$\begin{aligned}
&Users'|Plans'\\
&\xrightarrow{\tau} x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\} \parallel x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\}\\
&\quad \parallel req(r_2)\{\beta(r_2).rel(r_2)\} \parallel \bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle\\
&\xrightarrow{\tau} x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\} \parallel req(r_2)\{\beta(r_2).rel(r_2)\} \parallel req(r_2)\{\alpha(r_2).rel(r_2)\} \parallel \bar{x}\langle r_1\rangle\\
&\xrightarrow{\tau} req(r_2)\{\beta(r_2).rel(r_2)\} \parallel req(r_2)\{\alpha(r_2).rel(r_2)\} \parallel req(r_1)\{\alpha(r_1).rel(r_1)\}
\end{aligned}$$

In the current setting, the new three users make one request on the remaining resource r_1 and two requests on r_2. Since we have only one copy of r_2, requests should be done one at a time. Suppose to first satisfy the requests of r_1, as in the following transitions:

$$\begin{aligned}
&(r_1, \varphi_1, \alpha.rel)\{\alpha(r_1).rel(r_1)\}\\
&\xrightarrow{\alpha(r_1)} (r_1, \varphi_1, \alpha.rel.\alpha)\{rel(r_1)\}\\
&\xrightarrow{rel(r_1)} (r_1, \varphi_1, \alpha.rel.\alpha.rel)\{\mathbf{0}\}
\end{aligned}$$

Note that now the first resource is available again. Similarly, the second request proceeds as follows:

$$\begin{aligned}
&req(r_2)\{\beta(r_2).rel(r_2)\} \parallel (r_2, \varphi_2, \epsilon)\{\mathbf{0}\}\\
&\xrightarrow{\tau} (r_2, \varphi_2, \epsilon)\{\beta(r_2).rel(r_2)\}\\
&\xrightarrow{\beta(r_2)} (r_2, \varphi_2, \beta)\{rel(r_2)\}\\
&\xrightarrow{rel(r_2)} (r_2, \varphi_2, \beta.rel)\{\mathbf{0}\}
\end{aligned}$$

If the third request proceeded, then a forced release could occur. This happens because the user attempts to perform an α action on r_2, on which a β action was previously performed: since $c_\alpha + c_\beta > v_{\varphi_2}$, this amounts to a violation of the policy φ_2.

$$\begin{aligned}
&req(r_2)\{\alpha(r_2).rel(r_2)\} \parallel (r_2, \varphi_2, \beta.rel)\{\mathbf{0}\}\\
&\xrightarrow{\tau} (r_2, \varphi_2, \beta.rel)\{\alpha(r_2).rel(r_2)\}\\
&\xrightarrow{\overline{\alpha(r_2)}} (r_2, \varphi_2, \beta.rel)\{\mathbf{0}\} \parallel \; rel(r_2)
\end{aligned}$$

4.2 Control Flow Analysis

In this section, we present a CFA for our calculus, extending the one for π-calculus [26]. The CFA computes a safe over-approximation of all the possible communications of resource and channel names on channels. Furthermore, it provides an over-approximation of all the possible usage traces on the given resources. The analysis is performed under the perspective of processes. This amounts to saying that the analysis tries to answer the following question: "Are the resources initially granted sufficient to guarantee a correct usage?". We assume that a certain fixed amounts of resources is given and we do not consider any dynamic reconfiguration. Furthermore, we assume that all resource names are different.

For the sake of simplicity, we provide the analysis for a subset of our calculus, in which processes have *finite* behaviour, i.e. without replication, and all resource instances have different names. and processes enclosed in the scopes of resources are *sequential processes* (ranged over by Q, Q'), as described by the following syntax. Intuitively, a sequential process represents a single thread of execution in which one or more resources can be used.

$$\begin{array}{rcl@{\qquad}rcl}
P, P' & ::= & \text{processes} & Q, Q' & ::= & \textit{sequential processes} \\
& | & \mathbf{0} & & & \mathbf{0} \\
& | & \pi.P & & | & \pi.Q \\
& | & (\nu x)\ P & & | & (\nu x)\ Q \\
& | & P + P' & & | & Q + Q' \\
& | & P \parallel P' & & | & (r, \varphi, \eta)\{\mathbf{0}\} || Q \\
& | & req(s)\{Q\} & & | & req(s)\{Q\} \\
& | & (r, \varphi, \eta)\{Q\} & & | & (r, \varphi, \eta)\{Q\}
\end{array}$$

This implies that one single point for releasing each resource occurs in each non deterministic branch of a process. We assume that, by construction, in every sequential branch located in the scope of the resource r, there is always a release action $rel(r)$ coming after the last access to the resource. It is not difficult to provide a function that checks whether this condition is satisfied. The assumed constraint amounts to guaranteeing that resources are released after their use. Note that the only parallel branching configuration $(r, \varphi, \eta)\{\mathbf{0}\} \parallel Q$ is needed in order to handle release actions. The extension to general parallel processes is possible. Nevertheless, it requires some more complex technical machinery in order to check whether all the parallel branches synchronise among them, before releasing the shared resource.

In order to facilitate our analysis, we further associate labels $\chi \in \mathcal{L}$ with resource boundaries as follows: $(r, \varphi, \eta)\{Q\}^{\chi}$ and $req(r)\{Q\}^{\chi}$, in order to give a name to the sub-processes in the resource scopes. Note that this annotation can be performed in a pre-processing step and does not affect the semantics of the calculus. During the computation, resources are released and acquired by other processes. Statically, sequences of labels $S \in \mathcal{L}^*$ are used to record the sequences of sub-processes possibly entering the scope of a resource. Furthermore, to make our analysis more informative, we enrich the execution traces η with special actions that record the fact that a resource has been possibly:

- acquired by the process labelled χ: $in(\chi)$, with a successful request;
- released by the process labelled χ: $out(\chi)$ with a successful release;
- taken away from the process labelled χ: $err\_out(\chi)$, with a forced release because of an access action on r that does not satisfy the policy.

The new set of traces is $\widehat{\mathcal{A}}^*$, where $\widehat{\mathcal{A}} = \mathcal{A} \cup \{in(\chi), out(\chi), err\_out(\chi) \mid \chi \in \mathcal{L}\}$. The corresponding dynamic traces can be obtained by simply removing all the special actions.

The result of analysing a process P is a tuple (ρ, κ, Γ) called *estimate* of P, that provides an approximation of resource behaviour. More precisely, ρ and κ offer an over-approximation of all the possible values that the variables in the system may be bound to, and of the values that may flow on channels. The component Γ provides a set of traces of actions, including *bad* ones, on each resource. Using this information, we can statically check resource usages against the required policies.

To validate the correctness of a given estimate (ρ, κ, Γ), we state a set of clauses that operate upon judgements of the form $(\rho, \kappa, \Gamma) \models^\delta P$, where δ is a sequence of pairs $[(r, \varphi, \eta), S]$, recording the resource scope nesting. This sequence has initially empty components, denoted by $[\epsilon, \epsilon]$. In particular, the analysis keeps track of the following information:

- An approximation $\rho : \mathcal{N} \cup \mathcal{R} \to \wp(\mathcal{N} \cup \mathcal{R})$ of names bindings. If $a \in \rho(x)$ then the channel variable x can assume the channel value a. Similarly, if $r \in \rho(s)$ then the resource variable s can assume the resource value r.
- An approximation $\kappa : \mathcal{N} \to \wp(\mathcal{N} \cup \mathcal{R})$ of the values that can be sent on each channel. If $b \in \kappa(a)$, then the channel value b can be output on the channel a, while $r \in \kappa(a)$, then the resource value r can be output on the channel a.
- An approximation $\Gamma : \mathcal{R} \to \wp(\{[(\varphi, \eta), S] \mid \varphi \in \Phi, S \in \mathcal{L}^*, \eta \in \widehat{\mathcal{A}}^*\})$ of resource behaviour. If $[(\varphi, \eta), S] \in \Gamma(r)$ then η is one of the possible traces over r (with policy φ), that is performed by a sequence of sub-processes, whose labels χ are juxtaposed in S.

The judgements of the CFA, given in Tab. 4.1, are based on structural induction of processes. We use the following shorthands to simplify the treatment of the sequences δ. The predicate $[(r, \varphi, \eta), \chi] \Subset \delta$ is used to check whether the pair $[(r, \varphi, \eta), \chi]$ occurs in δ, i.e. whether $\delta = \delta'[r, (\varphi, \eta), \chi]\delta''$. Furthermore, we use $\delta\{[(r, \varphi, \eta.\alpha), S]/[(r, \varphi, \eta), S]\}$ to indicate that the pair $[(r, \varphi, \eta), S]$ is replaced by $[(r, \varphi, \eta.\alpha), S]$ in the sequence δ. With $\delta \setminus [(r, \varphi, \eta), S]$ we indicate the sequence where the occurrence $[(r, \varphi, \eta), S]$ has been removed, i.e. the sequence $\delta'\delta''$, if $\delta = \delta'[(r, \phi, \eta), S]\delta''$. The uniqueness of resources in the sequence δ is ensured by the fact that we assume a certain fixed amount of resources and no duplicate of resource names.

All the clauses dealing with a compound process check that the analysis also holds for its immediate sub-processes. In particular, the analysis of $(\nu x)P$ is equal to the one of P. We comment on the main rules. Besides the validation of the continuation process P, the rule for output, requires that the set of names that can be communicated along each element of $\rho(x)$ includes the names to which y can evaluate. Symmetrically, the rules for input demands that the set of names that can pass along x is included in the set of names to which y can evaluate. Intuitively, the estimate components take into account the possible dynamics of the process under consideration. The clauses' checks mimic the semantic evolution, by modelling the semantic preconditions and the consequences of the possible synchronisations. In the rule for input, e.g., CFA checks whether the precondition of a synchronisation is satisfied, i.e. whether there is a corresponding output possibly sending a value that can be received by the analysed input. The conclusion imposes the additional requirements on the estimate components, necessary to give a valid prediction of the analysed synchronisation action, mainly that the variable y can be bound to that value.

To gain greater precision in the prediction of resource usages, in the second rule, the continuation process is analysed, for all possible bindings of the resource variable s. This explains why we have all the other rules for resources, without resource variables.

The rule for *resource joint point* updates δ to record that the immediate sub-process is inside the scope of the new resource and there it is analysed. If the process is empty, i.e. in the case the resource is available, the trace of actions is recorded in $\Gamma(r)$.

In the rule for *resource request point*, the analysis for Q is performed for every possible element $[(\varphi, \eta), S]$ from the component $\Gamma(r)$. This amounts to saying that the resource r can be used starting from any possible previous trace η. Furthermore, η is enriched by the special action $in(\chi)$ that records the fact that the resource r can be possibly acquired by the process labelled χ. In order not to append the same trace more than once, we have the condition that S does not contain $in(\chi)$. This prevents the process labelled χ to do it.

According to the rule for *access action*, if the pair $[(r, \varphi, \eta), S\chi]$ occurs in δ (i.e. if we are inside the resource scope of r) and the updated history $\eta.\alpha$ obeys the policy φ, then the analysis result also holds for the immediate subprocess and δ is updated in δ', by replacing $[(r, \varphi, \eta), S\chi]$ in δ with $[(r, \varphi, \eta.\alpha), S\chi]$, therefore recording the resource accesses to r possibly made by the sub-process labelled by χ.

In case the action possibly violates the policy associated with r (see the last conjunct), the process labelled χ may loose the resource r, as recorded by the trace in Γ, $[(\varphi, \eta.err\_out(\chi)), S\chi]$, with the special action $err\_out(\chi)$ appended to η. If instead, the action on r is not viable because the process is not in the scope of r,

$(\rho,\kappa,\Gamma) \models^{\delta} \mathbf{0}$	iff true
$(\rho,\kappa,\Gamma) \models^{\delta} \tau.P$	iff $(\rho,\kappa,\Gamma) \models^{\delta} P$
$(\rho,\kappa,\Gamma) \models^{\delta} \bar{x}w.P$	iff $\forall a \in \rho(x) : \rho(w) \subseteq \kappa(a) \;\wedge\; (\rho,\kappa,\Gamma) \models^{\delta} P$
$(\rho,\kappa,\Gamma) \models^{\delta} x(y).P$	iff $\forall a \in \rho(x) : \kappa(a) \cap \mathcal{N} \subseteq \rho(y) \;\wedge\; (\rho,\kappa,\Gamma) \models^{\delta} P$
$(\rho,\kappa,\Gamma) \models^{\delta} x(s).P$	iff $\forall a \in \rho(x) : \kappa(a) \cap \mathcal{R} \subseteq \rho(s)$ $\wedge\ \forall r \in \rho(s) : (\rho,\kappa,\Gamma) \models^{\delta} P\{r/s\}$
$(\rho,\kappa,\Gamma) \models^{\delta} P_1 + P_2$	iff $(\rho,\kappa,\Gamma) \models^{\delta} P_1 \wedge (\rho,\kappa,\Gamma) \models^{\delta} P_2$
$(\rho,\kappa,\Gamma) \models^{\delta} P_1 \parallel P_2$	iff $(\rho,\kappa,\Gamma) \models^{\delta} P_1 \wedge (\rho,\kappa,\Gamma) \models^{\delta} P_2$
$(\rho,\kappa,\Gamma) \models^{\delta} (\nu x)P$	iff $(\rho,\kappa,\Gamma) \models^{\delta} P \wedge x \in \rho(x)$
$(\rho,\kappa,\Gamma) \models^{\delta} (r,\varphi,\eta)\{Q\}^{S}$	iff $(\rho,\kappa,\Gamma) \models^{\delta.[(r,\varphi,\eta),S]} Q$
$(\rho,\kappa,\Gamma) \models^{\delta} (r,\varphi,\eta)\{\mathbf{0}\}^{S}$	iff $(\rho,\kappa,\Gamma) \models^{\delta.[(r,\varphi,\eta),S]} \mathbf{0} \wedge [(\varphi,\eta),S] \in \Gamma(r)$
$(\rho,\kappa,\Gamma) \models^{\delta} req(r)\{Q\}^{\chi}$	iff $\forall[(\varphi,\eta),S] \in \Gamma(r) \wedge in(\chi) \notin S$ $\Rightarrow (\rho,\kappa,\Gamma) \models^{\delta.[(r,\varphi,\eta.in(\chi)),S\chi]} Q$
$(\rho,\kappa,\Gamma) \models^{\delta} \alpha(r).Q$	iff $[(r,\varphi,\eta),S\chi] \mathrel{\mathsf{E}} \delta \wedge \eta.\alpha \models \varphi \Rightarrow (\rho,\kappa,\Gamma) \models^{\delta'} Q$ $\wedge\ [(r,\varphi,\eta),S\chi] \mathrel{\mathsf{E}} \delta \wedge \eta.\alpha \not\models \varphi \Rightarrow \left\{ \begin{array}{l} (\rho,\kappa,\Gamma) \models^{\delta''} Q \;\wedge \\ [(\varphi,\eta.err\_out(\chi)),S\chi] \in \Gamma(r) \end{array} \right.$ $\wedge\ [(r,\varphi,\eta),S\chi] \not\mathrel{\mathsf{E}} \delta \Rightarrow (\rho,\kappa,\Gamma) \models^{\delta} Q$ with $\delta' = \delta\{[(r,\varphi,\eta.\alpha),S\chi]/[(r,\varphi,\eta),S\chi]\}$ and $\delta'' = \delta \setminus [(r,\varphi,\eta),S\chi]$
$(\rho,\kappa,\Gamma) \models^{\delta} rel(r).Q$	iff $[(r,\varphi,\eta),S\chi] \mathrel{\mathsf{E}} \delta \Rightarrow \left\{ \begin{array}{l} (\rho,\kappa,\Gamma) \models^{\delta'} Q \;\wedge \\ [(\varphi,\eta.rel.out(\chi)),S\chi] \in \Gamma(r) \end{array} \right.$ $\wedge\ [(r,\varphi,\eta),S\chi] \not\mathrel{\mathsf{E}} \delta \Rightarrow (\rho,\kappa,\Gamma) \models^{\delta} Q$ with $\delta' = \delta \setminus [(r,\varphi,\eta),S\chi]$
$(\rho,\kappa,\Gamma) \models^{\delta} (r,\varphi,\eta)\{\mathbf{0}\}^{S} \parallel Q$	iff $(\rho,\kappa,\Gamma) \models^{\delta} (r,\varphi,\eta)\{\mathbf{0}\}^{S} \wedge (\rho,\kappa,\Gamma) \models^{\delta} Q$

Table 4.1: CFA Rules.

then the analysis holds for the immediate subprocess, i.e. the action is skipped.

According to the rule for *release*, the trace of actions $\eta' = \eta.rel.out(\chi)$ over r at χ is recorded in $\Gamma(r)$. Other sub-processes can access the resource starting from the trace η'. Furthermore, $[(r, \varphi, \eta), S]$ is removed from δ and this reflects the fact that the process Q can regularly exit its scope, once released the resource r. If instead, the release action on r is not viable because the process is not in the scope of r, then the analysis holds for the immediate subprocess, i.e. the action is skipped.

4.2.1 Correctness

The analysis provides us with an approximation of the overall behaviour of the analysed process. Moreover, it is proved to be correct: the analysis indeed respects the operational semantics of G-Local π-calculus, as shown by the subject reduction theorem. More precisely, if an estimate is valid for a process, then it is also valid for all derivatives of that process. Before stating and proving it, we need some auxiliary lemmas.

In the following lemma, we prove that if (ρ, κ, Γ) is an estimate of P, a value v is possibly bound to x during the evolution of P, i.e. $v \in \rho(x)$, then (ρ, κ, Γ) is also an estimate of $P\{v/x\}$.

Lemma 4.2.1 (Substitution)**.** *If $(\rho, \kappa, \Gamma) \models^{\delta} P$ and $v \in \rho(x)$, where $x \in \mathsf{n}(P)$, then $(\rho, \kappa, \Gamma) \models^{\delta} P\{v/x\}$.*

Proof. First, we prove the following fact: $\forall y : \rho(y(\{v/x\})) \subseteq \rho(y)$. If $y \neq x$ then $\rho(y(\{v/x\})) = \rho(y)$, while if $y = x$, then $\rho(y(\{v/x\})) = \rho(v)$. Since $v = \rho(v)$ and $v \in \rho(x)$, we have $\rho(v) \subseteq \rho(x)$. The proof of thesis proceeds by structural induction on P. We consider here only the most interesting cases.

The case of $P = z(w).P'$: We may assume that $w \neq v, x$. $(\rho, \kappa, \Gamma) \models^{\delta} P$ amounts to checking that

$$\begin{array}{ll} (\rho, \kappa, \Gamma) \models^{\delta} P' \wedge s & \\ \forall a \in \rho(z) : \kappa(a) \cap \mathcal{N} \subseteq \rho(w) & (1) \end{array}$$

By induction hypothesis and the fact stated above, we have that $(\rho, \kappa, \Gamma) \models^{\delta} P'\{v/x\}$. Furthermore, since $\rho(z(\{v/x\})) \subseteq \rho(z)$, (1) implies that $\forall a \in \rho(z(\{v/x\})) : \kappa(a) \cap \mathcal{N} \subseteq \rho(w)$. This is equivalent to $(\rho, \kappa, \Gamma) \models^{\delta} P\{v/x\}$.

The case of $P = \alpha(r).Q$: $(\rho, \kappa, \Gamma) \models^{\delta} P$ amounts to checking that

$$\begin{array}{ll} ([(r, \varphi, \eta), S\chi] \in \delta \wedge \eta.\alpha \models \varphi & \Rightarrow (\rho, \kappa, \Gamma) \models^{\delta'} Q) \\ ([(r, \varphi, \eta), S\chi] \in \delta \wedge \eta.\alpha \not\models \varphi & \Rightarrow \left\{ \begin{array}{l} (\rho, \kappa, \Gamma) \models^{\delta''} Q \wedge \\ {[(\varphi, \eta.err\_out(\chi)), S\chi] \in \Gamma(r)} \end{array} \right. \\ ([(r, \varphi, \eta), S\chi] \notin \delta & \Rightarrow (\rho, \kappa, \Gamma) \models^{\delta} Q, \end{array}$$

where $\delta' = \delta\{[(r, \varphi, \eta.\alpha), S\chi]/[(r, \varphi, \eta), S\chi]$ and $\delta'' = \delta \setminus [(r, \varphi, \eta), S\chi]$. By the induction hypothesis, for any δ, $(\rho, \kappa, \Gamma) \models^{\delta} Q$ implies that $(\rho, \kappa, \Gamma) \models^{\delta} Q\{v/x\}$. Therefore, we have $(\rho, \kappa, \Gamma) \models^{\delta} P\{v/x\}$ as required. □

The following lemma states that congruent processes have the same valid estimates.

Lemma 4.2.2 (Congruence). *If $(\rho, \kappa, \Gamma) \models^\delta P$ and $P \equiv Q$, then $(\rho, \kappa, \Gamma) \models^\delta Q$.*

Proof. The proof amounts to a straightforward inspection of each of the clauses defining the structural congruence axioms. Here we consider the most interesting case.
Case of P is α-conversion of Q: since we exploit canonical names to maintain the identity of bound names, changes of bound names do not affect on results of CFA analysis. We have that $\rho(a) = \rho(a') = \rho(\lfloor a \rfloor)$ and $\kappa(a) = \kappa(a') = \kappa(\lfloor a \rfloor)$, where a and a' are names in the equivalent class $\lfloor a \rfloor$.
Case of $(r_1, \varphi_1, \eta_1)\{(r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \parallel P\}^{S_1} \equiv (r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \parallel (r_1, \varphi_1, \eta_1)\{P\}^{S_1}$. Note that S_1 and S_2 are sequences of labels indicating sub-processes that used resources r_1 and r_2 respectively. We have that

$$
\begin{gathered}
(\rho, \kappa, \Gamma) \models^\delta (r_1, \varphi_1, \eta_1)\{(r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \parallel P\}^{S_1} \\
\text{iff} \\
(\rho, \kappa, \Gamma) \models^{\delta.[(r_1,\varphi_1,\eta_1),S_1]} (r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \parallel P \\
\text{iff} \\
(\rho, \kappa, \Gamma) \models^{\delta.[(r_1,\varphi_1,\eta_1),S_1]} (r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \wedge\ (\rho, \kappa, \Gamma) \models^{\delta.[(r_1,\varphi_1,\eta_1),S_1]} P \\
\text{iff} \\
(\rho, \kappa, \Gamma) \models^{\delta.[(r_1,\varphi_1,\eta_1),S_1].[(r_2,\varphi_2,\eta_2),S_2]} \mathbf{0} \wedge\ (\rho, \kappa, \Gamma) \models^{\delta.[(r_1,\varphi_1,\eta_1),S_1]} P \\
\text{iff} \\
(\rho, \kappa, \Gamma) \models^{\delta.[(r_2,\varphi_2,\eta_2),S_2]} \mathbf{0} \wedge\ (\rho, \kappa, \Gamma) \models^{\delta.[(r_1,\varphi_1,\eta_1),S_1]} P \\
\\
\text{iff} \\
(\rho, \kappa, \Gamma) \models^\delta (r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \wedge\ (\rho, \kappa, \Gamma) \models^{\delta.[(r_1,\varphi_1,\eta_1),S_1]} P \\
\text{iff} \\
(\rho, \kappa, \Gamma) \models^\delta (r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \wedge\ (\rho, \kappa, \Gamma) \models^\delta (r_1, \varphi_1, \eta_1)\{P\}^{S_1} \\
\text{iff} \\
(\rho, \kappa, \Gamma) \models^\delta (r_2, \varphi_2, \eta_2)\{\mathbf{0}\}^{S_2} \parallel (r_1, \varphi_1, \eta_1)\{P\}^{S_1}
\end{gathered}
$$

□

Subject Reduction. The analysis correctly captures the behaviour of P with respect to the semantics, i.e. the estimate (ρ, κ, Γ) is valid for all the derivatives P' of P. First, we prove its correctness for immediate derivatives, i.e. a single-step evolution of P.

Lemma 4.2.3 (Subject Reduction). *If $(\rho, \kappa, \Gamma) \models^\delta P$ and $P \xrightarrow{\mu} P'$, then we have:*

(1) If $\mu = \tau$, then $(\rho, \kappa, \Gamma) \models^\delta P'$,

(2) If $\mu = \alpha(r)$, then $(\rho, \kappa, \Gamma) \models^{\delta} P'$,

(3) If $\mu = rel(r)$, then $(\rho, \kappa, \Gamma) \models^{\delta} P'$,

(4) If $\mu = \alpha?r$, then:

- *If $[r, \varphi, \eta, S\chi] \in \delta \wedge \eta.\alpha(r) \models \varphi$, then $(\rho, \kappa, \Gamma) \models^{\delta'} P'$*
- *If $[(r, \varphi, \eta), S\chi] \in \delta \wedge \eta.\alpha(r) \not\models \varphi$, then $[(\varphi, \eta.err\_out(\chi)), S\chi] \in \Gamma(r)$ and $(\rho, \kappa, \Gamma) \models^{\delta''} P'$*
- *If $[(r, \varphi, \eta), S\chi] \notin \delta$, then $(\rho, \kappa, \Gamma) \models^{\delta} P'$,*

where $\delta' = \delta\{[(r, \varphi, \eta.\alpha), S\chi]/[(r, \varphi, \eta), S\chi]\}$ and $\delta'' = \delta \setminus [(r, \varphi, \eta), S\chi]$,

(5) If $\mu = rel?r$, then:

- *If $[(r, \varphi, \eta), S\chi] \in \delta$, then $(\rho, \kappa, \Gamma) \models^{\delta'} P' \wedge [(\varphi, \eta.rel.out(\chi)), S\chi] \in \Gamma(r)$*
- *If $[(r, \varphi, \eta), S\chi] \notin \delta$, then $(\rho, \kappa, \Gamma) \models^{\delta} P'$,*

where $\delta' = \delta \setminus [(r, \varphi, \eta), S\chi]$,

(6) If $\mu = x(y)$, then $(\rho, \kappa, \Gamma) \models^{\delta} P'$ and $\forall a \in \rho(x) : \kappa(a) \cap \mathcal{N} \subseteq \rho(y)$,

(7) If $\mu = x(s)$, then $\forall a \in \rho(x) : \kappa(a) \cap \mathcal{R} \subseteq \rho(s)$ and $\forall r \in \rho(s) : (\rho, \kappa, \Gamma) \models^{\delta} P\{r/s\}$,

(8) If $\mu = \bar{x}y$, then $(\rho, \kappa, \Gamma) \models^{\delta} P'$ and $\forall a \in \rho(x) : \rho(y) \subseteq \kappa(a)$,

(9) If $\mu = \bar{x}(y)$, then $(\rho, \kappa, \Gamma) \models^{\delta} P'$ and $\forall a \in \rho(x) : \rho(y) \subseteq \kappa(a)$.

Proof. The proof is by induction on the depth of the construction of $P \xrightarrow{\pi} P'$.

The case (1). Clearly the rules [Act], [Act$_\mathrm{R}$], [Open] and [Policy$_1$] do not apply. Thanks to Lemma 4.2.2 and the induction hypothesis (1), it is easy to prove that the property is preserved by the rules [Choice], [Par], [Res], [Cong], [Local$_1$] and [Local$_2$]. The remaining cases are [Comm], [Comm$_\mathrm{R}$], [Close], [Policy$_2$] and [Acquire].

- [Comm]: We may assume that $P \equiv P_1 \parallel P_2$ such that $P_1 \xrightarrow{\bar{x}w} P_1'$ and $P_2 \xrightarrow{x(y)} P_2'$. By Lemma 4.2.2, we have $(\rho, \kappa, \Gamma) \models^{\delta} P_1$ and $(\rho, \kappa, \Gamma) \models^{\delta} P_2$. The induction hypothesis of cases (6) and (8) ensure that

$$(\rho, \kappa, \Gamma) \models^{\delta} P_1' \wedge (\rho, \kappa, \Gamma) \models^{\delta} P_2' \wedge w \in \rho(y)$$

  By Substitution Lemma, we have $(\rho, \kappa, \Gamma) \models^{\delta} P_2'\{w/x\}$. This establishes $(\rho, \kappa, \Gamma) \models^{\delta} P_1' \parallel P_2'\{w/x\}$ and this is equivalent to $(\rho, \kappa, \Gamma) \models^{\delta} P'$.

- [Comm$_\mathrm{R}$]: by a similar argument.

- [Close]: by a similar argument.

- [Policy$_2$]: Assuming that $(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta)\{P\}^S$, $P \xrightarrow{\alpha?r} P'$ and $\eta.\alpha(r) \not\models \varphi$. We need to show that $(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta.err\_out(\chi))\{\mathbf{0}\}^S || P'$, where χ is the last label in S (Recall that $err\_out(\chi)$ is the special action used in CFA and does not change the operational semantics. Here we can safely assume that when forcing to release a resource, $err\_out(\chi)$ is appended to the history of that resource). By assumption, we have:

  $$(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta)\{P\}^S$$
  $$\text{iff}$$
  $$(\rho, \kappa, \Gamma) \models^{\delta.[(r,\varphi,\eta),S]} P$$

  By induction hypothesis (4), $[(r, \varphi, \eta), S\chi] \sqsubseteq \delta$ and $\eta.\alpha(r) \not\models \varphi$ implies that $[(r, \varphi, \eta.err\_out(\chi)), S\chi] \in \Gamma(r) \wedge (\rho, \kappa, \Gamma) \models^{\delta} P'$, which is equivalent to

  $$(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta.err\_out(\chi))\{\mathbf{0}\}^S \wedge (\rho, \kappa, \Gamma) \models^{\delta} P'$$

  And this establishes the required result.

- [Acquire]: Assuming that $(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta)\{\mathbf{0}\}^S \parallel req(r)\{Q\}^{\chi}$.
  We need to prove $(\rho, \kappa, \Gamma) \models^{\delta.[(r,\varphi,\eta.in(\chi)),S\chi]} Q$, which implies that

  $$(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta.in(\chi)))^S\{Q\}$$

  By assumption, we have

  $$(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta)\{\mathbf{0}\}^S \text{ and therefore } [(\varphi, \eta), S] \in \Gamma(r)$$

  Furthermore, $(\rho, \kappa, \Gamma) \models^{\delta} req(r)\{Q\}^{\chi}$ implies that

  $$(\rho, \kappa, \Gamma) \models^{\delta.[(r,\varphi,\eta.in(\chi)),S\chi]} Q$$

  The required result is established.

- [Release]: Assuming that $(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta)\{P\}^S$, $P \xrightarrow{rel?r} P'$. We need to show that $(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta.rel.out(\chi))\{\mathbf{0}\}^S || P'$, where χ is the last label in S (Recall that $out(\chi)$ is a special action used in CFA and does not change the operational semantics. Here we can safely assume that when releasing a resource, $out(\chi)$ is appended to the history of that resource) By assumption, we have:

  $$(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta)\{P\}^S$$
  $$\text{iff}$$
  $$(\rho, \kappa, \Gamma) \models^{\delta.[(r,\varphi,\eta),S]} P$$

  By induction hypothesis (5), $[(r, \varphi, \eta), S] \sqsubseteq \delta$ implies that $[(r, \varphi, \eta.out(\chi)), S] \in \Gamma(r) \wedge (\rho, \kappa, \Gamma) \models^{\delta} P'$, which is equivalent to

  $$(\rho, \kappa, \Gamma) \models^{\delta} (r, \varphi, \eta.out(\chi))\{\mathbf{0}\}^S \wedge (\rho, \kappa, \Gamma) \models^{\delta} P'$$

  And this establishes the required result.

The cases (2), (3), (4), (5), (6), (7), (8) and (9) are straightforward. □

Now we can state the subject reduction theorem.

Theorem 4.2.4 (Subject Reduction)**.** *$(\rho, \kappa, \Gamma) \models^\delta P$ and $P \xrightarrow{\mu}^* P'$, then $(\rho, \kappa, \Gamma) \models^\delta P'$.*

Proof. Immediate from Lemma 4.2.3. □

4.2.2 Existence of Estimates

We can further prove that there always exists a least choice of (ρ, κ, Γ) that is acceptable for CFA rules, and therefore it always exists a least estimate. First, we need some auxiliary definitions. It is immediate that the set of all estimates forms a complete lattice with the following order.

Definition 4.2.5. A set of proposed estimates can be partially ordered by setting $(\rho, \kappa, \Gamma) \sqsubseteq (\rho', \kappa', \Gamma')$ if and only if $\forall w \in \mathcal{N} \cup \mathcal{R} : \rho(w) \subseteq \rho'(w)$, $\forall a \in N : \kappa(a) \subseteq \kappa'(a)$, $\forall r \in R : \Gamma(r) \subseteq \Gamma'(r)$. Furthermore, we denote $(\rho, \kappa, \Gamma) \sqcap ESTPRIM$ as the least estimate of (ρ, κ, Γ) and $(\rho', \kappa', \Gamma')$ (a similar notation for a set of estimates).

To prove the existence theorem, we show that the set of all estimates for a given process constitutes a Moore family.

The following theorem guarantees that there always exists a least solution to the specification in Tab. 4.1.

Theorem 4.2.6. ***(Existence of estimates)*** *For all δ, P, the set*

$$\{(\rho, \kappa, \Gamma) | (\rho, \kappa, \Gamma) \models^\delta P\}$$

is a Moore family.

Proof. We proceed by structural induction on P. Let $L = \{(\rho_j, \kappa_j, \Gamma_j) | j \in J\}$ and

$$J \subseteq \{(\rho, \kappa, \Gamma) | (\rho, \kappa, \Gamma) \models^\delta P\}$$

and define $(\rho', \kappa', \Gamma') = \sqcap L = \cap\{(\rho_j, \kappa_j, \Gamma_j) | j \in J\}$. The proof of the theorem amounts to checking $(\rho', \kappa', \Gamma') \models^\delta P$. For this we proceed by cases on P making use of the induction hypothesis. Most cases are straightforward and here we only consider the more interesting ones.

The case of $x(y).P$. Since $\forall j \in J : (\rho_j, \kappa_j, \Gamma_j) \models^\delta x(y).P$, we have

$$\forall j \in J : (\rho_j, \kappa_j, \Gamma_j) \models^\delta P \wedge \forall a \in \rho_j(x) : \kappa_j(a) \cap \mathcal{N} \subseteq \rho_j(y)$$

Using the induction hypothesis and the fact that $(\rho', \kappa', \Gamma')$ is obtained in a pointwise manner, we then obtain

$$(\rho', \kappa', \Gamma') \models^{\delta} P \wedge \forall a \in \rho'(x) : \kappa'(a) \cap \mathcal{N} \subseteq \rho'(y)$$

thus establishing the desired $(\rho', \kappa', \Gamma') \models^{\delta} x(y).P$.

The case of $\alpha(r).P$. Since $\forall j \in J : (\rho_j, \kappa_j, \Gamma_j) \models^{\delta} \alpha(r).P$ we have

$$\begin{array}{ll} [(r, \varphi, \eta), S\chi] \in \delta \wedge \eta.\alpha(r) \models \varphi & \Rightarrow (\rho_j, \kappa_j, \Gamma_j) \models^{\delta'} P \\ [(r, \varphi, \eta), S\chi] \in \delta \wedge \eta.\alpha(r) \not\models \varphi & \Rightarrow \left\{ \begin{array}{l} (\rho_j, \kappa_j, \Gamma_j) \models^{\delta''} P \wedge \\ {[(\varphi, \eta.err\_out(\chi)), S\chi] \in \Gamma_j(r)} \end{array} \right. \\ [(r, \varphi, \eta), S\chi] \notin \delta & \Rightarrow (\rho_j, \kappa_j, \Gamma_j) \models^{\delta} P \end{array}$$

Using the induction hypothesis and the fact that $(\rho', \kappa', \Gamma')$ is obtained in a pointwise manner, we then obtain

$$\begin{array}{ll} [(r, \varphi, \eta), S\chi] \in \delta \wedge \eta.\alpha(r) \models \varphi & \Rightarrow (\rho', \kappa', \Gamma') \models^{\delta'} P \\ [(r, \varphi, \eta), S\chi] \in \delta \wedge \eta.\alpha(r) \not\models \varphi & \Rightarrow \left\{ \begin{array}{l} (\rho', \kappa', \Gamma') \models^{\delta''} P \wedge \\ {[(\varphi, \eta.err\_out(\chi)), S\chi] \in \Gamma'(r)} \end{array} \right. \\ [(r, \varphi, \eta), S\chi] \notin \delta & \Rightarrow (\rho', \kappa', \Gamma') \models^{\delta} P \end{array}$$

Hence we obtain the desired $(\rho', \kappa', \Gamma') \models^{\delta} \alpha(r).P$. □

4.2.3 Policy Compliance

Our analysis offers information on the resource usage, included bad usages. The component Γ is indeed in charge of recording all the possible usage traces on each resource r. Actually, for each r, traces are composed of pairs $[(\varphi, \eta), S]$, where S is made of labels of the processes that acquired resource r and η records every action on r, included the special actions $in(\chi)$, $out(\chi)$ and $err\_out(\chi)$, that indicate that the process labelled χ may acquire and release (or it may be forced to release) the resource. This information offers a basis for studying dynamic properties, by suitably handling the safe over-approximation that the CFA introduces. We want to focus now on the traces including special error actions, that we call *faulty.*

Definition 4.2.7. A trace $\eta \in \widehat{\mathcal{A}}^*$ is *faulty* if it includes $err\_out(\chi)$ for some $\chi \in \mathcal{L}$.

Because of over-approximation, on the one hand if the analysis contains faulty traces, then there is the *possibility* of policy violations, while if all the traces are not faulty, then we can prove that policy violations cannot occur at run time, and therefore that the processes correctly use their resources. We can show it formally, as follows.

Definition 4.2.8. The process P, where r is declared with policy ϕ, P *complies with* φ for r, if and only if $P \xrightarrow{\mu}^* P'$ implies that there is no P'' such that $P' \xrightarrow{\overline{\alpha(r)}} P''$, where $\xrightarrow{\mu}^*$ is the reflexive and transitive closure of $\xrightarrow{\mu}$.

Definition 4.2.9. A process P, where r is declared with policy φ, is said to *respect* φ for r, if and only if

$$\exists(\rho,\kappa,\Gamma).(\rho,\kappa,\Gamma)^{[\epsilon,\epsilon]}P \text{ and } \forall[(\varphi,\eta),S]\in\Gamma(r),\ \eta \text{ is not faulty}$$

The below policy compliance theorem states that if a process respects a policy then it complies with that policy.

Theorem 4.2.10 (Policy Compliance)**.** *If P* respects *the policy φ for r then, P* complies with *φ.*

Proof. By the way of contradiction, suppose that P does not comply with φ, e.g there exists P', P'' such that $P \xrightarrow{\mu}{}^{*} P' \overset{\overline{\alpha(r)}}{\rightarrow} P''$, where $\alpha(r)$ is the first violation action occurred in the sequence of transitions. The proof of the theorem amounts to checking that P does not respect φ for r, which implies a contradiction. By Theorem 4.2.4, we have that $(\rho,\kappa,\Gamma)\models^{[\epsilon,\epsilon]} P$ implies $(\rho,\kappa,\Gamma)\models^{[\epsilon,\epsilon]} P'$.

For this we proceed by cases on P' making use of the induction hypothesis. Most cases are straightforward and here we only consider one of the most interesting cases.

The case of $P' \equiv (r,\varphi,\eta)\{Q\}$, $Q \overset{\alpha?r}{\rightarrow} Q'$ and $\eta.\alpha \not\models \varphi$:

By Lemma 4.2.3, it follows that $[(\varphi,\eta.err\_out(\chi)), S\chi]\in\Gamma(r)$, which means that P does not respect φ for r. $\square$

Example 4.2.11. We briefly interpret the results of CFA on our running example. First we associate labels with the resource boundaries as follows:

$$\begin{array}{rl}
Res ::= & (r_1,\varphi_1,\epsilon)\{\mathbf{0}\}^{\epsilon} \parallel (r_1,\varphi_1,\epsilon)\{\mathbf{0}\}^{\epsilon} \parallel (r_2,\varphi_2,\epsilon)\{\mathbf{0}\}^{\epsilon}\\
Users ::= & x_1(s_1).req(s_1)\{\alpha(s_1).rel(s_1)\}^{\chi^1_\alpha} | x_2(s_2).req(s_2)\{\alpha(s_2).rel(s_2)\}^{\chi^2_\alpha} \parallel\\
& x_3(s_3).req(s_3)\{\alpha(s_3).rel(s_3)\}^{\chi^3_\alpha} \parallel y(t).req(t)\{\beta(t).rel(t)\}^{\chi_\beta}\\
Plan ::= & \bar{x_1}\langle r_1\rangle.\bar{y}\langle r_2\rangle.\bar{x_2}\langle r_2\rangle.\bar{x_3}\langle r_1\rangle.\mathbf{0}\\
System ::= & Res \parallel Users \parallel Plan
\end{array}$$

The CFA entries include:

- $\rho(s_1)\supseteq\{r_1\}$, $\rho(s_2)\supseteq\{r_2\}$, $\rho(s_3)\supseteq\{r_1\}$, $\rho(t)\supseteq\{r_2\}$; correspondingly:
- $\kappa(x_1)\supseteq\{r_1\}$, $\rho(x_2)\supseteq\{r_2\}$, $\rho(x_3)\supseteq\{r_1\}$, $\rho(y)\supseteq\{r_2\}$;
- $\Gamma(r_1)\supseteq\{(\varphi_1,\epsilon),(\varphi_1,in(\chi^i_\alpha).\alpha.rel.out(\chi^i_\alpha)),(\varphi_1,in(\chi^i_\alpha).\alpha.rel.out(\chi^i_\alpha).in(\chi^j_\alpha)$ $.\alpha.rel.out(\chi^j_\alpha)),(\varphi_1,in(\chi^i_\alpha).\alpha.rel.out(\chi^i_\alpha).in(\chi^j_\alpha).\alpha.rel.out(\chi^j_\alpha).in(\chi^k_\alpha)$ $.\alpha.rel.out(\chi^k_\alpha))\}$ (with $i,j,k\in[1,2,3]$ and i,j,k distinct);
- $\Gamma(r_2)\supseteq\{(\varphi_2,\epsilon),(\varphi_2,in(\chi_\beta).\beta.rel.out(\chi_\beta)),(\varphi_2,in(\chi_\beta).\beta.rel.out(\chi_\beta).in(\chi_\alpha)$ $.err\_out(\chi_\alpha)),(\varphi_2,in(\chi_\alpha).\alpha.rel.out(\chi_\alpha)),(\varphi_2,in(\chi_\alpha).\alpha.rel.out(\chi_\alpha).in(\chi_\beta)$ $.\beta.err\_out(\chi_\beta))\}$;

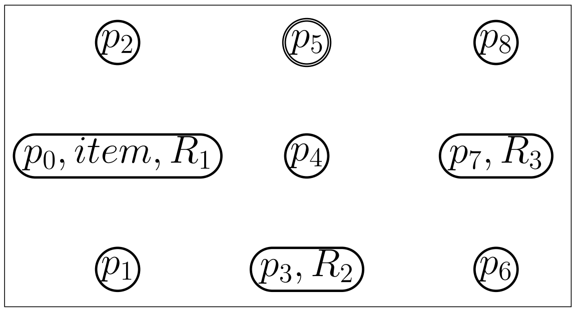

Figure 4.3: The initial configuration of the robot scenario.

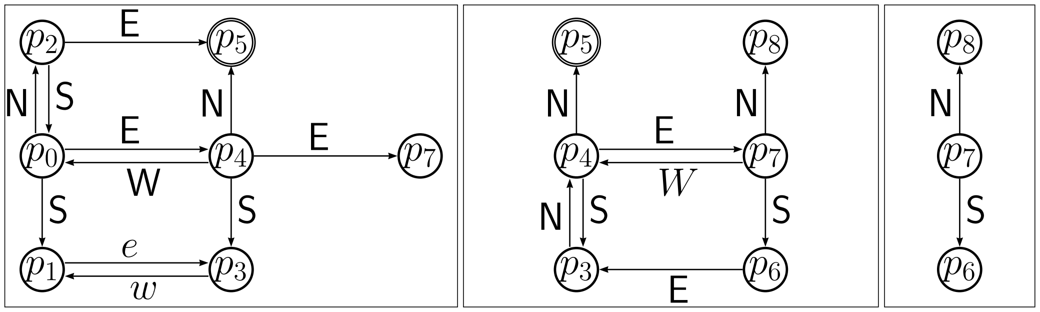

Figure 4.4: The policy automata of the robots' families: R_1 (left), R_2 (middle) and R_3 (right).

It is easy to see that there are at least two possible policy violations, which is captured by our CFA in the component $\Gamma(r_2)$. The first, given by th entry $(\varphi_2, in(\chi_\beta).\beta.rel.out(\chi_\beta).in(\chi_\alpha).err\_out(\chi_\alpha))$. and corresponds to the dynamic computation, developed in the previous section. It occurs when one of the user to perform an α action on r_2, whose cost, added to that of the previous action β, passes the fixed threshold.

4.3 A Case Study - Robot Scenario

We consider a scenario, where a set of robots collaborate to reach a certain goal, e.g. to move an item from one position to another. Without loss of generality, we assume that robots operate in a space represented by a two-dimensional grid. We also assume that certain positions over the grid are *faulty*, and therefore they cannot be crossed by robots. To move the item, a robot needs to take it, and this is allowed provided that the item is co-located within the range of robot's sensor. Moreover, since robots have a small amount of energy power, they can perform just a few of steps with the item. Finally, we consider three families of robots (R_1, R_2 and R_3): each robot in the family has different computational capabilities.

Fig. 4.3 gives a pictorial description of the initial configuration of the scenario. Positions are represented by circles and double circles. Double circles indicate faulty positions. The item is located at position p_0 and the goal is to move it to the position p_8. There is just one faulty position p_5, crossing through which is considered a failure. Moreover, we consider a scenario where the three families of robots R_1, R_2 and R_3 are initially located at p_0, at p_3 and at p_7, respectively (e.g. all the robots of the family R_1 are located at p_0).

Sensors are modelled by clearly identified resources. The sensor j^{th} of the i^{th} robot family is specified by the resource $(sns_{i,j}, \varphi_j, \eta_{i,j})$, where $sns_{i,j}$ is the name of the sensor, $\eta_{i,j}$ is the abstract representation of the sequence of moving actions which led the robot from its initial position to the current one and is initially equals to ϵ, and φ_j is the global policy on demand. We assume that each family of robots has its own policy described by the automata in Fig. 4.4. The policy constraints robots' movement in the grid. We model the movement activities of robots with the following actions: $\mathsf{E}(sns)$, $\mathsf{W}(sns)$, $\mathsf{S}(sns)$, and $\mathsf{N}(sns)$ that describe the movements on east (west, south and north, resp.). Basically, sensors are a sort of private resources of the robots (each robot will never release its sensor) and the actions over sensors update their states.

The item is modelled by a resource of the form (IT, φ_I, η), where η describes the sequence of actions performed on the item, and φ_I simply states that the item is never located at the position p_5. Initially, η is equal to ϵ. The same set of actions adopted for robots' movement (namely $\mathsf{E}(IT)$, $\mathsf{W}(IT)$, $\mathsf{S}(IT)$, and $\mathsf{N}(IT)$) are exploited to transport the item in the grid. Finally, each robot in the family $i \in \{1, 2, 3\}$ is specified by a process $R_{i,j}$ of the form: $(sns_{i,j}, \varphi_j, \eta_{i,j})\{Q_{i,j}\}^{\chi_{sij}}$, where $Q_{i,j}$ specifies the j^{th} robot's behaviour of the i^{th} robot family and χ_{sij} is a label associated with the resource boundary. For instance, in the process $Q_{2,3}$ (see below), the robot goes to north (without the item), then it tries to grasp the item. If this operation succeeds, the robot goes to east and releases the item there. Note that we use two monadic actions to move the item and the sensor together. This could be done by using polyadic actions, which however we leave for future work.

For the sake of simplicity, we do not model co-location of sensors and items. The specification of the robot scenario is given below.

$$
\begin{aligned}
&R_{1,1} := (sns_{1,1}, \varphi_1, p_0)\{req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{1,1}).\mathsf{S}(IT).\mathsf{S}(sns_{1,1}).rel(IT)\}^{\chi_{r11}}\}^{\chi_{s11}}\\
&R_{1,2} := (sns_{1,2}, \varphi_1, p_0)\{req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{1,2}).\mathsf{E}(IT).\mathsf{E}(sns_{1,2}).rel(IT)\}^{\chi_{r12}}\}^{\chi_{s12}}\\
&R_{1,3} := (sns_{1,3}, \varphi_1, p_0)\{req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{1,3}).rel(IT)\}^{\chi_{r13}}\}^{\chi_{s13}}\\
&R_{2,1} := (sns_{2,1}, \varphi_2, p_3)\{req(IT)\{\mathsf{N}(IT).\mathsf{N}(sns_{2,1}).\mathsf{E}(IT).\mathsf{E}(sns_{2,1}).rel(IT)\}^{\chi_{r21}}\}^{\chi_{s21}}\\
&R_{2,2} := (sns_{2,2}, \varphi_2, p_3)\{req(IT)\{\mathsf{N}(IT).\mathsf{N}(sns_{2,2}).\mathsf{N}(IT).\mathsf{N}(sns_{2,2}).rel(IT)\}^{\chi_{r22}}\}^{\chi_{s22}}\\
&R_{2,3} := (sns_{2,3}, \varphi_2, p_3)\{NR(sns_{2,3}).req(IT)\{\mathsf{E}(IT).\mathsf{E}(sns_{2,2}).rel(IT)\}^{\chi_{r23}}\}^{\chi_{s23}}\\
&R_{3,1} := (sns_{3,1}, \varphi_3, p_7)\{req(IT)\{\mathsf{S}(IT).\mathsf{S}(sns_{3,1}).rel(IT)\}^{\chi_{r31}}\}^{\chi_{s31}}\\
&R_{3,2} := (sns_{3,2}, \varphi_3, p_7)\{req(IT)\{\mathsf{N}(IT).\mathsf{N}(sns_{3,2}).rel(IT)\}^{\chi_{r32}}\}^{\chi_{s32}}
\end{aligned}
$$

$$
System := (IT, \varphi_I, p_0)\{\mathbf{0}\}^{\chi_{IT}} \parallel R_{1,1} \parallel R_{1,2} \parallel R_{1,3} \parallel R_{2,1} \parallel R_{2,2} \parallel R_{2,3} \parallel R_{3,1} \parallel R_{3,2}
$$

The following trace illustrates the behaviour of the specification of the scenario. At the beginning, the item lies in the range of the family of robot R_1. Then a reconfiguration step putting together the robot $R_{1,1}$ and the item is performed.

$$\begin{aligned} System &:= (IT, \varphi_I, \epsilon)\{\mathbf{0}\}||(sns_{1,1}, \varphi_1, \epsilon)\{Q_{1,1}\}||R_{1,2}||R_{1,3}||R_{2,1}||R_{2,2}||R_{2,3}||R_{3,1}||R_{3,2} \\ &\equiv (sns_{1,1}, \varphi_1, \epsilon)\{(IT, \varphi_I, \epsilon)\{\mathbf{0}\}||Q_{1,1}\}||R_{1,2}||R_{1,3}||R_{2,1}||R_{2,2}||R_{2,3}||R_{3,1}||R_{3,2} \end{aligned}$$

As a result, robot $R_{1,1}$ can grasp (acquire) the item; the pair item-robot moves on east, then on south. Finally, the robot disposes the item at the position p_3.

$$\begin{aligned} System \xrightarrow{\tau} \;& (sns_{1,1}, \varphi_1, p_0)\{(IT, \varphi_I, \epsilon)\{Q_{1,1}\}||R_{1,2}||R_{2,1}||R_{2,2}||R_{3,1}||R_{3,2} \\ & \xrightarrow{E(IT)}\xrightarrow{E(sns_{1,1})}\xrightarrow{S(IT)}\xrightarrow{S(sns_{1,1})}\xrightarrow{rel(IT)} \\ & (IT, \varphi_I, E.S.rel)\{\mathbf{0}\}||(sns_{1,1}, \varphi_1, E.S)\{\mathbf{0}\}||R_{1,2}||R_{2,1}||R_{2,2}||R_{3,1}||R_{3,2} \end{aligned}$$

It is easy, given an initial location, to map a sequence of actions performed over the item into a path on the grid, namely each action operated over the item (i.e. $\mathsf{E}(IT)$, $\mathsf{W}(IT)$, $\mathsf{S}(IT)$, and $\mathsf{N}(IT)$) corresponds to a single moving step in the space grid. The release action, instead, is interpreted as a sort of self-loop in the grid, i.e. the execution of the release action does not move the item. For example, the sequence $E.S.$ in the above setting would model the path $p_0p_4p_3$. From now on, by abuse of notation, we will freely use paths in place of sequences of actions over the item/sensors.

Now, the item is in the range of the family of robots R_2. Again by applying the reconfiguration step, robot $R_{2,1}$ is allowed to operate with the item. Then, it takes the item, makes a move on north, then on east, and disposes the item at the position p_7. For the sake of simplicity, in the following we show only sub-processes of the system that involve computation:

$$\begin{aligned} & (IT, \varphi_I, p_0p_4p_3p_3)\{\mathbf{0}\}||R_{2,1} \\ & \xrightarrow{\tau}\xrightarrow{N(IT)}\xrightarrow{N(sns_{2,1})}\xrightarrow{E(IT)}\xrightarrow{E(sns_{2,1})}\xrightarrow{rel(IT)} \\ & (IT, \varphi_I, p_0p_4p_3p_3p_4p_7p_7)\{\mathbf{0}\}||(sns_{2,1}, \varphi_2, p_3p_4p_7)\{\mathbf{0}\} \end{aligned}$$

Note that a forced release would have occurred at this step if the item proceeded governed by the robot $R_{2,2}$. The reason is that $R_{2,2}$ attempts to move the item into the position p_5 and this results in releasing the item at the position p_4 by the rule $Policy_2$. Now the robot $R_{3,2}$ has the chance to take the item, and, if the north move occurs, the goal is achieved and the task is completed.

$$\begin{aligned} & (IT, \varphi_I, p_0p_4p_3p_3p_4p_7p_7)\{\mathbf{0}\}||R_{3,2} \\ & \xrightarrow{\tau}\xrightarrow{N(IT)}\xrightarrow{N(sns_{3,2})}\xrightarrow{rel(IT)} \\ & (IT, \varphi_I, p_0p_4p_3p_3p_4p_7p_7p_8p_8)\{\mathbf{0}\}||(sns_{3,2}, \varphi_3, p_7p_8)\{\mathbf{0}\} \end{aligned}$$

Now we explain the features of the CFA. The CFA (in particular the Γ component) computes the set of possible traces of the trajectories in the grid reaching the goal, among which the ones below:

$in(\chi_{r11}).E.S.rel.out(\chi_{r11}).in(\chi_{r21}).N.E.rel.out(\chi_{r21}).in(\chi_{r32}).N.rel.out(\chi_{r32})$
$in(\chi_{r11}).E.E.rel.out(\chi_{r11}).in(\chi_{r32}).N.rel.out(\chi_{r32})$
$in(\chi_{r13}).E.rel.out(\chi_{r13}).in(\chi_{r23}).E.rel.out(\chi_{r23}).in(\chi_{r32}).N.rel.out(\chi_{r32})$
$in(\chi_{r11}).E.S.rel.out(\chi_{r11}).in(\chi_{r22}).N.err\_out(\chi_{r22}.in(\chi_{r23}).E.rel.out(\chi_{r23}).in(\chi_{r32}).N.rel.out(\chi_{r32})$

This set produces the following sequences of positions:

$$p_0p_4p_3p_3p_4p_7p_7p_8p_8$$
$$p_0p_4p_7p_7p_8p_8$$
$$p_0p_4p_4p_7p_7p_8p_8$$
$$p_0p_4p_3p_3p_4p_4p_7p_7p_8p_8$$

Note that the last trace is faulty since it contains a forced release $err\_out(\chi_{2,2})$ (see below). Consequently, the system *does not respect* the policy φ_{IT} for the item. In particular, there are three faulty traces found by the analysis, which have the following common prefix:

$$in(\chi_{r11}).E.S.rel.out(\chi_{r11}).in(\chi_{r22}).N.out\_err(\chi_{r22}), \chi_{r11}\chi_{r22}$$

The reason is that the robot $R_{2,2}$ is forced to release the item when attempting to move it into the bad position p_5. Instead, there is no faulty trace of actions over sensors, which means the system *respects* the policies $\varphi_{i,j}$ for sensors and therefore complies with them.

4.4 Related Works and Discussions

Understanding the foundations of the distributed management of resources can support state-of-the-art advances of programming language constructs, algorithms and reasoning techniques for resource-aware programming. In the last few years, the problem of providing the mathematical basis for the mechanisms for resource usages has been received a major attention. As a consequence, a number of models has been proposed (see e.g. [16, 36, 66, 47, 64], to cite only a few) for resource management.

The G-Local π-calculus design. We started the design of our calculus by adopting the history-based approach studied in [9, 56, 101, 18]. The history-dependent framework overcomes the weaknesses of stack based approaches [57] that record only a fragment of the trace instead of the whole trace (called *history*). In [16] an extension of the λ-calculus is proposed to statically verify resource usages. The work combines *local checks* of program points, where critical resources can be accessed, with *global policies*, which enforces a global invariant to hold at any program point. Our work is inspired by these works. We end up following an approach that also borrow many ideas from service oriented computing (SOC) [91]. The main theme of SOC is to design a general theory of services, often based on mobile calculi, for formalising and programming service-oriented applications [107, 33], for developing the suitable verification techniques [38, 60, 76]. Although our model is based on mobile calculi, we instead consider the orthogonal issue concerning the correctness of resource usages in modern distributed settings.

The novel feature of our proposal relies on the *interaction patterns* between resources and computational processes. The interactions between resources and computational processes are established through *publish-subscribe* paradigm. Notice that these features have been exploited in service oriented computing (SOC). It is worthwhile to stress which are differences of our design, compared to SOC. The emphasis on design of service orientation so far has been on description of interactions and development of related concepts, like context-awareness or service sessions. Indeed, invocation of "services" establishes a session, whose interactions follow a certain protocol through the standard mechanism of communication, provided by mobile calculi. In this sense, services could be considered as resources at a high level view. The notion of resources in our approach are represented by structures with states and usage polices. The novel feature of our approach relies on mechanisms related to *event-notifications* that intent to capture resource accesses. Usage policies related to individual resources provide indeed a flexible way to define fine-grain access control on resources. This emphasises the fact that the focus of our approach is on *correct usages of resources* rather than *discipline of interactions* like in SOC.

Resources in the G-Local π-calculus have scopes, that can be thought as resource *administrative domains*, similar to the scope of locations in Mobile Ambients [44]. Closer to ours is the work in [55], where an ambient is considered as a unit for monitoring and coordination. More precisely, each ambient is equipped with a guardian, which monitors the activities of sub-components (i.e. processes and sub-ambients). Unlike Mobile Ambients, the scope of resources is more restricted since the scopes cannot be *open*. Placing restriction on the scope of resources is a design decision, that makes the control of resource management easier. While the scopes of locations in Mobile Ambient are managed by explicit actions of processes, configurations of resources in our approach are not under control of processes. Furthermore, resources in our approach have no control over other resources. This assumption is justified in terms of loosely coupling design that are typical of modular distributed applications.

Models of resources. The simplest model of resources is given by the notion of *names*. As seen in Chapter 2, in name-passing process calculus, names can be communicated and exchanged. In the case of ambient-like calculi, names assume the role of locations. Thus, it is natural to treat names as resources in many process-based approaches [108, 70, 104, 105]. In [70], resource usages are simply bounds on the number of communications in channels. The work presented in [105] focuses on the ownership and publication of names. In [108, 104], allocation/deallocation of resources are the interesting properties. In more details, in [108] reconfigurations steps are internalised inside processes via the operations for allocating and de-allocating channels. A similar idea is found in [104], where closer explicit transition rules for eliminating *dead* processes through different *garbage-collectable* relations on processes. The drawback of these works is that properties of resources are quite limited. In [63], more advanced properties of names are introduced and studied, where a logic is introduced to express the relations between processes and resources. In our approach resources are structured in a such way that it is possible to reason on the actual states of resources.

The π-calculus dialect of [66] provides a general framework for checking resource usages in distributed systems. The treatment of resources in this approach is closer to ours in terms of properties of resources. Indeed, private names are extended to resources, i.e. names with a set of traces to define control over resources. Also resource request and resource release are simulated through communicating private names and structural rules respectively. There is a shared semantics of resources, i.e. several processes can have a concurrent access to resources (by communication of private names). Resources in our approach may be considered as names with additional structures, however they cannot be *private*. Unlike private names with the dynamic boundaries given by the scope extrusion, resource boundaries is based on explicit acquisitions. Our approach also differs from this work in the semantic treatment of resources; when a process obtains a resource, it has an exclusive access to the resource.

The works in [47, 36] present an explicit notion of resources, different from our notion. The work in [47] proposed a process calculus with an explicit representation of resources in which the evolution of processes and resources happens in a SCCS style. More precisely, resources form a monoid, that is, a set of elements with a binary operator. Thus, a resource can be non-deterministically split into *smaller* pieces (by the binary operator defined in the monoid) to be distributed among processes. In this way, the notion of resources is closely related to the sharing interpretation of the **BI**-family logics (see [88] for details). Also, a modal logic, based on the **BI**-family logics, is developed to specify resource properties. The drawback of this approach is the co-evolution of processes and resources. It requires a pre-defined model of resources (also taking process evolution into account), which is sometimes difficult to define. In our approach, resources are independent and stateful entities, thus subject to be requested, and are equipped with their own global interaction usage policies, defined as a set of traces. Therefore, LTL formulas or equivalent formalisms are used to specify temporal properties of resources.

The work presented in [36] mainly focuses on specifying SLA by describing resources as suitable constraints. In this proposal, c-semirings [24] act as a model of resources. The shared store of constraints on resources represents SLA contracts established through allocation and deallocation of resources. C-semirings allow for expressing *soft* constraints, i.e. constraints which give informative values instead of true or false. Similar to ours, available resources are obtained though suitable requests. It is easy to see that we may exploit constraints to express global resource usages as well.

Control Flow Analysis. We developed a Control Flow Analysis. The CFA computes an approximation of communication-based and finite resource-based behaviour. First, resource-based behaviour is described with their possible traces and configurations (i.e. the resource contexts). An analysis for processes with infinite behaviour is subject of future work. The CFA presented in [29] is closest to ours, where it takes into account context information. Here, we focus on reachability properties of resources, while the work in [29] deals with interactions among encapsulated processes.

Programming Abstractions for Resource Awareness. Abstraction mechanisms, like objects, classes or abstract data types ... have introduced in programming languages are often motivated by the idea of helping developers to focus on the software design. Indeed, focusing on business logic of software systems allows for freeing developers from implementation details. For example, the "try-with-resources" statement has recently been introduced in JDK7 [89] to help developers in ensuring proper termination over resources. The separation of concerns between business logic and operation logic is indeed the main focus of our work. The term "operation logic", coined in [23], refers to resource management in distributed settings, where applications are capable of accessing a variety of resources.

Our work is the initial attempt to provide a foundational basis for resource-aware programming abstractions. Here, we outline how our approach suggests some resource-aware programming abstractions. For purpose of illustration, we use informal notations. Let us assume to have a storage service. The declaration of the resource, seen as storage, could be given as follows

```
Def Resource storage {
        State: eta;
        Ensuring: P;
        Behaviour: Null
        }
```

The previous declaration introduces a stateful resource whose name is `storage`, whose state information is stored in variable `eta` and where `P` describes the policy (SLA) any interaction with the storage has to satisfy. In the initial configuration, the encapsulated behaviour is empty: it will be instantiated with the code provided by the client, when the binding between the client and the resource is established. In this simple example, the policy expresses that the correct way of operating over the storage resource is that in all the sessions action *open* must occur before action *write*. Every attempt to access the resource must obey this policy. The resource pool is implemented as parallel composition of the resources. Furthermore, assume that the resource is dynamically allocated to the client requests on demand is handled by the resource pool.

```
\\ The script describing the main program
\\ of the client process is shown below
c = connect(resource-pool);
ss=c.receive(storage)
//internal activity)
bind(ss) {
   ss.open(clientName);
   // produce some data
   ss.write(clientName,data);
   ss.release(clientName);
}
//Other work
```

The meaning of the script is intuitive. The client process establishes a connection with the the resource pool, then binds the local variable `ss` with the actual resource and operates over it.

Closest to our idea of resource-aware programming are the works presented in [5, 103]. The approach in [5] introduces a *typesafe-oriented programming language* by extending the object paradigm with *object states*. Objects are modelled in terms of *changing states*, rather than *classes*. The development of the resource-aware programming *Plaid*, introduced in [103], follows the idea outlined above. *Typesafe programming*, originally introduced in [102], expresses that each state of an object has its own representation and methods (only these methods are available at the object in this state) and may lead the object into a new state. The dynamics of states allows developers to control program behaviour. In this sense, objects could be though as be linked to a security policy to restrict the transitions of objects from one state to another. In broader view of resources, objects can be considered as resources, and therefore look similar to our notion of resources. The essential difference between the typesafe approach and ours is that resources in our approach are independent entities and we rely on loose coupling design to separate fragments of code that use resources, from resources themselves, rather than scattering fixed operations through a set of states of objects. More precisely, access control over resources in the programming language [103] is more fine-grained than ours. This is because the access control in [103] includes shared, immutable and exclusive accesses, while our approach supports only exclusive resource accesses.

Future Work. A number of extensions is possible. Here we outline some of them for future work.

- **Model of Resources.** We have defined resource policies as a set of traces. It is quite useful to consider other formalisms such as c-semirings to exploit constraint-based approach. Due to the nature of publish-subscribe paradigm, we loose *privacy* of resources, i.e. every process knowing resource names potentially access them. Instead of taking private names as resources, we extend resources with privacy, that is, r can be also a private name. For instance, consider the following example:

$$P ::= (\nu r)((r, \varphi, \eta)\{\mathbf{0}\} \parallel (r, \varphi, \eta)\{\mathbf{0}\} \parallel Q),$$

representing the situation where two identical resources are available under a private name r for being used by processes knowing r (Q in this case). The nature of private names with notion of group is not new, however the new treatment of resource privacy could give a closer view of resource usages. We believe that this view could potentially separate two important aspects of resource usages:

i) an internal view, which focuses on the states of resources;

ii) an external view, which focuses on external information by processes that request resources.

- **Elasticity of resources vs Replication.** Still in the spirit of the rules [Appear] and [Disappear] of the Remark 4.1.5, one can advocate the structural rule for the replication for the same purpose, however it requires that resource instances must match exactly their syntax. More precisely, the replication can be seen as the two following transition:

$$\begin{array}{ll} [\text{Rep\_1}] & !(r, \varphi, \eta)\{\mathbf{0}\} \xrightarrow{\tau} !(r, \varphi, \eta)\{\mathbf{0}\} \parallel (r, \varphi, \eta)\{\mathbf{0}\} \\ [\text{Rep\_2}] & !(r, \varphi, \eta)\{\mathbf{0}\} \parallel (r, \varphi, \eta)\{\mathbf{0}\} \xrightarrow{\tau} !(r, \varphi, \eta)\{\mathbf{0}\} \end{array}$$

This is only true for a resource, where its instances have the same state. In general, it could not be used for resource instances with different states.

- **Polyadic Requests.** It is desirable to have a polyadic requests since obtaining a bunch of resources is often seen as a scenario in cloud systems. Unlike polyadic input/outputs, where synchronisation involves only two parties, polyadic resource requests make transition rules more complicated as they involve multiple parties.

- **Resource Movement.** The structural rule for resource management is unconditional. The closer view could specify some conditions to restrict the movement of resources. For instance, in the robot scenario, it would be nice if only a robot located at the same location of the item can take it. The structural rule would have the following form:

$$(r_2, \varphi_2, \eta_2)\{\mathbf{0}\} \parallel (r_1, \varphi_1, \eta_1)\{P\} \equiv (r_1, \varphi_1, \eta_1)\{(r_2, \varphi_2, \eta_2)\{\mathbf{0}\} \parallel P\} \text{ if } comp(\eta_1, \eta_2),$$

where $comp(\eta_1, \eta_2)$ is a condition function on the states of the two resources. Basically if it is true, it allows them to cross each others. This feature was introduced in [32].

- **CFA implementation.** Developing an efficient algorithm for computing CFA estimates has not been addressed yet. To this purpose, we need suitable abstractions of histories in order to reduce the complexity. Such issues are left for future work.

Chapter 5

The Type and Effect System for the G-Local π-Calculus

In this chapter, we develop a second static technique for the G-Local calculus. First, an extension of G-Local π-calculus is presented for developing behavioural type system. Starting from the behavioural types for the standard π-calculus [4], types are equipped with resource-access actions $\alpha(R)$, where R is a finite set of resource names over which α possibly acts.

Basically, types abstract two kinds of behavioural information: communication-based and resource-based. The former describes the abstract behaviour of processes on channels, i.e. the dependencies and interactions among channels as described in [4, 62], while the latter describes the abstract resource behaviour of processes [66]. Our approach keeps the abstract behaviour on communications of the extended type system as close as possible to [4]. At the same time, the type system allows to extract the resource behaviour as side effects. To this end, a symmetric treatment of input/output on channels is required. Moreover, effects are expressed in terms of BPP processes, which are then used to model check regular linear time properties of resource usages of processes.

5.1 Extension of the G-Local π-Calculus

The development of behavioural types that will be introduced in the next section requires a minor extension of the G-local π-calculus. Following the approach of [4], processes are annotated with type information. More precisely, the restrictions are annotated with channel types, which will be defined in the next section, and a different treatment of replication is used to reflect the behavioural correspondence between processes and types. Again we reuse some notation introduced in the previous chapters to describe resource and access actions. we introduce again the needed notation and also the version of G-Local calculus, adapted for handling types. The following notations will be used through this chapter.

Definition 5.1.1. Assume that $\mathcal{N}$ is a set of channel names (ranged over by a, b, x, y, z), $\mathcal{R}$ is a set of resource names (ranged over by r, s), $\mathcal{A}$ is a set of actions (ranged over by α, β) for running over resources, and Φ is a set of policies (ranged over by φ, φ'). Policies are defined by LTL formulas as illustrated in Chapter 2. A special action $rel \notin \mathcal{A}$ is also assumed. We use w, w' to range over channel and resource names. We assume that these sets are pairwise disjoint.

Definition 5.1.2. The set $\mathcal{P}_{egl}$ of extended processes is defined as in def. 4.1.2. There are two main exceptions: guarded replication is used and $(\nu z)P$ is replaced by $(\nu z : t)P$. For completeness, we report the syntax below.

$$
\begin{array}{llll}
P, P' & ::= & & \textit{processes} \\
 & & \mathbf{0} & \text{empty process} \\
 & \mid & \pi.P & \text{prefix action} \\
 & \mid & P + P' & \text{choice} \\
 & \mid & P \parallel P' & \text{parallel composition} \\
 & \mid & (\nu x : t)\ P & \text{restriction} \\
 & \mid & !a(w).P & \text{replication} \\
 & \mid & (r, \varphi, \eta)\{P\} & \text{resource joint point} \\
 & \mid & req(r)\{P\} & \text{resource request point} \\
\pi, \pi' & ::= & & \textit{action prefixes} \\
 & & a(w) & \text{free input} \\
 & \mid & \bar{a}w & \text{free output} \\
 & \mid & \tau & \text{internal action} \\
 & \mid & \alpha(s) & \text{event action} \\
 & \mid & rel(s) & \text{event action}
\end{array}
$$

The notions of names $\mathsf{n}()$, free names $\mathsf{fn}()$, bound names $\mathsf{bn}()$ and substitution $\{-/-\}$ are defined as expected, except for the set of free names of annotated restrictions, where $\mathsf{fn}((\nu x : t)P) = \mathsf{fn}(t) \cup \mathsf{fn}(P) \setminus \{x\}$.

As in Section 4.1.1, we assume a notion of structural congruence, denoted by $\equiv$. The structural congruence is the relation, reported in Fig. 5.1, that includes the standard laws of the π-calculus, such as the monoidal laws for the parallel composition and the choice operator. To preserve type information, as in [4, 63], we require only the following rule for restriction: $(\nu x : t)(P|Q) \equiv (\nu x : t)P|Q$, if $x \notin \mathsf{fn}(Q)$. Notice that we do not have a structural rule for replication, but an operational one.

The operational semantics of the calculus, reported in Fig. 5.2, extends the standard semantics of π-calculus with suitable rules to deal with resource constructs. Transitions are annotated with labels as in Chapter 4. We use the transition label $\langle a \rangle$ for communications. In the next section, we will see that communications occur in the corresponding types of a process.

The operational semantics of the calculus, reported in Fig. 5.2 Apart from replication, the operational semantics are the same introduced in Chapter 4. The operational rule [Rep_Comm] is added to handle the treatment of the replication. More

$$
\begin{array}{ll}
P \equiv Q \text{ if } P \text{ and } Q \text{ are } \alpha\text{-equivalent} & \\
(P + Q) + R \equiv P + (Q + R) & (P \parallel Q) \parallel R \equiv P \parallel (Q \parallel R) \\
P + Q \equiv Q + P & P \parallel Q \equiv Q \parallel P \\
P + \mathbf{0} \equiv P & P \parallel \mathbf{0} \equiv P \\
(\nu x : t)P \parallel Q \equiv (\nu x : t)(P \parallel Q) \ x \notin \mathsf{fn}(Q), & \\
 & \\
(r, \varphi, \eta)\{P_1\} \equiv (r, \varphi, \eta)\{P_2\} \text{ if } P_1 \equiv P_2 & \\
req(r)\{P_1\} \equiv req(r)\{P_2\} \text{ if } P_1 \equiv P_2 & \\
(\nu x)(r, \varphi, \eta)\{P\} \equiv (r, \varphi, \eta)\{(\nu x)P\} & \\
(\nu x)req(r)\{P\} \equiv req(r)\{(\nu x)P\} & \\
(r_2, \varphi_2, \eta_2)\{\mathbf{0}\} \parallel (r_1, \varphi_1, \eta_1)\{P\} \equiv (r_1, \varphi_1, \eta_1)\{(r_2, \varphi_2, \eta_2)\{\mathbf{0}\} \parallel P\} &
\end{array}
$$

Figure 5.1: Structural congruence.

precisely, an instance of a replication is instantiated when the replication performs an action. A further small difference is that we use the transition label $\langle a \rangle$ for communications.

In the next section, we introduce a type and effect system with the aim of guaranteeing that well-typed processes never violate the required resource policies.

5.2 The syntax and semantics of types

This section introduces a type and effect system for the G-local π-calculus. The type system prevents policy violations, and it is inspired by the type systems in [4, 14, 16]. In our proposal, effects are resource-based behavioural abstractions that annotate types. The syntax of types is described by the following syntax.

5.2.1 Syntax of types

Definition 5.2.1. Assume that $\mathcal{V}_T$ is a set of type variables, ranged over by X, Y. We use Θ, ranged over by θ, θ', to denote a set of finite set of pairs of resource names and policies $\langle r, \varphi \rangle$.

$$
\begin{array}{lll}
(\textit{process types})\ T, T' & ::= \mathbf{0} \mid X \mid \pi.T \mid (\nu x : t)\, T \mid T + T' \mid T \parallel T' \mid !\pi.T \\
(\textit{prefix actions})\ \pi, \pi' & ::= \tau \mid a(t) \mid \overline{a} \mid \alpha(R) \\
 & \\
(\textit{resource types})\ u, u' & ::= res(\theta) \\
(\textit{channel types})\ t, t' & ::= (x : t)T \mid (s : u)T \mid ()T, \text{ where } x \text{ is a free name in } T\ ,
\end{array}
$$

where R is a non-empty finite set of resource names. With abuse of the notation, we use π to denote the prefix actions. Function $rn(\theta)$ is defined to be the set of resource names in θ, i.e. $rn(\theta) = \{r | \langle r, \varphi \rangle \in \theta\}$. A resource type $res(\theta)$ describes the set of associations $\langle r, \varphi \rangle$ between resource names and usage policies. For the

$$[\text{Act}] \quad \pi.P \xrightarrow{\pi} P \qquad \pi \neq \alpha(r), rel(r) \qquad\qquad [\text{Cong}] \quad \frac{P_1 \equiv P_1' \quad P_1' \xrightarrow{\mu} P_2' \quad P_2' \equiv P_2}{P_1 \xrightarrow{\mu} P_2}$$

$$[\text{Par}] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \parallel P_2 \xrightarrow{\mu} P_1' \parallel P_2} \ \mathsf{bn}(\mu) \cap \mathsf{fn}(P_2) = \emptyset \qquad\qquad [\text{Choice}] \quad \frac{P_1 \xrightarrow{\mu} P_1'}{P_1 + P_2 \xrightarrow{\mu} P_1'}$$

$$[\text{Res}] \quad \frac{P \xrightarrow{\mu} P'}{(\nu x : t)P \xrightarrow{\mu} (\nu x : t)P'} \ x \notin \mathsf{n}(\mu) \qquad\qquad [\text{Open}] \quad \frac{P \xrightarrow{\bar{a}x} P'}{(\nu x : t)P \xrightarrow{\bar{a}(x)} P'} \ x \neq a$$

$$[\text{Comm}] \quad \frac{P_1 \xrightarrow{\bar{a}y} P_1' \quad P_2 \xrightarrow{a(z)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\langle a \rangle} P_1' \parallel P_2'\{y/z\}} \qquad\qquad [\text{Close}] \quad \frac{P_1 \xrightarrow{a(z)} P_1' \quad P_2 \xrightarrow{\bar{a}(y)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\langle a \rangle} (\nu y)(P_1' \parallel P_2'\{y/z\})}$$

$$[\text{Act}_\text{R}] \quad \begin{array}{l} \alpha(r).P \xrightarrow{\alpha?r} P \\ rel(r).P \xrightarrow{rel?r} P \end{array} \qquad\qquad [\text{Comm}_\text{R}] \quad \frac{P_1 \xrightarrow{\bar{x}r} P_1' \quad P_2 \xrightarrow{x(s)} P_2'}{P_1 \parallel \ P_2 \xrightarrow{\langle a \rangle} P_1' \parallel P_2'\{r/s\}}$$

$$[\text{Rep\_Comm}] \quad !a(w).P \xrightarrow{a(w)} P \parallel !a(w).P$$

$$[\text{Acquire}] \quad req(r)\{P\} \parallel (r, \varphi, \eta)\{\mathbf{0}\} \xrightarrow{\tau} (r, \varphi, \eta)\{P\}$$

$$[\text{Release}] \quad \frac{P \xrightarrow{rel?r} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{rel(r)} (r, \varphi, \eta.rel)\{\mathbf{0}\} \parallel P'}$$

$$[\text{Policy}_1] \quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \models \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\tau} (r, \varphi, \eta.\alpha)\{P'\}} \qquad\qquad [\text{Policy}_2] \quad \frac{P \xrightarrow{\alpha?r} P' \quad \eta.\alpha \not\models \varphi}{(r, \varphi, \eta)\{P\} \xrightarrow{\tau} (r, \varphi, \eta)\{\mathbf{0}\} \parallel [P']_r}$$

$$[\text{Local}_1] \quad \frac{P \xrightarrow{\mu} P'}{(r, \varphi, \eta)\{P\} \xrightarrow{\mu} (r, \varphi, \eta)\{P'\}} \ r \notin \mathsf{n}(\mu) \qquad\qquad [\text{Local}_2] \quad \frac{P \xrightarrow{\mu} P'}{req(r)\{P\} \xrightarrow{\mu} req(r)\{P'\}} \ r \notin \mathsf{n}(\mu)$$

Figure 5.2: Operational Semantics of G-Local π processes.

resource type $res(\theta)$ of a resource r, we assume that $rn(\theta) = \{r\}$. A type T is called *closed* if it does not contain any type variable. A type T is called *open* if it contains type variables. The set of closed types is denoted by $\mathcal{T}_0$.

Remark 5.2.2. In this section, we consider only closed types. Open types are used in the type inference algorithm, which will be presented in the next section.

As expected, the notion of free and bound names applies to types as well.

$$\begin{array}{llll}
\mathsf{fn}(\mathbf{0}) & = \emptyset & \mathsf{fn}(res(\theta)) & = rn(\theta) \\
\mathsf{fn}(!a(t).T) & = \mathsf{fn}(a(t).T) & \mathsf{fn}(\alpha(R).T) & = R \cup \mathsf{fn}(T) \\
\mathsf{fn}(a(t).T) & = \{a\} \cup \mathsf{fn}(t) \cup \mathsf{fn}(T) & \mathsf{fn}(\bar{a}.T) & = \{a\} \cup \mathsf{fn}(T) \\
\mathsf{fn}(\tau.T) & = \mathsf{fn}(T) & \mathsf{fn}(T + T') & = \mathsf{fn}(T) \cup \mathsf{fn}(T') \\
\mathsf{fn}(T \parallel T') & = \mathsf{fn}(T) \cup \mathsf{fn}(T') & \mathsf{fn}((\nu x : t)T) & = \mathsf{fn}(t) \cup \mathsf{fn}(T) \setminus \{x\} \\
\mathsf{fn}((x : t)T) & = \mathsf{fn}(t) \cup \mathsf{fn}(T) \setminus \{x\} & \mathsf{fn}((s : u)T) & = \mathsf{fn}(u) \cup \mathsf{fn}(T) \setminus \{s\}
\end{array}$$

$$\begin{array}{llll}
\mathsf{fn}(\mathbf{0}) & = \emptyset & \mathsf{fn}(res(\theta)) & = \emptyset \\
\mathsf{bn}(!a(t).T) & = \mathsf{bn}(a(t).T) & \mathsf{bn}(\alpha(R).T) & = \mathsf{bn}(T) \\
\mathsf{bn}(a(t).T) & = \mathsf{bn}(T) & \mathsf{bn}(\bar{a}.T) & = \mathsf{bn}(T) \\
\mathsf{bn}(\tau.T) & = \mathsf{bn}(T) & \mathsf{bn}(T + T') & = \mathsf{bn}(T) \cup \mathsf{bn}(T') \\
\mathsf{bn}(T \parallel T') & = \mathsf{bn}(T) \cup \mathsf{bn}(T') & \mathsf{bn}((\nu x : t)T) & = \{x\} \cup \mathsf{bn}(T) \\
\mathsf{bn}((x : t)T) & = \{x\} \cup \mathsf{bn}(T) & \mathsf{bn}((s : u)T) & = \{s\} \cup \mathsf{bn}(T)
\end{array}$$

Note that t in a process type $a(t).T$ contributes (with its free names) to the set of free names of that type, and the same holds for a channel type $(x : t)T$ where t provides a similar contribution.

Notation 5.2.3. *For the sake of simplicity, we write $y\#T$ for $y \notin \mathsf{fn}(T)$. This notation extends to a set R of resources, i.e. $R\#T$.*

Behavioural types T look like CCS terms and describe how a process accesses a resource and communicates over channels. As usual, x in a channel type $(x : t)T$ acts as a binder for x with the scope T. We assume that $x \in \mathsf{fn}(T)$ and T does not contain any action prefix $\alpha(R)$. A channel, capable of receiving a resource names, has the special channel type $(s : u)T$, where u is a resource type.

The type $\mu.T$ describes a process that performs the action μ and then behaves as prescribed by T. In particular, $a(t).T$ describes a process in which the channel a can carry names of channel type t. Similarly, $\alpha(R).T$ describes a process that can perform an access action α on resources that belong to the finite set R. Intuitively, R represents the set of resource names, on which the action α may be performed at run-time.

We use $T\{b/x\}$ ($T\{r/s\}$ resp.) to denote substitution of x (s, resp.) with b (r, resp.). The notion of substitution also apply to $\langle r, \varphi\rangle$ and Φ as expected. For the sake of simplicity, we write $y\#T$ for $y \in \mathsf{fn}(T)$ that extends as $R\#T$ for a finite set R of resource names.

$$\begin{array}{ll}
T \equiv T' \text{ if } T' \text{ is } \alpha\text{-equivalent of } T & \\
T \parallel \mathbf{0} \equiv \mathbf{0} \parallel T = T & T + \mathbf{0} \equiv \mathbf{0} + T \equiv T \\
T_1 + T_2 \equiv T_2 + T_1 & T_1 \parallel T_2 \equiv T_2 \parallel T_1 \\
(T_1 + T_2) + T_3 \equiv T_1 + (T_2 + T_3) & (T_1 \parallel T_2) \parallel T_3 \equiv T_1 \parallel (T_2 \parallel T_3)
\end{array}$$

Figure 5.3: Structural Congruence on Types

$$\text{[bt\_act]} \quad \pi.T \xrightarrow{\pi} T \qquad\qquad \text{[bt\_cong]} \quad \frac{T_1 \equiv T_1' \;\; T_1' \xrightarrow{\mu} T_2' \;\; T_2' \equiv T_2}{T_1 \xrightarrow{\mu} T_2}$$

$$\text{[bt\_par]} \quad \frac{T_1 \xrightarrow{\mu} T_1'}{T_1 \parallel T_2 \xrightarrow{\mu} T_1' \parallel T_2} \qquad\qquad \text{[bt\_choice]} \quad \frac{T_1 \xrightarrow{\mu} T_1'}{T_1 + T_2 \xrightarrow{\mu} T_1'}$$

$$\text{[bt\_res]} \quad \frac{T \xrightarrow{\mu} T' \qquad \mu \neq x(t), \bar{x}}{(\nu x)T \xrightarrow{\mu} (\nu x)T'} \qquad \text{[bt\_comm]} \quad \frac{T_1 \xrightarrow{\bar{a}} T_1' \;\; T_2 \xrightarrow{a} T_2'}{T_1 \parallel \; T_2 \xrightarrow{\langle a \rangle} T_1' \parallel T_2'}$$

$$\text{[bt\_rep]} \quad !a.T \xrightarrow{a} !a.T \parallel T$$

Figure 5.4: Operational Semantics of Types.

5.2.2 Operational Semantics

Since types are equipped with behavioural information, one has to precisely define their meaning. The semantics of types, reported in Tab. 5.4, is defined through a labelled transition system, similar to the one defined for processes in Section 2.4.4. Labels μ, μ' for transitions are τ for silent actions, $a, \bar{a}$ for *abstract* input/output, $\alpha(R)$ for resource-access actions and $\langle a \rangle$ for communications. Formally, the labelled transition system is based on the structural congruence on types that includes those defined in Section 3 of Chapter 2 and is defined in Fig. 5.3.

Now we are ready to comment on semantic rules. The rule [bt_act] describes actions of types. A type $\mu.T$ performs an action μ, then behaves like P. In the rule [bt_cong], congruent types can perform the same action. The rule [bt_par] expresses the parallel behaviour of types, while the rule [c_choice] represents a choice among alternatives. The rule [bt_comm] is used to synchronise a free name. The rule [bt_res] manages restrictions. The rule [bt_res] ensures that an action μ of T is also an action of $(\nu x)T$, if the action is not restricted by x, i.e $\mu \neq x(t)$ and $\mu \neq \overline{x}$, otherwise μ becomes τ of $(\nu x)T$ as in the rule [bt_res$_2$]. Finally, the rule [bt_rep] instantiates an instance of $!a(t).T$ by performing $a(t)$. The result is the parallel composition of T and $!a(t).T$.

5.3 Typing systems

We now introduce the type and effect system. As usual, a context Γ is a map from a channel name/resource name to channel/resource types. We write $\Gamma \vdash a : t$ if $a \in dom(\Gamma)$ and $\Gamma(a) = t$. Judgements of the type system have form $\Gamma \vdash P : T, E$, where Γ is a context, $P \in \mathcal{P}$, $T \in \mathcal{T}$ and E is a set of resource constraints. The intuitive reading of $\Gamma \vdash P : T, E$ is that T is the abstract behaviour of P and E describes the side effects occurring when P executes.

Definition 5.3.1. Let Γ be a context. We say that Γ is well-formed if whenever $\Gamma \vdash a : (x : t)T$ then $\mathsf{fn}(t, T) \subseteq \{x\} \cup dom(\Gamma)$.

Convention 5.3.2. From now on, we only consider well-formed contexts, unless stated otherwise.

A resource constraint ϱ has the form $\varphi\langle T\rangle$ or $\langle T\rangle\varphi$, where $T \in \mathcal{T}$ and $\varphi \in \Phi$. We use ξ to denote the empty resource constraint. Intuitively, T in $\varphi\langle T\rangle$ or $\langle T\rangle\varphi$ has to be understood as a fragment of usages on a resource associated with the policy φ. Resource constraints are thus considered as forms of side effects, which record a *fragment* of the resource behaviour of processes with respect to a usage policy. The form $\varphi\langle T\rangle$ means that T is the initial fragment of usage, while $\langle T\rangle\varphi$ describes a fragment of usage that is performed by a process. We use the symbol $\vee$ to union the resource constraints. By abuse of notation, we use $\equiv$ to denote $\varphi\langle T\rangle \equiv \varphi\langle T'\rangle$ whenever $T \equiv T'$. Furthermore, given sets E and E' of resource constraints, we write $E \equiv E'$ to denote congruent resource constraints, namely, if there is a bijective map f from E to E' such that for each $\varrho \in E$, $\varrho \equiv f(\varrho)$.

Before going into details of the judgement rules, we introduce some auxiliary definitions. To handle the infinite behaviour of the guarded replication, we introduces a new operator $!_\#$ on side effects. Intuitively, all the resource constraints in E are infinitely repeated as the result of replication. Formally, it is defined as follows

$$!_\#(E \vee \varphi\langle T\rangle) \;\; = !_\# E \vee \varphi\langle !T\rangle \quad !_\#(E \vee \langle T\rangle\varphi) \;\; = !_\# E \vee \langle !T\rangle\varphi \quad !_\#\xi \;\; = \xi$$

Remark 5.3.3. The operator $!_\#$ on effects yields to a result in an infinite number of parallel resource usages. Alternatively, the operator could have been defined *sequentially*, i.e. an infinite concatenation of resource usages. However, this could lead to *undecidable* type inference algorithm as we will see in the next section.

We introduce the hiding operator on closed types. The type $T_{\downarrow R}$ describes a process that behaves like T, except that actions that are *not* related to R are replaced by invisible actions τ. Intuitively, $T_{\downarrow R}$ represents the abstract resource behaviour on

R extracted from T. Formally, it is defined as follows:

$$\begin{array}{llll}
\mathbf{0}_{\downarrow R} & = \mathbf{0} & (\tau.T)_{\downarrow R} & = \tau.T_{\downarrow R} \\
(\alpha(R').T)_{\downarrow R} & = \begin{cases} \alpha(R \cap R').T_{\downarrow R} & \text{if } R \cap R' \neq \emptyset \\ \tau.T_{\downarrow R} & \text{if } R \cap R' = \emptyset \end{cases} & ((\nu x).T)_{\downarrow R} & = T_{\downarrow R} \\
(a.T)_{\downarrow R} & = \tau.T_{\downarrow R} & (\bar{a}.T)_{\downarrow R} & = \tau.T_{\downarrow R} \\
(T_1 + T_2)_{\downarrow R} & = T_{1\downarrow R} + T_{2\downarrow R} & (T_1 \parallel T_2)_{\downarrow R} & = T_{1\downarrow R} \parallel T_{2\downarrow R} \\
(!a.T)_{\downarrow R} & =!(a.T)_{\downarrow R} & &
\end{array}$$

Similarly, the type $T_{\uparrow R}$ describes a process that behaves like T, except that actions over resources included in R are replaced by invisible actions τ. Intuitively, $T_{\uparrow R}$ represents the abstract behaviour extracted from T by removing the resource behaviour on R.

$$\begin{array}{llll}
\mathbf{0}_{\uparrow R} & = \mathbf{0} & (\tau.T)_{\uparrow R} & = \tau.T_{\uparrow R} \\
(\alpha(R').T)_{\uparrow R} & = \begin{cases} \alpha(R \setminus R').T_{\uparrow R} & \text{if } R \setminus R' \neq \emptyset \\ \tau.T_{\uparrow R} & \text{if } R \setminus R' = \emptyset \end{cases} & ((\nu x).T)_{\uparrow R} & = (\nu x).T_{\uparrow R} \\
(a.T)_{\uparrow R} & = a.T_{\uparrow R} & (\bar{a}.T)_{\uparrow R} & = \bar{a}.T_{\uparrow R} \\
(T_1 + T_2)_{\uparrow R} & = T_{1\uparrow R} + T_{2\uparrow R} & (T_1 \parallel T_2)_{\uparrow R} & = T_{1\uparrow R} \parallel T_{2\uparrow R} \\
(!a.T)_{\uparrow R} & =!(a.T)_{\uparrow R} & &
\end{array}$$

These "hiding" operators selectively mask part of the actions and resemble the constructors introduced in [4]. However, in our proposal these operators are specialised to distinguish between access actions and all the other actions.

Convention 5.3.4. We often write $T_{\uparrow r}$ ($T_{\downarrow r}$) for $T_{\uparrow\{r\}}$ ($T_{\uparrow\{r\}}$, resp.) and $T_{\uparrow\_}$ ($T_{\downarrow\_}$) for $T_{\uparrow\mathcal{R}}$ ($T_{\uparrow\mathcal{R}}$, resp.).

The rules of the type system are given in Tab. 5.5. The rules [ts_par], [ts_choice], [ts_res], [ts_act], [ts_tau] and [ts_rep] are similar to those in Section 2.4.4. In the rules [ts_par] and [ts_choice], the resulting type is the composition of types of the components, while in the rules [ts_res], [ts_act] and [ts_tau], the resulting type extends the types of the premise to reflect the structure of processes. Also note that in the rules [ts_par] and [ts_choice] the resource constraints of the components are embedded in the resulting type, while in [ts_res], [ts_act] and [ts_tau], the resource constraints remain the same. In the rule [ts_rep], all the resource constraints of the conclusion are obtained as result of the operator $!_{\#}$.

In the rule [ts_act], a resource variable is replaced by the set of its possible names, i.e. $rn(\Theta)$. In the rule [ts_input_res], the type of P in $a(s).P$ is split into T_1 and T_2 with the condition that $s\#T_1$. This condition guarantees that all information about s is included in T_2. Furthermore, the abstraction of communication-based behaviour encoded in T_2 (obtained by excluding resource behaviour $T_{2\uparrow\_}$), must exactly match T_a, that describes the usage of argument s in the process P when s is received on channel a. Notice that the split $T_1 \parallel T_2$ also splits resource behaviour into two parts.

[ts_empty] $\Gamma \vdash \mathbf{0} : \mathbf{0}, \xi$

[ts_output]
$$\frac{\Gamma \vdash P : T_1, E \quad \Gamma \vdash a : (y : t)T_2 \quad \Gamma \vdash b : t}{\Gamma \vdash \bar{a}b.P : \bar{a}.(T_1 \parallel T_2\{b/y\}), E}$$

[ts_output_res]
$$\frac{\Gamma \vdash P : T_1, E \quad \Gamma \vdash a : (s : res(\Theta))T_2 \quad \Gamma \vdash r : res(\Theta_r) \quad \Theta_r \subseteq \Theta}{\Gamma \vdash \bar{a}r.P : \bar{a}.(T_1 \parallel T_2\{r/s\}), E}$$

[ts_input]
$$\frac{\Gamma, y : t \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (y : t)T_a \quad T_{2\uparrow\_} = T_a \; y\#T_1}{\Gamma \vdash a(y).P : a.(T_1 \parallel T_{2\downarrow\_}), E}$$

[ts_input_res]
$$\frac{\Gamma, s : t \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (s : u)T_2 \quad T_{2\uparrow\_} = T_a \; s\#T_1}{\Gamma \vdash a(s).P : a.(T_1 \parallel T_{2\downarrow\_}), E}$$

[ts_choice]
$$\frac{\Gamma \vdash P_1 : T_1, E_1 \qquad \Gamma \vdash P_2 : T_2, E_2}{\Gamma \vdash P_1 + P_2 : T_1 + T_2, E_1 \vee E_2}$$

[ts_par]
$$\frac{\Gamma \vdash P_1 : T_1, E_1 \; \Gamma \vdash P_2 : T_2, E_2}{\Gamma \vdash P_1 \parallel P_2 : T_1 \parallel T_2, E_1 \vee E_2}$$

[ts_rep]
$$\frac{\Gamma \vdash a(w).P : T, E}{\Gamma \vdash !a(w).P : !T, !_{\#}E}$$

[ts_res]
$$\frac{\Gamma : x : t \vdash P : T, E}{\Gamma \vdash (\nu x)P : (\nu x)T, E}$$

[ts_act]
$$\frac{\Gamma \vdash P : T, E \quad \Gamma \vdash s : res(\Theta) \quad R = rn(\Theta)}{\Gamma \vdash \alpha(s).P : \alpha(R).T, E}$$

[ts_tau]
$$\frac{\Gamma \vdash P : T, E}{\Gamma \vdash \tau.P : \tau.T, E}$$

[ts_resjoin]
$$\frac{\Gamma \vdash P : T, E \quad \Gamma \vdash r : res(\Theta) \quad \langle r, \varphi \rangle \in \Theta}{\Gamma \vdash (r, \varphi, \eta)\{P\} : T_{\uparrow\{r\}}, E \vee \varphi\langle \eta.T_{\downarrow\{r\}} \rangle}$$

[ts_resreq]
$$\frac{\Gamma \vdash P : T, E \quad \Gamma \vdash s : res(\Theta) \quad R = rn(\Theta)}{\Gamma \vdash req(s)\{P\} : T_{\uparrow R}, E \vee \bigvee_{\langle r, \varphi \rangle \in \Theta} \langle \eta.T_{\downarrow\{r\}} \rangle \varphi}$$

[ts_eq]
$$\frac{\Gamma \vdash P : T, E \quad T \equiv T' \quad E \equiv E'}{\Gamma \vdash P : T', E'}$$

Figure 5.5: Typing rules.

Out treatment of resource requires that resource information must be maintained in both sides (senders and receivers). That is, the rule requires to keep the abstraction of resource behaviour of T_2, i.e. $T_{2\downarrow\_}$, in parallel with T_1. For example, consider a process

$$(r, \varphi, \epsilon)\{\alpha(r).\mathbf{0} \parallel a(y).\beta(r).y.\mathbf{0}\} \parallel \bar{a}\langle b\rangle.\mathbf{0}$$

Here, y causally depends on $\beta(r)$, hence the type of a must depend on $\beta(r)$, i.e. $\beta(r).y.\mathbf{0}$. In result, $\beta(r).y.\mathbf{0}$ is transferred to sender when typing the input subprocess $a(y).\beta(r).y.\mathbf{0}$, therefore we end up loosing $\beta(r)$ when typing $(r, \varphi, \epsilon)\{\alpha(r).\mathbf{0} \parallel a(y).\beta(r).y.\mathbf{0}\}$, where $\alpha(r).\mathbf{0} \parallel a(y).\beta(r).y.\mathbf{0}$ has the type $\alpha(r).\mathbf{0} \parallel \mathbf{0}$. Consequently, incorrect resource usages are obtained.

In the rule [ts_output_res], the type $\bar{a}r.P$ is the parallel composition of the type of P and the continuation $T_a\{r/s\}$ with the actual argument r, provided that a has the type $(s : res(\Theta))T_a$. In addition, the rule [ts_output_res] requires that Θ_r in the resource type of r is *included* in the set Θ, declared by the channel a.

In the rules [ts_resjoin] and [ts_resreq], the resource-based abstractions are extracted from the behavioural type of P. A set of newly generated resource constraints are added to the side effect. More precisely, in the rule [ts_resjoin], the resource behaviour of P on r, i.e $T_{\downarrow r}$, and the policy φ declared by r form a resource constraint on usages of r. Similarly, in the rule [ts_resreq], a resource constraint is generated for each pair $\langle r, \varphi\rangle$ declared in s.

Finally, the rule [ts_eq] is related to sub-typing. The structural congruence used in this rule, instead of preorders, is a key point of the type system to maintain spatial structure of processes in types.

We are now ready to define the satisfaction of a policy for a given type. A type T satisfies a policy φ if the set of its traces is included in that policy. Recall that policies are expressed as LTL formulas and that we use $Traces(T)$ to denote a set of traces, which are generated by T. Formally, we have:

Definition 5.3.5. Given a $T \in \mathcal{T}_0$ and a policy φ, we say that T *satisfies* φ, denoted by $T \models \varphi$, if $Traces(T) \subseteq \varphi$.

Resource policies can be either *local* or *global*, which leads to two ways of interpreting a given set E of resource constraints $\langle T_i\rangle\varphi$ (or $\varphi\langle T_i\rangle$) of a given policy φ. In the global case, to check a given set of resource constraints $\langle T_i\rangle\varphi$ (or $\varphi\langle T_i\rangle$) one needs to simultaneously check all of them together. This amounts to checking all T_i in parallel. In the local case, more simply, we need to individually check each T_i against the policy.

We are now ready to define the notion of *well-typedness*.

Definition 5.3.6 (Local Satisfaction)**.** Given a finite set E of resource constraints $\langle T_i\rangle\varphi$ (or $\varphi\langle T_i\rangle$) of a given policy φ, we say that E is locally satisfied if for each $\langle T_i\rangle\varphi$ (or $\varphi\langle T_i\rangle$), $T_i \models \varphi$

Definition 5.3.7 (Global Satisfaction)**.** Given a finite set E of resource constraints $\langle T_i \rangle \varphi$ (or $\varphi \langle T_i \rangle$) of a given policy φ, where $i \in I$ (I is the indexing set), we say that E is globally satisfied if $\prod_I T_i \models \varphi$, where $\prod$ denotes the parallel operator of T_i. We say that E is satisfied, if it is either locally or globally satisfied.

Definition 5.3.8. Given a well-formed context Γ, a process P is *locally (globally, resp.) well-typed* under Γ if $\Gamma \vdash P : T, E$, $T \in \mathcal{T}_0$, and all the resource constraints in E are locally (globally, resp.) satisfied. We say that P is *well-typed* under Γ if it is either locally or globally well-typed under Γ.

Remark 5.3.9. Notice that the notion of well-typedness is defined under Γ. This means that information about channel types is given. Our type system assigns types to processes under a fixed type environment. Consider the following process:

$$P = a(y).(\bar{y}\langle y \rangle \parallel \bar{a}\langle a \rangle \parallel \bar{b}\langle b \rangle)$$

If we do not fix Γ, according to the rule [ts_input], P can either be typed by letting T_1 be the type of $\bar{y}\langle y \rangle$ or the type $\bar{y}\langle y \rangle \parallel \bar{a}\langle a \rangle$. Hence, without type information, the type system does not guarantee the uniqueness typing.

Remark 5.3.10. The notion of well-typedness guarantees that a well-typed process correctly uses resources. However, this does not meant that a well-typed process can not *fail* (as a result of performing an open action). Consider the process $\alpha(r).\mathbf{0}$. It is well-typed under $\Gamma \triangleq r : \langle r, \varphi \rangle$ as follows:

$$\Gamma \vdash \alpha(r).\mathbf{0} : \alpha(r).\mathbf{0}, \xi$$

However, it can perform an open action $\alpha?r$.

5.3.1 Examples

Storage Service. We consider a storage service with a finite set A of update actions, divided into two groups A_α and A_β, such that each $\alpha_i \in A_\alpha$ and its counterpart $\beta_i \in A_\beta$ make a pair of action/co-action. The policy of the storage requires that once an action is performed, its co-action is forbidden. The global policy is formally defined by the following LTL formula:

$$\varphi_{ss} := \bigwedge_{i \in I} \mathbf{G}((\alpha_i \rightarrow \mathbf{G}\neg\beta_i) \wedge (\beta_i \rightarrow \mathbf{G}\neg\alpha_i)),$$

where I is an index set of A_α and A_β.

Let us consider a system consisting of a storage service and three client applications. Assume that the storage system is specified as follows:

$$\begin{array}{ll}
\mathit{App}_1 & = \mathit{req}(ss)\{\alpha_1(ss) + \alpha_2(ss)\} \\
\mathit{App}_2 & = \mathit{req}(ss)\{\beta_3(ss) + \alpha_3(ss)\} \\
\mathit{App}_3 & = \mathit{req}(ss)\{\beta_2(ss) + \alpha_1(ss)\} \\
\mathit{Storage} & = (ss, \varphi_{ss}, \epsilon)\{\mathbf{0}\} \\
\mathit{System} & = \mathit{Storage} \parallel \mathit{App}_1 \parallel \mathit{App}_2 \parallel \mathit{App}_3
\end{array}$$

We assume that the type environment Γ contains only $ss : \langle ss, \varphi_{ss}\rangle$. The types of the systems are as follows:

$$\begin{array}{l}
\Gamma \vdash \mathit{App}_1 : \tau + \tau, E_1 \\
\Gamma \vdash \mathit{App}_2 : \tau + \tau, E_2 \\
\Gamma \vdash \mathit{App}_3 : \tau + \tau, E_3 \\
\Gamma \vdash \mathit{Storage} : \mathbf{0}, E_4 \\
\Gamma \vdash \mathit{System} : \tau + \tau \parallel \tau + \tau \parallel \tau + \tau, E,
\end{array}$$

where

$$\begin{array}{ll}
E_1 & = \langle \alpha_1 + \alpha_2 \rangle \varphi_{ss} \\
E_2 & = \langle \beta_3 + \alpha_3 \rangle \varphi_{ss} \\
E_3 & = \langle \beta_2 + \alpha_1 \rangle \varphi_{ss} \\
E_4 & = \varphi_{ss}\langle \mathbf{0} \rangle \\
E & = E_1 \wedge E_2 \wedge E_3 \wedge E_4
\end{array}$$

It is easy to see that the client applications App_1 and App_3 have a conflict as their update actions α_2 and β_2 are co-actions.

Mobile Reader. Consider again the example used in Chapter 2, i.e. consider reading e-books from an online store on tablet devices. A user, when reading an e-book, may write some annotations. The way of using the online store depends on which kind of connections, low-bandwidth or high-bandwidth, a tablet device has. In the former case, the tablet needs to load an e-book from the store to its local memory before any other actions and if a user makes annotations on the e-book, it requires to *store* them back on the online store. In the latter case, the user directly reads/writes e-books, however it is required that the user eventually releases the connection due to the high cost of the connection itself. We use rd, wr, ld and st to model operations of reading e-books, writing annotations, loading e-books from the online store to local memory and storing them back to the online store, respectively.

We use r_h and r_l to denote high-bandwidth and low-bandwidth connection resources, respectively. Similarly φ_l and φ_h denote policies for low-bandwidth and high-bandwidth connections, respectively. Formally, φ_l and φ_h are defined by the following LTL formulas:

i) $\varphi_l = (ld)\mathit{true} \wedge (\mathbf{G}(wr \rightarrow \mathbf{F}st))$, that requires that load is the first action and every write will *eventually* lead to a store operation;

ii) $\varphi_h = \mathbf{F}\ \mathit{rel}$, that requires that *eventually* in the future rel holds.

The specification of the two tablet devices and of the two connections is the following:

$$\begin{array}{ll}
connection_l & ::= (r_l, \varphi_l, \epsilon)\{\mathbf{0}\} \parallel !\bar{x}\langle r_l\rangle \\
tablet_1 & ::= !x(s).req(s)\{ld(s).(rd(s).rel(s) + wr(s).st(s).rel(s))\} \\
connection_h & ::= (r_h, \varphi_h, \epsilon)\{\mathbf{0}\} \parallel !\bar{x}\langle r_h\rangle \\
tablet_2 & ::= !x(s).req(s)\{read(s) + write(s) + rel(s)\} \\
System_1 & ::= connection_l \parallel connection_h \parallel tablet_1 \\
System_2 & ::= connection_l \parallel connection_h \parallel tablet_2
\end{array}$$

The first device $tablet_1$ always loads e-books to the local memory, and stores annotations after writing them. The second device $tablet_2$ works fine with high-bandwidth connection, since it guarantees read/write operations without loading e-books on local memory.

Now we consider the types of $System_1$ and $System_2$ in details. First, we assume a type environment Γ such that s and x have the following types:

$$\begin{array}{l}
\Gamma \vdash x : (s : res(\{r_{lb}, r_{hb}\}, \{\varphi_{lh}, \varphi_{lh}\}))\mathbf{0} \\
\Gamma \vdash s : res(\{r_{lb}, r_{hb}\}, \{\varphi_{lh}, \varphi_{lh}\})
\end{array}$$

The result of the typing process is as follows:

$$\begin{array}{l}
\Gamma \vdash x : (s : res(\{r_{lb}, r_{hb}\}, \{\varphi_{lh}, \varphi_{lh}\}))\mathbf{0} \\
\Gamma \vdash s : res(\{r_{lb}, r_{hb}\}, \{\varphi_{lh}, \varphi_{lh}\}) \\
\Gamma \vdash tablet_1 : !x.\tau(\tau.\tau + \tau.\tau.\tau), !E_1 \\
\Gamma \vdash tablet_2 : !x.\tau.\tau.\tau, !E_2 \\
\Gamma \vdash connection_{hb} : !\bar{x}, E_3 \\
\Gamma \vdash connection_{hb} : !\bar{x}, E_4 \\
\Gamma \vdash System_1 : !x \parallel !x.(\tau(\tau\tau + \tau\tau\tau)) \parallel !\bar{x} \parallel !\bar{x}, E_1 \wedge E_3 \wedge E_4 \\
\Gamma \vdash System_2 : !x \parallel !x.(\tau + \tau + \tau) \parallel !\bar{x} \parallel !\bar{x}, E_1 \wedge E_3 \wedge E_4
\end{array}$$

where

$$\begin{array}{ll}
E_1 & = \langle T_1\rangle\varphi_{hb} \wedge \langle T_1\rangle\varphi_{lb} \\
E_3 & = \varphi_{hb}\langle\mathbf{0}\rangle \\
E_4 & = \varphi_{lb}\langle\mathbf{0}\rangle \\
E & = E_1 \wedge E_3 \wedge E_4 \\
T_1 & = load.(rd.rel + wr.st.rel) \\
T_2 & = rd + wr + rel
\end{array}$$

A fragment of the typing is the following:

$$\cfrac{\cfrac{\cfrac{\vdots}{\vdash ld(s).(rd(s).rel(s) + wr(s).st(s).rel(s)) : l(R).(rd(R).rel(R) + wr(R).st(R).rel(R)), true}}{\vdash req(s)\{ld(s).(rd(s).rel(s) + wr(s).st(s).rel(s))\} : \tau(\tau.\tau + \tau.\tau.\tau), C_1}}{\vdash x(s).req(s)\{ld(s).(rd(s).rel(s) + wr(s).sr(s).rel(s))\} : x.\tau(\tau.\tau + \tau.\tau.\tau), C_1}$$

$$\cfrac{\vdots}{\Gamma \vdash System_1 : !x.\tau(\tau.\tau + \tau.\tau.\tau) \parallel !x.\tau.\tau.\tau \parallel !\bar{x} \parallel !\bar{x}, C_1 \wedge C_3 \wedge C_4}$$

To prove the validity of connection usages of the tablet devices, we need to check whether the types $T_1 = ld.(rd.rel + wr.st.rel)$ of $tablet_1$ and $T_2 = rd + wr + rel$ of $tablet_2$ satisfy φ_l and φ_h or not. The result of type checking shows that T_1 represents a valid usage of both connections, i.e. the first device uses them correctly, unlike the second device. We find indeed an infinite trace with only read/write operations that violates φ_h, as it never releases the connection on that trace. This source of infinity can be effectively handled by our treatment of replication. More precisely, the replication of a resource usage can be seen as an infinite number of parallel usages of the resource. Note that this is also a source of imprecision because of interleaving behaviours.

5.3.2 Properties of the Type System

The main goal of this section is to prove the *subject reduction* property of the type system. To prove it and better understand the type system, we first study some basic properties of the type system. The first property we consider is the *normal* form of type derivations, which can be thought as *syntax-directed* type derivation. Next, we prove the type inversion theorem, which shows the structural correspondence between processes and types. Then, we study types of congruent processes (the subject congruence theorem). Basically, congruent processes have the same type. Finally, we introduce the notion of simulation on types, denoted by $\preceq$. Intuitively, the relation $\preceq$ models the *subtyping* relation on types, i.e. if $T_1 \preceq T_2$, then T_1 is able to *simulate* what T_2 can do.

Definition 5.3.11. A simulation $\preceq$ is the largest binary relation on closed process types such that whenever $T_1 \preceq T_2$

- if $T_1 \xrightarrow{l} T_1'$, where $l \in \{a, \bar{a}, \langle a \rangle\}$ then there exists T_2' s.t. $T_2 \xrightarrow{l} T_2'$ and $T_1' \preceq T_2'$.
- if $T_1 \xrightarrow{\tau} T_1'$ then there exists T_2' s.t. $T_2 \xrightarrow{\tau}^* T_2'$ and $T_1' \preceq T_2'$.
- if $T_1 \xrightarrow{\alpha(S)} T_1'$ then there exist T_2' and S' s.t. $T_2 \xrightarrow{\alpha(S')} T_2'$ such that $S \subseteq S'$ and $T_1' \preceq T_2'$.

We write $T_1 \sim T_2$ for $T_1 \preceq T_2$ and $T_2 \preceq T_1$.

Note that our definition differs from the standard [82] in the third clause, where a set of possible resource names of α in T_2 is at least as in T_1.

Normal Derivation.

Definition 5.3.12. A type derivation $\Gamma \vdash P : T, E$ is called *normal*, denoted by $\Gamma \vdash_N P : T, E$, if the rule [ts_eq] is applied immediately above the rules [ts_input] and [ts_input_res].

In the following lemma, some basic properties of types are proved. More precisely, In the clause (1) congruent types have the same set of traces, while in the clause (2) congruent types satisfy the same usage policies. The property (3) says that if a name is not included in the set of free names of a given process, then it is also not included in the type of that process. The next two properties are essential for proving the subject reduction theorem (see below). The property (4) states that excluding a resource from a given type does not get any new behaviour. The property (5) states that the operator of splitting a given type into two parts, namely resource-based and communication-based, preserves all behaviour of that type. The last property (6) shows that congruence of types is preserved under the "hiding" operators.

Lemma 5.3.13 (Basic Properties)**.**

(1) If given $T_1, T_2 \in \mathcal{T}_0$, $T_1 \equiv T_2$ then $Traces(T_1) = Traces(T_2)$.

(2) If given $T_1, T_2 \in \mathcal{T}_0$, $T_1 \models \varphi$ and $T_1 \equiv T_2$ then $T_2 \models \varphi$.

(3) If $\Gamma \vdash P : T, E$ and $a\#P$ then $a\#T$.

(4) $T_{\uparrow r} \preceq T$ for any $T \in \mathcal{T}_0$.

(5) $T \preceq T_{\uparrow\text{-}} \parallel T_{\downarrow\text{-}}$ for any $T \in \mathcal{T}_0$.

(6) Given $T, T' \in \mathcal{T}_0$. If $T \equiv T'$, then $T_{\downarrow R} \equiv T'_{\downarrow R}$ and $T_{\uparrow R} \equiv T'_{\uparrow R}$.

Proof. (1) It is straightforward by induction on the structure of T_1.

(2) It is immediate from (1).

(3) The proof proceeds by standard induction on the derivation of $\Gamma \vdash P : T, E$.

(4) We need to prove that $\mathbb{R} = \{(T_{\uparrow r}, T) | T \in \mathcal{T}_0\} \cup \{(T, T) | T \in \mathcal{T}_0\}$ is a simulation relation. The proof proceeds by induction on the structure of T. Most of the cases are obvious. We consider the most interesting cases.

   – The case of $T = \alpha(R).T'$: we have that $T_{\uparrow r} \overset{\alpha(R\setminus\{r\})}{\rightarrow} T'_{\uparrow r}$. We need to show that there exist T'' and R' such that $T \overset{\alpha(R)}{\rightarrow} T''$ and $(T'_{\uparrow r}, T'') \in \mathbb{R}$. Let $T'' \overset{def}{=} T'$ and $R' \overset{def}{=} R$. Since $T \overset{\alpha(R)}{\rightarrow} T'$,$R \setminus \{r\} \subseteq R$ and $(T'_{\uparrow r}, T') \in \mathbb{R}$, the required result follows.

(5) It is sufficient to show that $\mathbb{R} = \{(T, T_{\downarrow\text{-}} \parallel T_{\uparrow\text{-}}) | T \in \mathcal{T}_0\} \cup \{(T, T) | T \in \mathcal{T}_0\}$ is a simulation relation. The proof is standard.

(6) It is straightforward by induction on the structure of T.

□

The normal derivation theorem given below states that, from any type derivation, it is possible to obtain a "syntax-directed" type derivation of the same conclusion. It is useful for studying the corresponding structure between a process and its process type.

Theorem 5.3.14 (Normal Derivation)**.** *If $\Gamma \vdash P : T, E$, then $\Gamma \vdash_N P : S, E'$ for some $S \equiv T$ and $E \equiv E'$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash P : T, E$ by distinguishing the last typing rule applied. Many cases are obvious. We consider the most interesting cases.

- the case of the rule [ts_input]: Consider the judgement

  $$\Gamma \vdash a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E$$

  which is deduced from

  $$\Gamma, y : t \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (y : t)T_a \quad T_{2\uparrow\_} = T_a \quad y\#T_1.$$

  By induction hypothesis, we have $\Gamma, y : t \vdash_N P : S, E'$, where $S \equiv T_1 \parallel T_2$ and $E \equiv E'$. By applying the rule [ts_eq] to $\Gamma, y : t \vdash_N P : S, E'$, we have the required result:

  $$\Gamma \vdash_N a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E'.$$

- the case of the rule [ts_input_res]: Consider the judgement

  $$\Gamma \vdash a(s)P : a.(T_1 \parallel T_{2\downarrow\_}), E$$

  which is deduced from

  $$\Gamma, s : u \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (s : u)T_a \quad T_{2\uparrow\_} = T_a \quad s\#T_1.$$

  By induction hypothesis, we have $\Gamma, s : u \vdash_N P : S, E'$, where $S \equiv T_1 \parallel T_2$ and $E \equiv E'$. By applying the rule [ts_eq] to $\Gamma, s : u \vdash_N P : S, E'$, we have the required result:

  $$\Gamma, s : u \vdash_N a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E'.$$

- the case of the rule [ts_res_join]: Consider the judgement

  $$\Gamma \vdash (r, \varphi, \eta)\{P\} : T_{\uparrow r}, E \vee \varphi\langle \eta.T_{\downarrow\_}\rangle$$

  which is deduced from

  $$\Gamma \vdash P : T, E \quad \Gamma \vdash r : res(\Phi) \quad \langle r, \varphi\rangle \in \Phi.$$

  By induction hypothesis, we have $\Gamma \vdash_N P : T', E'$, where $T' \equiv T$ and, hence by the structural rules it implies that $T_{\uparrow r} \equiv T'_{\uparrow r}$ and $E \vee \varphi\langle \eta.T_{\downarrow r}\rangle \equiv E' \vee \varphi\langle \eta.T'_{\downarrow r}\rangle$. We can thus obtain the required result

  $$\Gamma \vdash_N (r, \varphi, \eta)\{P\} : T'_{\uparrow r}, E \vee \varphi\langle \eta.T'_{\downarrow r}\rangle$$

□

Example 5.3.15. Let us consider the following process (for the sake of simplicity, we omit the trailing $\mathbf{0}$) :

$$P = a(y).(\bar{y} \parallel \bar{c} \parallel \bar{b})$$

under the type environment $\Gamma = a : (y : ()\mathbf{0})(\bar{y} \parallel \bar{b}), b : ()\mathbf{0}, c : ()\mathbf{0}$. The process P has a non-normal type derivation as described below:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash \bar{y} : \bar{y}, \xi} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash \bar{c} \parallel \bar{b} : \bar{c} \parallel \bar{b}, \xi}}{\Gamma \vdash \bar{c} \parallel \bar{b} : \bar{b} \parallel \bar{c}, \xi}\,[\text{ts\_eq}]}{\Gamma \vdash \bar{y} \parallel \bar{c} \parallel \bar{b} : \bar{y} \parallel \bar{b} \parallel \bar{c}, \xi}\,[\text{ts\_par}]}{\Gamma \vdash P : a.\bar{c}, \xi}\,[\text{ts\_input}]$$

By swapping the rules [ts_eq] and [ts_par] in the above type derivation, we can obtain the normal type derivation for P:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash_N \bar{y} : \bar{y}, \xi} \quad \dfrac{\vdots}{\Gamma \vdash_N \bar{c} \parallel \bar{b} : \bar{c} \parallel \bar{b}, \xi}}{\Gamma \vdash_N \bar{y} \parallel \bar{c} \parallel \bar{b} : \bar{y} \parallel \bar{c} \parallel \bar{b}, \xi}\,[\text{ts\_par}]}{\Gamma \vdash_N \bar{y} \parallel \bar{c} \parallel \bar{b} : \bar{y} \parallel \bar{b} \parallel \bar{c}, \xi}\,[\text{ts\_eq}]}{\Gamma \vdash_N P : a.\bar{c}, \xi}\,[\text{ts\_input}]$$

Convention 5.3.16. From now on, we consider $\Gamma \vdash P : T, E$ up to equivalence of the resource constraint E, unless stated otherwise.

Type Inversion. The type inversion lemma allows us to *reveal* the structure of a type of a given well-typed process. We state the weaken and contraction lemma first, then the type inversion lemma. The weakening property says that a fresh name in a type derivation can be added to the context of that type derivation, while the contraction property says that a name in a type derivation that is fresh to a typed process can be excluded from the context of that type derivation.

Lemma 5.3.17 (Weakening and Contraction)**.**

1 *(Weakening) If* Γ *is well-formed,* $\Gamma \vdash P : T, E$ *and* $x\#P, x\#\Gamma$*, and* $x\#E$ *then* $\Gamma, x : t \vdash P : T, E$*.*

2 *(Contraction) If* Γ *is well-formed,* $\Gamma, x : t \vdash P : T, E$ *and* $x\#P, \Gamma$*, then* $\Gamma \vdash P : T, E$*.*

Proof. It is straightforward by induction on the derivation of $\Gamma \vdash P : T, E$ by distinguishing the last typing rule applied. Many cases are obvious. We consider the most interesting cases.

(1) Weakening:

- the case of the rule [ts_input]: given that $\Gamma \vdash a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E$ is obtained from

$$\Gamma, y : t' \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (y : t')T_a \quad T_{2\uparrow\_} = T_a \quad y\#T_1.$$

We can safely assume that $x \neq y$ (by α-conversion if necessary). By induction hypothesis, we have $\Gamma, x : t, y : t' \vdash P : T_1 \parallel T_2, E$. This implies the required result

$$\Gamma, x : t \vdash a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E.$$

- the case of the rule [ts_input_res]: given that $\Gamma \vdash a(s)P : a.(T_1 \parallel T_{2\downarrow\_}), E$ which is obtained from

$$\Gamma, s : u \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (s : u)T_a \quad T_{2\uparrow\_} = T_a \quad s\#T_1.$$

We can safely assume that $x \neq s$ (by α-conversion if necessary). By induction hypothesis, we have $\Gamma, x : t, s : u \vdash P : T_1 \parallel T_2, E$. This implies the required result

$$\Gamma, x : t \vdash a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E$$

(2) Contraction:

- the case of the rule [ts_input]: given that $\Gamma, x : t \vdash a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E$ which is obtained from

$$\Gamma, x : t, y : t' \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (y : t')T_a \quad T_{2\uparrow\_} = T_a \quad y\#T_1.$$

By induction hypothesis, we have $\Gamma, y : t' \vdash P : T_1 \parallel T_2, E$. This implies the required result $\Gamma \vdash a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E$.

- the case of the rule [ts_input_res]: given that $\Gamma, x : t \vdash a(s)P : a.(T_1 \parallel T_{2\downarrow\_}), E$ which is obtained from

$$\Gamma, x : t, s : u \vdash P : T_1 \parallel T_2, E \quad \Gamma \vdash a : (s : u)T_a \quad T_{2\uparrow\_} = T_a \quad s\#T_1.$$

By induction hypothesis, we have $\Gamma, s : u \vdash P : T_1 \parallel T_2, E$. This implies the required result $\Gamma \vdash a(y)P : a.(T_1 \parallel T_{2\downarrow\_}), E$.

□

The inversion lemma given below shows that we can obtain the *shadow* structure of a process reflected in its type.

Lemma 5.3.18 (Type Inversion)**.** *Given* $\Gamma \vdash_N P : T, E$*,* $\Gamma \vdash a : (x : t)U_a$*,* $\Gamma \vdash_N r : res(\Phi_r)$*,* $\Gamma \vdash c : (x : res(\Phi))U_c$ *and* $\Gamma \vdash_N b : t,$*. Then for any* Q, Q_1, Q_2 *and* $\pi.Q$*, it holds that:*

- *1. If $P = a(x).Q$, then $T = a.(S \parallel U_{\downarrow\_})$ for some S such that $\Gamma \vdash_N Q : S \parallel U, E$, $x\#S$ and $U_{\uparrow\_} = U_a$.*
- *2. If $P = c(s).Q$, then $T = a.(S \parallel U_{\downarrow\_})$ for some S such that $\Gamma \vdash_N Q : S \parallel U, E$, $x\#S$ and $U_{\uparrow\_} = U_c$.*
- *3. If $P = \bar{a}b.Q$ then $T = \bar{a}.(S \parallel S')$ for some S such that $\Gamma \vdash_N Q : S, E$ and $S' = U_a\{b/x\}$.*
- *4. If $P = \bar{a}r.Q$ then $T = \bar{a}.(S \parallel S')$ for some S such that $\Gamma \vdash_N Q : S, E$, $\Phi_a \subseteq \Phi$ and $S' = U_a\{r/x\}$.*
- *5. If $P = \tau.Q$ then $T = \tau.S$ for some S such that $\Gamma \vdash_N Q : S, E$.*
- *6. If $P = (\nu x)Q$ then $T = (\nu x)S$ for some S and t' such that $\Gamma, z : t' \vdash_N Q : S, E$.*
- *7. If $P = Q_1 \parallel Q_2$ then $T = S_1 \parallel S_2$ for some S_1 and S_2 such that $\Gamma \vdash_N Q_1 : S_1, E_1$, $\Gamma \vdash_N Q_2 : S_2, E_2$ and $E \equiv E_1 \vee E_2$.*
- *8. If $P = Q_1 + Q_2$ then $T = S_1 + S_2$ for some S_1 and S_2 such that $\Gamma \vdash_N Q_1 : S_1, E_1$, $\Gamma \vdash_N Q_2 : S_2, E_2$ and $E \equiv E_1 \vee E_2$.*
- *9. If $P = !a(x).Q$ then $T = !a.S$ for some S such that $\Gamma \vdash_N a(x)Q : a.S, E'$ and $E = !E'$.*
- *10. If $P = (r, \varphi, \eta)\{Q\}$ then $T = T'_{\uparrow r}$ for some T', E' such that $\Gamma \vdash Q : T', E'$, $\Gamma \vdash r : res(\Phi)$, $\langle r, \varphi\rangle \in \Phi$ and $E = E' \vee \varphi\langle \eta.T'_{\downarrow r}\rangle$.*
- *11. If $P = req(s)\{Q\}$ then $T = T'_{\uparrow R}$ for some T', E' such that $\Gamma \vdash Q : T', E'$, $\Gamma \vdash s : res(\Phi)$, $S = rn(\Phi)$ and $E = E' \vee \bigvee_{\langle r, \varphi\rangle \in \Phi} \langle T'_{\downarrow r}\rangle \varphi$.*

Proof. It is straightforward by induction on the derivation of $\Gamma \vdash_N P : T, E$ by distinguishing the last typing rule applied. All cases are obvious (recall that the rule [ts_eq] cannot be the last applied one in a normal derivation).

□

Subject Congruence. The subject congruence lemma shows that a congruent process has the same type. In the following lemma, we prefer a more general formulation of side effects, since changes by structural congruence in processes also reflect changes in the corresponding side effects. s

Lemma 5.3.19 (subject congruence)**.** *If $\Gamma \vdash P : T, E$ and $P \equiv Q$ then there exists E' such that $\Gamma \vdash Q : T, E'$ and $E \equiv E'$.*

Proof. The proof proceeds by induction on the derivation of $P \equiv Q$ by distinguishing the last structural rule applied. Many cases are obvious. We consider the most interesting cases.

- the case of $P = (\nu a)P_1 \parallel P_2$ and $Q = (\nu a)(P_1 \parallel P_2)$, where $a\#P_2$: by applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $S \equiv T$ and $\Gamma \vdash_N P : S, E$. By applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E$, we have

  $$\begin{array}{ll} \Gamma \vdash_N (\nu a)P_1 : S_1, E_1 & \Gamma \vdash_N P_2 : S_2, E_2 \\ S = S_1 \parallel S_2 & E \equiv E_1 \vee E_2. \end{array}$$

  Again by Lemma 5.3.18

  $$\Gamma, a : t \vdash_N P_1 : S_1', E_1 \quad S_1 = (\nu a)S_1'.$$

  Moreover, $\Gamma \vdash_N P_2 : S_2, E_2$ and $a\#\Gamma, P_2$, hence by Lemma 5.3.17, we have that $\Gamma, a : t \vdash_N P_2 : S_2, E_2$. By applying the rule [ts_par] to $\Gamma, a : t \vdash_N P_1 : S_1', E_1$ and $\Gamma, a : t \vdash_N P_2 : S_2, E_2$, we have

  $$\Gamma, a : t \vdash_N P_1 \parallel P_2 : S_1' \parallel S_2, E.$$

  By the rule [ts_res], $\Gamma \vdash_N (\nu a)(P_1 \parallel P_2) : (\nu a)(S_1' \parallel S_2), E$. Since $T \equiv S = (\nu a)(S_1') \parallel S_2$. Note that given $\Gamma, a : t \vdash_N P_2 : S_2, E$ and $a\#\Gamma, P_2$, a is not a free name of S_2 (it is straightforward by induction of the type derivation), and therefore $T \equiv S = (\nu a)(S_1' \parallel S_2)$ By the rule [ts_eq], $\Gamma \vdash (\nu a)(P_1 \parallel P_2) : T, E$.

- the case of $P = (\nu a)(P_1 \parallel P_2)$ and $Q = (\nu a)P_1 \parallel P_2$, where $a\#P_2$: it is similar.

- the case of $P = (r, \varphi, \eta)\{P'\}$ and $Q = (r, \varphi, \eta)\{Q'\}$, where $P' \equiv Q'$: by applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $S \equiv T$ and $\Gamma \vdash_N P : S, E$. By applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E$, we have

  $$\begin{array}{lll} \Gamma \vdash_N P' : T', E' & S = T'_{\uparrow\text{-}} & E = E' \vee \varphi\langle \eta . T'_{\downarrow r} \rangle \\ \Gamma \vdash r : res(\Phi) & \langle r, \varphi \rangle \in \Phi & \end{array}$$

  and by induction hypothesis

  $$\Gamma \vdash_N Q' : T', E' \quad S = T'_{\uparrow r} \quad E = E' \vee \varphi\langle \eta . T'_{\downarrow r} \rangle.$$

  By applying the rule [ts_res_joint] to $\Gamma \vdash_N Q' : T', E'$, we get

  $$\Gamma \vdash_N Q : S, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S, E$.

- the case of $P = req(s)\{P'\}$ and $Q = req(s)\{Q'\}$, where $P' \equiv Q'$: by applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $\Gamma \vdash_N P : S, E$ and $S \equiv T$. By applying Lemma 5.3.18 to $\Gamma \vdash P : T, E$, we have

$$\begin{array}{l} \Gamma \vdash s : res(\Phi) \quad R = rn(\Phi) \\ \Gamma \vdash_N Q : T', E' \quad S = T'_{\uparrow R} \quad E = E' \vee \bigvee_{\langle r,\, \varphi\rangle \in \Phi} \langle T'_{\downarrow r}\rangle \varphi \end{array}$$

  by induction hypothesis

$$\Gamma \vdash_N Q' : T', E' \quad S = T'_{\uparrow s} \quad \Gamma \vdash s : res(\Phi) \quad E = E' \vee \bigvee_{\langle r,\, \varphi\rangle \in \Phi} \langle T'_{\downarrow r}\rangle \varphi.$$

  By applying the rule [ts_res_req] to $\Gamma \vdash_N Q' : T', E'$, we get

$$\Gamma \vdash_N Q : S, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S, E$.

- the case of $P = (\nu a)(r, \varphi, \eta)\{R\}$ and $Q = (r, \varphi, \eta)\{(\nu a)R\}$: by applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $\Gamma \vdash_N P : S, E$ and $S \equiv T$. By applying Lemma 5.3.18 to $\Gamma \vdash P : T, E$, we have

$$\begin{array}{l} \Gamma, a : t \vdash_N (r, \varphi, \eta)\{R\} : T', E \quad S = (\nu a)T' \\ \text{and} \\ \Gamma, a : t \vdash_N R : T'', E' \quad T' = T''_{\uparrow r} \quad E = E' \vee \varphi\langle \eta . T''_{\downarrow r}\rangle \\ \Gamma \vdash r : res(\Phi) \quad \langle r, \varphi\rangle \in \Phi \end{array}$$

  By applying the rule [ts_res] to $\Gamma, a : t \vdash_N R : T'', E'$

$$\Gamma \vdash_N (\nu a)R : (\nu a)T'', E' \quad .$$

  and note that $((\nu a)T'')_{\downarrow r} = T''_{\downarrow r}$, hence $E = E' \vee \varphi\langle \eta . T''_{\downarrow r}\rangle = E' \vee \varphi\langle \eta . ((\nu a)T'')_{\downarrow r}\rangle$ and $S = (\nu a)T' = (\nu a)T''_{\uparrow r} = ((\nu a)T'')_{\uparrow r}$. By applying the rule [ts_res_joint] to $\Gamma \vdash_N (\nu a)R : (\nu a)T'', E'$, we get

$$\Gamma \vdash_N Q : S, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S, E$.

- the case of $P = (r, \varphi, \eta)\{(\nu a)R\}$ and $Q = (\nu a)(r, \varphi, \eta)\{R\}$. By applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $\Gamma \vdash_N P : S, E$ and $S \equiv T$. By applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E$, we have

  $\Gamma \vdash_N (\nu a)R : T', E' \quad S = T'_{\uparrow r} \quad E = E' \vee \varphi\langle \eta.T'_{\downarrow r}\rangle$
  $\Gamma \vdash r : res(\Phi) \quad \langle r, \varphi\rangle \in \Phi$
  and
  $\Gamma, a : t \vdash_N R : T'', E' \quad T' = (\nu a)T''$

  Note that $S = T'_{\uparrow r} = ((\nu a)T'')_{\uparrow r} = (\nu a)T''_{\uparrow r}$ and $T''_{\downarrow r} = ((\nu a)T'')_{\downarrow r} = T'_{\downarrow r}$ implies $E \equiv E' \vee \varphi\langle \eta.T''_{\downarrow r}\rangle$. By applying the rule [ts_res_joint] to $\Gamma, a : t \vdash_N R : T'', E'$, we get

  $$\Gamma, a : t \vdash_N (r, \varphi, \eta)\{R\} : T''_{\uparrow r}, E'$$

  By the rule [ts_res] to $\Gamma, a : t \vdash_N (r, \varphi, \eta)\{R\} : T''_{\uparrow r}, E'$, we get

  $$\Gamma \vdash_N Q : S, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S, E$.

- the case of $P = (\nu a)req(s)\{P'\}$ and $Q = req(s)\{(\nu a)P'\}$. By applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $\Gamma \vdash_N P : S, E$ and $S \equiv T$. Now by applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E$, we have

  $\Gamma, a : t \vdash_N req(s)\{P'\} : T', E \quad S = (\nu a)T'$
  and
  $\Gamma \vdash s : res(\Phi) \quad R = rn(\Phi)$
  $\Gamma, a : t \vdash_N P' : T'', E' \quad T' = T''_{\uparrow R} \quad E = E' \vee \bigvee_{\langle r, \varphi\rangle \in \Phi} \varphi\langle \eta.T''_{\downarrow r}\rangle.$

  By applying the rule [ts_res] to $\Gamma, a : t \vdash_N P' : T'', E'$, we get

  $$\Gamma \vdash_N (\nu a)P' : (\nu a)T'', E'$$

  Note that $S = (\nu a)T' = (\nu a)T''_{\uparrow R} = ((\nu a)T'')_{\uparrow R}$. Moreover, $T''_{\downarrow r} = ((\nu a)T'')_{\downarrow r}$ implies $E = E' \vee \bigvee_{\langle r, \varphi\rangle \in \Phi} \varphi\langle \eta.T''_{\downarrow r}\rangle = E' \vee \bigvee_{\langle r, \varphi\rangle \in \Phi} \varphi\langle \eta.((\nu a)T'')_{\downarrow r}\rangle$. By applying the rule [ts_res_req] to $\Gamma \vdash_N (\nu a)P' : (\nu a)T'', E'$, we get

  $$\Gamma \vdash_N Q : S, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S, E$.

- the case of $P = req(s)\{(\nu a)P'\}$ and $Q = (\nu a)req(s)\{P'\}$: by applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $\Gamma \vdash_N P : S, E$ and $S \equiv T$. By applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E$, we have

  $$\begin{array}{l} \Gamma \vdash s : res(\Phi) \quad R = rn(\Phi) \\ \Gamma \vdash_N (\nu a)P' : T', E' \quad S = T'_{\uparrow R} \quad E = E' \vee \bigvee_{\langle r,\, \varphi\rangle \in \Phi} \varphi\langle \eta.T'_{\downarrow r}\rangle \\ \text{and} \\ \Gamma, a : t \vdash_N P' : T'', E' \quad T' = (\nu a)T'' \end{array}$$

  Note that $T''_{\downarrow R} = ((\nu a)T'')_{\downarrow R} = T'_{\downarrow}$ implies $E \equiv E' \vee \bigvee_{\langle r,\, \varphi\rangle \in \Phi} \varphi\langle \eta.T''_{\downarrow r}\rangle$. By applying the rule [ts_res_req] to $\Gamma, a : t \vdash_N P' : T'', E'$, we get

  $$\Gamma, a : t \vdash_N req(s)\{P'\} : T''_{\uparrow s}, E'$$

  Note that $S = T'_{\uparrow s} = ((\nu a)T'')_{\uparrow s} = (\nu a)T''_{\uparrow s}$. By applying the rule [ts_res] to $\Gamma, a : t \vdash_N req(s)\{P'\} : T''_{\uparrow s}, E'$, we get

  $$\Gamma \vdash_N Q : S, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S, E$.

- the case of $P = (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} \parallel (r_2, \varphi_2, \eta_2)\{P'\}$ and $Q = (r_2, \varphi_2, \eta_2)\{P' \parallel (r_1, \varphi_1, \eta_1)\{\mathbf{0}\}\}$. By applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $\Gamma \vdash_N P : S, E$ and $S \equiv T$. By applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E$, we have

  $$\begin{array}{l} S = T_1 \parallel T_2, E = E_1 \vee E_2 \\ \text{and} \\ \Gamma \vdash_N (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} : T_1, E_1 \quad T_1 = \mathbf{0} \quad E_1 = \varphi_1\langle \eta_1\rangle \\ \Gamma \vdash_N (r_2, \varphi_2, \eta_2)\{R\} : T_2, E_2 \\ \Gamma \vdash_N P' : T_2', E_2' \quad T_2 = T'_{2\uparrow r_2} \quad E_2 = E_2' \vee \varphi_2\langle \eta_2.T'_{2\downarrow r_2}\rangle \end{array}$$

  By applying the rule [ts_par] to $\Gamma \vdash_N (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} : T_1, E_1$ and $\Gamma \vdash_N P' : T_2', E_2'$, we get

  $$\Gamma \vdash_N (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} \parallel P' : T_1 \parallel T_2', E_2' \vee E_1$$

  Note that since $T_1 = \mathbf{0}$, $(T_1 \parallel T_2')_{\uparrow r_2} = T_1 \parallel T'_{2\uparrow r_2} = T_1 \parallel T_2$ and $T_1 \parallel T'_{2\downarrow r_2} = T'_{2\downarrow r_2}$. By applying rule rule [ts_resjoin] to $\Gamma \vdash_N (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} \parallel P' : T_1 \parallel T_2', E_2' \vee E_1$, we have

  $$\Gamma \vdash_N Q : S, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S, E$.

- the case of $P = (r_2, \varphi_2, \eta_2)\{R \parallel (r_1, \varphi_1, \eta_1)\{\mathbf{0}\}\}$ and $Q = (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} \parallel (r_2, \varphi_2, \eta_2)\{R\}$. By applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exists S such that $\Gamma \vdash_N P : S, E$ and $S \equiv T$. By applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E$, we have

  $\Gamma \vdash_N R \parallel (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} : T'E' \quad S = T'_{\uparrow r_2} \quad E = E' \vee \varphi_2\langle \eta_2 . T'_{\downarrow r_2}\rangle$
  and
  $\Gamma \vdash_N R : T_1, E_1 \quad \Gamma \vdash_N (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} : T_2, E_2 \quad T_2 = \mathbf{0}$
  with $E_2 = \varphi_1\langle \eta_1 \rangle, T' = T_1 \parallel T_2, E' = E_1 \vee E_2$.

  Since $S = T'_{\uparrow r_2} = (T_1 \parallel T_2)_{\uparrow r_2} \equiv T_{1\uparrow r_2}$ and $T_{1\downarrow r_2} \equiv (T_1 \parallel T_2)_{\downarrow r_2} = (T')_{\downarrow r_2}$, by the rule [ts_resjoint] and [ts_eq] to $\Gamma \vdash_N R : T_1, E_1$, we get

  $$\Gamma \vdash_N, E(r_2, \varphi_2, \eta_2)\{R\} : S, E_1 \vee \varphi_2\langle \eta_2 . T'_{\downarrow r_2}\rangle$$

  By applying the rule [ts_par] to

  $$\begin{array}{l}\Gamma \vdash_N (r_2, \varphi_2, \eta_2)\{R\} : S, E_1 \\ \Gamma \vdash_N (r_1, \varphi_1, \eta_1)\{\mathbf{0}\} : T_2, E_2,\end{array}$$

  we get

  $$\Gamma \vdash_N Q : S \parallel T_2, E$$

  and the required result $\Gamma \vdash Q : T, E$ follows by applying the rule [ts_eq] to $\Gamma \vdash_N Q : S \parallel T_2, E$ since $S \parallel T_2 \equiv S \equiv T$.

□

Subject Reduction. The subject reduction theorem establishes the semantic correctness of the type system. Intuitively, if a process P has a type T, then any evaluation of P has a type T' and there is a evaluation of T, called T'' such that T'' simulates T', i.e. $T' \preceq T''$. To prove it, we need to introduce some auxiliary technicalities.

Definition 5.3.20.

An evaluation context is defined by the following grammar:
$C = \bullet \mid C + P \mid P + C \mid C \parallel P \mid P \parallel C \mid (\nu z)C \mid (r, \varphi, \eta)\{C\} \mid req(s)\{C\}$

A typed evaluation context is defined by the following grammar:
$D = \bullet \mid D + T \mid T + D \mid D \parallel T \mid T \parallel D \mid (\nu z)D \mid D_{\uparrow S}$,

where S is a set of resource names.

The following lemma generalises the type inversion lemma. The result of the type inversion lemma is generalised to an arbitrary evaluation context.

Lemma 5.3.21. *Given a evaluation context C, if $\Gamma \vdash_N P : T, E$ and $\Gamma \vdash_N C[P] : T', E'$ then $T' = D[T]$, for some typed context D.*

Proof. The proof proceeds by induction on the structure of the context C. We show the most significant cases.

- The case of $C = \bullet$ is obvious.
- The case of $C + P'$: assume that $\Gamma \vdash_N C[P] + P' : T', E'$.

  By applying Lemma 5.3.18 to $\Gamma \vdash_N C[P] + P' : T', E'$, we get

  $$\begin{array}{l}\Gamma \vdash_N C[P] : T_1, E_1 \\ \Gamma \vdash_N P' : T_2, E_2\end{array}$$

  such that $T' = T_1 + T_2$ and $E \equiv E_1 \vee E_2$. By applying the induction hypothesis to $\Gamma \vdash_N C[P] : T_1, E$, there exists D s.t. $D'[T] = T_1$. Let $D = D' + T_2$, hence $D[T] = T'$. The required result is established.
- The case of $(r, \varphi, \eta)\{C\}$: assume that $\Gamma \vdash_N (r, \varphi, \eta)\{C[P]\} : T', E'$. By applying Lemma 5.3.18 to $\Gamma \vdash_N (r, \varphi, \eta)\{C[P]\} : T', E'$, we get

  $$\Gamma \vdash_N C[P] : S, E''$$

  such that $T' = S_{\uparrow r}$ and $E' = E'' \vee \varphi\langle S_{\downarrow r}\rangle$. By applying the induction hypothesis to $\Gamma \vdash_N C[P] : S, E''$, there exists D' such that $D'[T] = S$. Let $D = D'_{\uparrow r}$, hence $D[T] \equiv S_{\uparrow r} = T'$. The required result is established.
- The remaining cases can be proved by resorting to the similar argument.

□

Substitution of a name for another of the same type in a given type derivation results in a new type derivation.

Lemma 5.3.22 (substitution)**.** *Given $\Gamma, x; t \vdash P : T, E$, Γ and $\Gamma, x : t$ are well-formed. If $\Gamma \vdash b : t$ then $\Gamma\{b/x\} \vdash P\{b/x\} : T\{b/x\}, E\{b/x\}$.*

Proof. The proof proceeds by induction on the derivation of $\Gamma, x; t \vdash P : T, E$ by distinguishing the last typing rule applied. Without loss of generality, we can assume that all bound names are different from each others and from free names.

- The cases of the rules [ts_empty], [ts_choice], [ts_par], [ts_rep], [ts_res], [ts_act], [ts_tau] and [ts_eq] are straightforward.
- The case of the rule [ts_output]: by $\Gamma, x : t \vdash \bar{a}c.P : \bar{a}.(S \parallel T\{c/y\}), E$. Now, the premise of the rule allows us to derive

  $$\begin{array}{l} \Gamma, x : t \vdash a : (y : t')T \text{ with y\#\{x,b\}} \\ \Gamma, x : t \vdash c : t' \\ \Gamma, x : t \vdash P : S, E \end{array}$$

  By applying the induction hypothesis to $\Gamma, x : t \vdash P : S, E$, we get

  $$\Gamma\{b/x\} \vdash P\{b/x\} : S\{b/x\}, E\{b/x\}$$

  Moreover, by definition of the well-formed context,

  $$\begin{array}{l} \Gamma\{b/x\} \vdash c\{b/x\} : t'\{b/x\} \\ \Gamma\{b/x\} \vdash a\{b/x\}(y : t'\{b/x\})T\{b/x\} \end{array}$$

  Therefore,

  $$\Gamma\{b/x\} \vdash \bar{a}\{b/x\}c\{b/x\}.P\{b/x\} : \bar{a}\{b/x\}.(S\{b/x\} \parallel T\{b/x\}\{c\{b/x\}/y\}), E\{b/x\},$$

  that is $\Gamma\{b/x\} \vdash (\bar{a}c.P)\{b/x\} : (\bar{a}.(S \parallel T\{c/y\}))\{b/x\}, E\{b/x\}$.
- The case of the rule [ts_output_res]: by $\Gamma, x : t \vdash \bar{a}r.P : \bar{a}.(S \parallel T\{r/s\}), E$. Now, the premise of the rule allows us to derive

  $$\begin{array}{l} \Gamma, x : t \vdash a : (s : res(\Phi))T \text{ with s\#\{x,b\}} \\ \Gamma, x : t \vdash r : res(\theta_r) \text{ with } \theta_r \in \Phi \\ \Gamma, x : t \vdash P : S, E \end{array}$$

  By applying the induction hypothesis to $\Gamma, x : t \vdash P : S, E$, we get

  $$\Gamma\{b/x\} \vdash P\{b/x\} : S\{b/x\}, E\{b/x\}.$$

  Moreover, by definition of the well-formed context,

  $$\begin{array}{l} \Gamma\{b/x\} \vdash r\{b/x\} : res(\theta_r\{b/x\}) \\ \Gamma\{b/x\} \vdash a\{b/x\} : (s : res(\Phi\{b/x\}))T\{b/x\}. \end{array}$$

  Therefore,

  $$\Gamma\{b/x\} \vdash \bar{a}\{b/x\}r\{b/x\}.P\{b/x\} : \bar{a}\{b/x\}.(S\{b/x\} \parallel T\{b/x\}\{r\{b/x\}/s\}), E\{b/x\},$$

  that is $\Gamma\{b/x\} \vdash (\bar{a}r.P)\{b/x\} : (\bar{a}.(S \parallel T\{r/s\}))\{b/x\}, E\{b/x\}$.

- The case of the rule [ts_input]: by $\Gamma, x : t \vdash a(y).P : a.(S \parallel T_{\downarrow\_}), E$. Now, the premise of the rule allows us to derive

  $$\begin{array}{l}\Gamma, x : t \vdash a : (y : t')T' \text{ with y\#\{x,b\}} \\ \Gamma, x : t \vdash P : S \parallel T, E \text{ with } y\#S \text{ and } T' = T_{\uparrow\_}\end{array}$$

  By applying the induction hypothesis to $\Gamma, x : t \vdash P : S \parallel T, E$, we get

  $$\Gamma\{b/x\} \vdash P\{b/x\} : S\{b/x\} \parallel T\{b/x\}, E\{b/x\}.$$

  Moreover, by definition of the well-formed context,

  $$\begin{array}{l}\Gamma\{b/x\} \vdash a\{b/x\} : (y : t'\{b/x\})T'\{b/x\} \\ y\#S\{b/x\} \\ T'\{b/x\} = T_{\uparrow\_}\{b/x\} = (T\{b/x\})_{\uparrow\_}\end{array}$$

  Therefore,

  $$\Gamma\{b/x\} \vdash a\{b/x\}(y).P\{b/x\} : a\{b/x\}.(S\{b/x\} \parallel (T\{b/x\})_{\downarrow\_}), E\{b/x\},$$

  that is $\Gamma\{b/x\} \vdash (a(y).P)\{b/x\} : (a.(S \parallel T_{\downarrow\_}))\{b/x\}, E\{b/x\}$.

- The case of the rule [ts_input_res]: by $\Gamma, x : t \vdash a(y).P : a.(S \parallel T_{\downarrow\_}), E$. Now, the premise of the rule allows us to derive

  $$\begin{array}{l}\Gamma, x : t \vdash a : (s : u)T' \text{ with s\#\{x,b\}} \\ \Gamma, x : t \vdash P : S \parallel T, E \text{ with } s\#S \text{ and } T' = T_{\uparrow\_}\end{array}$$

  By applying the induction hypothesis to $\Gamma, x : t \vdash P : S \parallel T, E$, we get

  $$\Gamma\{b/x\} \vdash P\{b/x\} : S\{b/x\} \parallel T\{b/x\}, E\{b/x\}.$$

  Moreover, by definition of the well-formed context,

  $$\begin{array}{l}\Gamma\{b/x\} \vdash a\{b/x\} : (s : u\{b/x\})T'\{b/x\} \\ s\#S\{b/x\} \\ T'\{b/x\} = T_{\uparrow\_}\{b/x\} = (T\{b/x\})_{\uparrow\_}\end{array}$$

  Therefore,

  $$\Gamma\{b/x\} \vdash a\{b/x\}(s).P\{b/x\} : a\{b/x\}.(S\{b/x\} \parallel (T\{b/x\})_{\downarrow\_}), E\{b/x\},$$

  that is $\Gamma\{b/x\} \vdash (a(s).P)\{b/x\} : (a.(S \parallel T_{\downarrow\_}))\{b/x\}, E\{b/x\}$.

- The case of the rule [ts_res_join]: by $\Gamma, x : t \vdash (r, \varphi, \eta)\{P\} : T_{\uparrow r}, E \vee \varphi\langle \eta . T_{\downarrow r}\rangle$. Now, the premise of the rule allows us to derive

$$\Gamma, x : t \vdash r : res(\Phi) \text{ with } \langle r, \varphi\rangle \in \Phi$$
$$\Gamma, x : t \vdash P : T, E \text{ with } T' = T_{\uparrow r}$$

  By applying the induction hypothesis to $\Gamma, x : t \vdash P : T, E$, we get

$$\Gamma\{b/x\} \vdash P\{b/x\} : T\{b/x\}, E\{b/x\}.$$

  Moreover, by definition of the well-formed context,

$$\Gamma\{b/x\} \vdash r\{b/x\} : res(\Phi\{b/x\})$$
$$\langle r\{b/x\}, \varphi\{b/x\}\rangle \in \Phi\{b/x\}.$$

  Therefore,

$$\Gamma\{b/x\} \vdash (r\{b/x\}, \varphi, \eta)\{P\{b/x\}\} : (T\{b/x\})_{\uparrow r\{b/x\}}, E\{b/x\},$$

  that is $\Gamma\{b/x\} \vdash (r, \varphi, \eta)\{P\}\{b/x\} : T_{\uparrow r}\{b/x\}, E\{b/x\}$.

- The case of the rule [ts_res_req]: by $\Gamma, x : t \vdash req(r)\{P\} : T_{\uparrow R}, E \vee \bigvee_{\langle r,\, \varphi\rangle \in \Phi} \langle\varphi\rangle T_{\downarrow R}$, where $R = rn(\Phi)$. Now, the premise of the rule allows us to derive

$$\Gamma, x : t \vdash s : res(\Phi)$$
$$\Gamma, x : t \vdash P : T, E \text{ with } T' = T_{\uparrow R}$$

  By applying the induction hypothesis to $\Gamma, x : t \vdash P : T, E$, we get

$$\Gamma\{b/x\} \vdash P\{b/x\} : T\{b/x\}, E\{b/x\}.$$

  Moreover, by definition of the well-formed context,

$$\Gamma\{b/x\} \vdash s\{b/x\} : res(\Phi\{b/x\}),\ R\{b/x\} = rn(\Phi\{b/x\}).$$

  Therefore,

$$\Gamma\{b/x\} \vdash req(s\{b/x\})\{P\{b/x\}\} : (T\{b/x\})_{\uparrow R\{b/x\}}, E\{b/x\},$$

  that is $\Gamma\{b/x\} \vdash req(s)\{P\}\{b/x\} : T_{\uparrow R}\{b/x\}, E\{b/x\}$.

□

Now we are ready to prove the subject reduction theorem. The proof proceeds in three steps, which correspond to the three kinds of transition labels in the semantics of types.

Lemma 5.3.23. *Given $\Gamma \vdash P : T, E$, if $P \overset{\langle a \rangle}{\to} P'$ then $\Gamma \vdash P' : T', E'$ and $T \overset{\langle a \rangle}{\to} T''$ for some E', T', T'' such that $T' \preceq T''$. Moreover. if E is satisfied then E' is also satisfied.*

Proof. The proof proceeds by induction on the depth of the derivation of $P \overset{\langle a \rangle}{\to} P'$, where the last rule applied is distinguished. Without loss of generality, we can safely assume that all bound names are different from each other and from free names.

- The base case of the rule [Comm]: we proceed by induction on the structure of evaluation contexts
  - The base case $P = a(y).P_1 \parallel \bar{a}b.P_2 \overset{\langle a \rangle}{\to} P_1\{b/a\} \parallel P_2$: Suppose $\Gamma \vdash P : T, E$. By Lemma 5.3.14 and 5.3.18, we have

    $\Gamma \vdash a : (y : t)T_a \quad \Gamma \vdash b : t$
    $\Gamma, y : t \vdash_N P_1 : T_1 \parallel T_3, E_1 \quad y\#T_1 \quad \Gamma \vdash_N a(y).P_1 : a.(T_1 \parallel T_{3\downarrow\_}), E_1$
    $\Gamma \vdash_N P_2 : T_2, E_2 \quad \Gamma \vdash_N \bar{a}b.P_2 : \bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/a\}), E_2,$

    where $T \equiv a.(T_1 \parallel T_{3\downarrow\_}) \parallel \bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/y\})$, $E \equiv E_1 \vee E_2$ and $T_{3\uparrow\_} = T_a$. We have

    $$T \equiv a.(T_1 \parallel T_{3\downarrow\_}) \parallel \bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/y\}) \overset{\langle a \rangle}{\to} T_1 \parallel T_{3\downarrow\_} \parallel T_2 \parallel T_{3\uparrow\_}\{b/y\} \triangleq T'$$

    and by Lemma 5.3.22 and $y\#T_1$

    $$\Gamma \vdash_N P_1\{b/y\} : T_1 \parallel T_3\{b/y\}, E_1$$

    and therefore

    $$\Gamma \vdash_N P_1\{b/y\} \parallel P_2 : T_1 \parallel T_2 \parallel T_3\{b/y\}, E_1 \vee E_2.$$

    We need to show that $T_1 \parallel T_2 \parallel T_3\{b/y\} \preceq T'$. By Lemma 5.3.13, $T_3 \preceq T_{3\downarrow\_} \parallel T_{3\uparrow\_}$, therefore

    $$T_3\{b/y\} \preceq T_{3\downarrow\_}\{b/y\} \parallel T_{3\uparrow\_}\{b/y\} \equiv T_{3\downarrow\_} \parallel T_{3\uparrow\_}\{b/y\}$$

    hence,

    $$T_1 \parallel T_2 \parallel T_3\{b/y\} \preceq T'.$$

    Moreover, $E \equiv E_1 \vee E_2$ implies that if E is satisfied, then $E_1 \vee E_2$ is also satisfied.
  - The base case $P = !a(y).P_1 \parallel \bar{a}b.P_2 \overset{\langle a \rangle}{\to} !a(y).P_1 \parallel P_1\{b/a\} \parallel P_2$: Suppose $\Gamma \vdash P : T, E$. By Lemmas 5.3.14 and 5.3.18, we have

    $\Gamma \vdash a : (y : t)T_a \quad \Gamma \vdash b : t$
    $\Gamma, y : t \vdash_N P_1 : T_1 \parallel T_3, E_1 \quad y\#T_1$
    $\Gamma \vdash_N !a(y).P_1 : !a.(T_1 \parallel T_{3\downarrow\_}), !E_1$
    $\Gamma \vdash_N a(y).P_1 : a.(T_1 \parallel T_{3\downarrow\_}), E_1$
    $\Gamma \vdash_N P_2 : T_2, E_2 \quad \Gamma \vdash_N \bar{a}b.P_2 : \bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/a\}), E_2,$

where $T \equiv !a.(T_1 \parallel T_{3\downarrow\_}) \parallel \bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/y\})$, $E \equiv !E_1 \vee E_2$ and $T_{3\uparrow\_} = T_a$. We have

$$T \equiv !a.(T_1 \parallel T_{3\downarrow\_}) \parallel \bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/y\}) \xrightarrow{\langle a \rangle} !a.(T_1 \parallel T_{3\downarrow\_}) \parallel T_1 \parallel T_{3\downarrow\_} \parallel T_2 \parallel T_{3\uparrow\_}\{b/y\} \triangleq T'$$

and by Lemma 5.3.22 and $y \# T_1$

$$\Gamma \vdash_N P_1\{b/y\} : T_1 \parallel T_3\{b/y\}, E_1$$

and therefore

$$\Gamma \vdash_N !a(y).P_1 \parallel P_1\{b/y\} \parallel P_2 : !a.(T_1 \parallel T_{3\downarrow\_}) \parallel T_1 \parallel T_2 \parallel T_3\{b/y\}, !E_1 \vee E_2.$$

We need to show that $!a.(T_1 \parallel T_{3\downarrow\_}) \parallel T_1 \parallel T_2 \parallel T_3\{b/y\} \preceq T'$. By Lemma 5.3.13, $T_3 \preceq T_{3\downarrow\_} \parallel T_{3\uparrow\_}$, therefore

$$T_3\{b/y\} \preceq T_{3\downarrow\_}\{b/y\} \parallel T_{3\uparrow\_}\{b/y\} \equiv T_{3\downarrow\_} \parallel T_{3\uparrow\_}\{b/y\}$$

hence,

$$!a.(T_1 \parallel T_{3\downarrow\_}) \parallel T_1 \parallel T_2 \parallel T_3\{b/y\} \preceq T'.$$

Moreover, $E \equiv !E_1 \vee E_2$ implies that if E is satisfied then $!E_1 \vee E_2$ is also satisfied.

– The induction case $P = C_1[a(y).P_1] + P_3 \parallel C_2[\bar{a}b.P_2] \xrightarrow{\langle a \rangle} C_1'[P_1\{b/a\}] \parallel C_2'[P_2]$: Suppose $\Gamma \vdash P : T, E$. By Lemmas 5.3.14 and 5.3.18, we have

$$\begin{array}{l}\Gamma \vdash_N C_1[a(y).P_1] : T_1, E_1 \\ \Gamma \vdash_N C_2[\bar{a}b.P_2] : T_2, E_2 \\ \Gamma \vdash_N P_3 : T_3, E_3,\end{array}$$

where $T \equiv T_1 \parallel T_2 \parallel T_3$ and $E = E_1 \vee E_2 \vee E_3$. By induction hypothesis applying to $C_1[a(y).P_1] \parallel C_2[\bar{a}b.P_2]$, there exist T', T'' such that

$$\begin{array}{l}\Gamma \vdash C_1'[P_1\{b/y\}] \parallel C_2'[P_2] : T' \\ T_1 \parallel T_2 \xrightarrow{\langle a \rangle} T'' \text{ and } T' \preceq T''.\end{array}$$

This implies the required result $\Gamma \vdash P_1\{b/y\}] \parallel P_2 : T'$, $T_1 + T_3 \parallel T_2 \xrightarrow{\langle a \rangle} T''$ and $T' \preceq T''$.

– The induction case $P = C_1[!a(y).P_1] + P_3 \parallel C_2[\bar{a}b.P_2] \xrightarrow{\langle a \rangle} C_1'[!a(y).P_1 \parallel P_1\{b/a\}] \parallel C_2'[P_2]$: it is similar to the above argument.

– The induction case $P = C_1[a(y).P_1] \parallel C_2[\bar{a}b.P_2] + P_3 \xrightarrow{\langle a \rangle} C_1'[P_1\{b/a\}] \parallel C_2'[P_2]$: it is similar to the above argument.

– The induction case $P = C_1[!a(y).P_1] \parallel C_2[\bar{a}b.P_2] + P_3 \xrightarrow{\langle a \rangle} C_1'[!a(y).P_1 \parallel P_1\{b/a\}] \parallel C_2'[P_2]$: it is similar to the above argument.

– The induction case $P = C_1[a(y).P_1] \parallel C_2[\bar{a}b.P_2] \parallel P_3 \overset{\langle a\rangle}{\to} C_1'[P_1\{b/a\}] \parallel C_2'[P_2] \parallel P_3$: it is similar to the above argument.

– The induction case $P = C_1[!a(y).P_1] \parallel C_2[\bar{a}b.P_2] \parallel P_3 \overset{\langle a\rangle}{\to} C_1'[!a(y).P_1 \parallel P_1\{b/a\}] \parallel C_2'[P_2] \parallel P_3$: it is similar to the above argument.

– The induction case $P = (\nu z)(C_1[a(y).P_1]) \parallel C_2[\bar{a}b.P_2] \overset{\langle a\rangle}{\to} (\nu z)C_1'[P_1\{b/a\}] \parallel C_2'[P_2]$: By assumptions at the beginning, bound names are different from each other and from free names. We have

$$P \equiv (\nu z)(C_1[a(y).P_1] \parallel C_2[\bar{a}b.P_2])$$

Applying induction hypothesis to $C_1[a(y).P_1] \parallel C_2[\bar{a}b.P_2]$, we get the required result.

– The induction case $P = (\nu z)(C_1[!a(y).P_1]) \parallel C_2[\bar{a}b.P_2] \overset{\langle a\rangle}{\to} (\nu z)C_1'[!a(y).P_1 \parallel P_1\{b/a\}] \parallel C_2'[P_2]$: it is similar to the above argument.

– The induction case $P = C_1[a(y).P_1] \parallel (\nu z)(C_2[\bar{a}b.P_2]) \overset{\langle a\rangle}{\to} C_1'[P_1\{b/a\}] \parallel (\nu z)(C_2'[P_2])$: it is similar to the above argument.

– The induction case $P = C_1[!a(y).P_1] \parallel (\nu z)(C_2[\bar{a}b.P_2]) \overset{\langle a\rangle}{\to} C_1'[!a(y).P_1 \parallel P_1\{b/a\}] \parallel (\nu z)(C_2'[P_2])$: it is similar to the above argument.

– The induction case $P = (r, \varphi, \eta)\{C_1[a(y).P_1]\} \parallel C_2[\bar{a}b.P_2] \overset{\langle a\rangle}{\to} (r, \varphi, \eta)\{ C_1'[P_1\{b/a\}]\} \parallel C_2'[P_2]$: Suppose

$$\Gamma \vdash P : T, E$$

We assume that C_1 and C_2 do not contains *restrictions*. Otherwise, by structural rules restrictions can be moved to the top level and then we can apply induction hypothesis to the process inside the scope of restrictions by the above cases. Also assume that C_1 and C_2 are simple evaluation contexts (i.e. the hole in these contexts is a single-operand summation, for the general case it is similar), that is

$$P \overset{\langle a\rangle}{\to} (r, \varphi, \eta)\{C_1[P_1\{b/x\}]\} \parallel C_2[P_2]$$

By Lemmas 5.3.14, 5.3.18 and 5.3.21, we have

$$\begin{array}{l} \Gamma \vdash a : (y:t)T_a \quad \Gamma \vdash b : t \\ \Gamma, y : t \vdash_N P_1 : T_1 \parallel T_3, E_{11} \quad\quad y\#T_1 \quad\quad \Gamma \vdash_N C_1[a(y).P_1] : D_1[a.(T_1 \parallel T_{3\downarrow\_})], E_1 \\ \Gamma \vdash_N P_2 : T_2, E_{22} \quad\quad \Gamma \vdash_N C_2[\bar{a}b.P_2] : D_2[\bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/a\})], E_2, \end{array}$$

where $T \equiv D_1[a.(T_1 \parallel T_{3\downarrow\_})] \parallel D_2[\bar{a}.(T_2 \parallel T_{3\uparrow\_}\{b/y\})]$, $E \equiv E_1 \vee E_2$ and $T_{3\uparrow\_} = T_a$. Since D_1 and D_2 preserve the input/output transitions, we have (it is easy to see that D_1 and D_2 contain no restriction and the operator $\uparrow_R$ does not modify the input/output transitions), we have

$$D_1[a.(T_1 \parallel T_{3\downarrow\text{-}})] \parallel D_2[\bar{a}.(T_2 \parallel T_{3\uparrow\text{-}}\{b/y\})] \xrightarrow{\langle a\rangle} D_1[T_1 \parallel T_{3\downarrow\text{-}}] \parallel D_2[T_2 \parallel T_{3\uparrow\text{-}}\{b/y\}] \triangleq T''$$

Note that y is a bound name and E_{11} does not contain any bound name, it follows that $E_{11}\{b/y\} = E_{11}$. By Lemma 5.3.22 and $y\#T_1$

$$\begin{array}{l}\Gamma \vdash_N P_1\{b/y\} : T_1 \parallel T_3\{b/y\}, E_{11}\\ \Gamma \vdash_N C_1[P_1\{b/y\}] : D_1[T_1 \parallel T_3\{b/y\}], E'_1\\ \Gamma \vdash_N C_2[P_2] : D_2[T_2], E'_2\\ \Gamma \vdash_N C_1[P_1\{b/y\}] \parallel C_2[P_2] : D_1[T_1 \parallel T_3\{b/y\}] \parallel D_2[T_2], E'_1 \vee E'_2.\end{array}$$

for some E'_1, E'_2. Moreover, it is easy to prove that $(T_1 \parallel T_3)_{\downarrow\text{-}} \sim (T_1 \parallel T_3\{b/y\})_{\downarrow\text{-}}$ and $(T_2)_{\downarrow\text{-}} \sim (T_2 \parallel T_{3\uparrow\text{-}}\{b/y\})_{\downarrow\text{-}}$. This implies that $E \sim E'_1 \vee E'_2$. Hence, if E is satisfied, then $E'_1 \vee E'_2$ is also satisfied.
Now, we need to show that $T' \triangleq D_1[T_1 \parallel T_3\{b/y\}] \parallel D_2[T_2] \preceq T''$. By Lemma 5.3.13, $T_3 \preceq T_{3\downarrow\text{-}} \parallel T_{3\uparrow\text{-}}$, and therefore

$$T_3\{b/y\} \preceq T_{3\downarrow\text{-}}\{b/y\} \parallel T_{3\uparrow\text{-}}\{b/y\} \equiv T_{3\downarrow\text{-}} \parallel T_{3\uparrow\text{-}}\{b/y\}$$

Note that $(T_{3\uparrow\text{-}}\{b/y\})_{\uparrow R} = T_{3\uparrow\text{-}}\{b/y\}$. So, we have

$$D_1[T_1 \parallel T_3\{b/y\}] \preceq D_1[T_1 \parallel T_{3\downarrow\text{-}} \parallel T_{3\uparrow\text{-}}\{b/y\}] \equiv D_1[T_1 \parallel T_{3\downarrow\text{-}}] \parallel T_{3\uparrow\text{-}}\{b/y\}$$

hence,

$$\begin{aligned}D_1[T_1 \parallel T_3\{b/y\}] \parallel D_2[T_2] &\preceq D_1[T_1 \parallel T_{3\downarrow\text{-}}] \parallel T_{3\uparrow\text{-}}\{b/y\} \parallel D_2[T_2]\\ &\equiv D_1[T_1 \parallel T_{3\downarrow\text{-}}] \parallel D_2[T_2 \parallel T_{3\uparrow\text{-}}\{b/y\}] = T''\end{aligned}$$

This establishes the required result.

- The induction case $P = (r, \varphi, \eta)\{C_1[!a(y).P_1]\} \parallel C_2[\bar{a}b.P_2] \xrightarrow{\langle a\rangle} (r, \varphi, \eta)\{C'_1[!a(y).P_1 \parallel P_1\{b/a\}]\} \parallel C'_2[P_2]$: it is similar to the above argument.
- The induction case $P = C_1[a(y).P_1] \parallel (r, \varphi, \eta)\{C_2[\bar{a}b.P_2]\} \xrightarrow{\langle a\rangle} C'_1[P_1\{b/a\}] \parallel (r, \varphi, \eta)\{C'_2[P_2]\}$: it is similar to the above argument.
- The induction case $P = C_1[!a(y).P_1] \parallel (r, \varphi, \eta)\{C_2[\bar{a}b.P_2]\} \xrightarrow{\langle a\rangle} C'_1[!a(y).P_1 \parallel P_1\{b/a\}] \parallel (r, \varphi, \eta)\{C'_2[P_2]\}$: it is similar to the above argument.
- The induction case $P = req(r)\{C_1[a(y).P_1]\} \parallel C_2[\bar{a}b.P_2] \xrightarrow{\langle a\rangle} req(r)\{C'_1[P_1\{b/a\}]\} \parallel C'_2[P_2]$: it is similar to the above argument.
- The induction case $P = req(r)\{C_1[!a(y).P_1]\} \parallel C_2[\bar{a}b.P_2] \xrightarrow{\langle a\rangle} req(r)\{C'_1[!a(y).P_1 \parallel P_1\{b/a\}]\} \parallel C'_2[P_2]$: it is similar to the above argument.

  - The induction case $P = C_1[a(y).P_1] \parallel req(r)\{C_2[\bar{a}b.P_2]\} \overset{\langle a\rangle}{\rightarrow} C_1'[P_1\{b/a\}] \parallel req(r)\{C_2'[P_2]\}$: it is similar to the above argument.
  - The induction case $P = C_1[!a(y).P_1] \parallel req(r)\{C_2[\bar{a}b.P_2]\} \overset{\langle a\rangle}{\rightarrow} C_1'[!a(y).P_1 \parallel P_1\{b/a\}] \parallel req(r)\{C_2'[P_2]\}$: it is similar to the above argument.

- Induction cases of [ts_par], [ts_choice], [ts_cong], [ts_local$_1$], [ts_local$_2$] are straightforward.

□

Example 5.3.24. Let us consider the following process:

$$P = (r, \varphi, \epsilon)\{\bar{a} \parallel a \parallel \alpha(r) \parallel \beta(r)\}$$

under the type environment $\Gamma = a : ()\mathbf{0}$ and the policy $\varphi = \mathbf{G}\neg\beta$. The type derivation of P is as follows:

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma \vdash \bar{a} : \bar{a}, \xi} \quad \cfrac{\vdots}{\Gamma \vdash a \parallel \alpha(r) \parallel \beta(r) : a \parallel \alpha(r) \parallel \beta(r), \xi}}{\Gamma \vdash \bar{a} \parallel a \parallel \alpha(r) \parallel \beta(r) : \bar{a} \parallel a \parallel \alpha(r) \parallel \beta(r), \xi}\,[\text{ts\_par}]}{\Gamma \vdash P : T, E}$$

where $T = \bar{a} \parallel a$ and $E = \varphi\langle\alpha \parallel \beta\rangle$. Now, suppose that $P \overset{\langle a\rangle}{\rightarrow} P_1 = (r, \varphi, \epsilon)\{\alpha(r) \parallel \beta(r)\}$. We have:

$$\cfrac{\vdots}{\Gamma \vdash P_1 : T_1, E}$$

where $T_1 = \mathbf{0}$. It is easy to see that the semantics of the type T can perform the communication $\langle a\rangle$ as well, that is, $T = \bar{a} \parallel a \overset{\langle a\rangle}{\rightarrow} \mathbf{0} = T'$. Moreover, we have $T' = T_1$.

Lemma 5.3.25. *Given a judgement $\Gamma \vdash P : T, E$, if $P \overset{\mu}{\rightarrow} P'$, $\mu = \tau$, $\overline{\alpha(r)}$ or $\alpha(r)$ then $\Gamma \vdash P' : T', E'$ and $T \overset{\tau}{\rightarrow}^* T''$ for some E', T', T'' such that $T'_{\uparrow\text{-}} \preceq T''_{\uparrow\text{-}}$. Moreover if E is satisfied then E' is also satisfied.*

Proof. The proof proceeds by induction on the depth of the derivation of $P \overset{\mu}{\rightarrow} P'$, where the last rule applied is distinguished. Without loss of generality, we can safely assume that all bound names are different from each other and from free names.

- The base case of [Act]: suppose $P = \tau.P'$. It is straightforward.
- The base case of [Acquire]: assume that $P = (r, \varphi, \eta)\{\mathbf{0}\} \parallel req(r)\{P'\}$ and

$$\Gamma \vdash P : T, E$$
$$(r, \varphi, \eta)\{\mathbf{0}\} \parallel req(r)\{P'\} \overset{\tau}{\rightarrow} (r, \varphi, \eta)\{P'\}$$

By applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, we have $\Gamma \vdash_N P : S, E'$ such that $S \equiv T$ and $E' \equiv E$. By applying Lemma 5.3.18 to $\Gamma \vdash_N P : S, E'$, we get

$$\begin{array}{l}\Gamma \vdash_N (r, \varphi, \eta)\{\mathbf{0}\} : \mathbf{0}, \varphi\langle\eta\rangle \\ \Gamma \vdash r : res(\Phi) \text{ with } \langle r, \varphi\rangle \in \Phi \\ \Gamma \vdash_N req(r)\{P'\} : T_{1\uparrow R}, E_1 \vee \bigvee_{\langle r,\, \varphi\rangle \in \Phi} \langle T_{1\downarrow r}\rangle\varphi \\ \Gamma \vdash_N P' : T_1, E_1,\end{array}$$

where $R = rn(\Phi)$, $S = \mathbf{0} \parallel T_{1\uparrow R}$ and $E' \equiv E_1 \vee \bigvee_{\langle r,\, \varphi\rangle \in \Phi} \langle T_{1\downarrow r}\rangle\varphi \vee \varphi\langle\eta\rangle$. By applying the rule [ts_resjoint] to $\Gamma \vdash_N P' : T_1, E_1$, we get

$$\Gamma \vdash_N (r, \varphi, \eta)\{P'\} : T_{1\uparrow r}, E_1 \vee \varphi\langle\eta.T_{2\downarrow r}\rangle$$

Let $T' = T_{1\uparrow r}$ and $T'' = \mathbf{0} \parallel T'_{\uparrow R}$. Now, we need to show that $(T'_{\uparrow r})_{\uparrow\_} \preceq (T'_{\uparrow R})_{\uparrow\_}$. It is easily derived from $(T'_{\uparrow r})_{\uparrow\_} \equiv (T''_{\uparrow R})_{\uparrow\_}$. Moreover, if the constraints $\varphi\langle\eta\rangle$ and $\langle T'_{\downarrow r}\rangle\Phi$ are satisfied, then $\varphi\langle\eta.T'_{\downarrow r}\rangle$ is also satisfied. This implies the required result.

- the base case of [Release]: suppose $P = (r, \varphi, \eta)\{P'\}$. We proceed by induction on the structure of P'. Many cases are obvious. We consider only the most interesting cases.

  – The base case of $P = (r, \varphi, \eta)\{rel(r).P_1\} \xrightarrow{\tau} P = (r, \varphi, \eta.rel)\{\mathbf{0}\} \parallel P_1$: By applying Lemma 5.3.14 to $\Gamma \vdash P : T, E$, there exist S, E' such that

  $$\Gamma \vdash_N (r, \varphi, \eta)\{P\} : S, E',$$

  where $S \equiv T$ and $E' \equiv E$. By applying Lemma 5.3.18 to $\Gamma \vdash_N (r, \varphi, \eta)\{P\} : S, E'$, we get

  $$\begin{array}{l}\Gamma \vdash_N P_1 : T_1, E'' \text{ for some } T_1 \text{ and } E'' \\ \Gamma \vdash_N rel(r).P_1 : rel(r).T_1, E'',\end{array}$$

  where $E' = E'' \vee \varphi\langle\eta.(rel(r).T_1)_{\downarrow r}\rangle$ and $S = (rel(r).T_1)_{\uparrow r} = \tau.T_{1\uparrow r}$. By applying the rule [ts_par] to $\Gamma \vdash_N (r, \varphi, \eta.rel)\{\mathbf{0}\} : \mathbf{0}, \varphi\langle\eta.rel\rangle$ and $\Gamma \vdash_N P_1 : T_1, E''$, we have

  $$\Gamma \vdash_N (r, \varphi, \eta.rel)\{\mathbf{0}\} \parallel P_1 : T_1 \parallel \mathbf{0}, E'' \vee \varphi\langle\eta.rel\rangle$$

  Now, we can see that

  $$T \equiv \tau.T_{1\uparrow r} \xrightarrow{\tau} T_{1\uparrow r} \triangleq T''$$

  and let $T' \triangleq T_1 \equiv \mathbf{0} \parallel T_1$. It follows $T'_{\uparrow\_} = T''_{\uparrow\_}$, hence $T'_{\uparrow\_} \preceq T''_{\uparrow\_}$. It is easy to prove that whenever $\varphi\langle\eta.rel.T_{1\downarrow r}\rangle$ is true, so is $\varphi\langle\eta.rel\rangle$. Hence, the required result follows.

- The base case of [Policy$_1$]: it is similar to the above argument.
- The base case of [Policy$_2$]: it is similar to the above argument.
- The induction cases of [Par], [Choice], [Cong], [Res], [Local$_1$], [Local$_2$] are straightforward.

$\square$

Example 5.3.26. Back to Example 5.3.24. If $P \overset{\alpha(r)}{\rightarrow} P_2 = (r, \varphi, \alpha)\{\bar{a} \parallel a \parallel \beta(r)\}$, then we have

$$\frac{\vdots}{\Gamma \vdash P_2 : T_2, E_2}$$

where $T_2 = \bar{a} \parallel a$ and $E_2 = \varphi\langle\alpha.\beta\rangle$. Notice that the type $T_2$of P_2 is the same type T of P and the side effect E_2 is "sub-effect" of E. Notice that the erroneous event β is captured in both cases.

Example 5.3.27. Consider another scenario of 5.3.24, where $P \overset{\overline{\beta(r)}}{\rightarrow} P_3 = (r, \varphi, \alpha)\{\mathbf{0}\} \parallel \bar{a} \parallel a \parallel \alpha(r)$, then we have

$$\frac{\vdots}{\Gamma \vdash P_3 : T_3, E_3}$$

where $T_3 = \bar{a} \parallel a \parallel \alpha(r)$ and $E_3 = \xi$. In this case, the side effect E_3 is empty since the process P_3 does not use r anymore. Furthermore, T_3 contains the dangling action $\alpha(r)$ (it is indicated by P_3), while the original T does not. This reflects the fact that only communication-based abstractions in process types are preserved by evaluation, i.e. guaranteeing $T_{\uparrow\_} \equiv T_{3\uparrow\_}$.

Lemma 5.3.28. *Given $\Gamma \vdash P : T, E$, if $P \overset{\alpha?r}{\rightarrow} P'$ then $\Gamma \vdash P' : T', E'$ and $T \overset{\alpha(r)}{\rightarrow} T''$ for some E', T', T'' such that $T' \preceq T''$. Moreover if E is satisfied then so is E'.*

Proof. The proof proceeds by induction on the depth of the derivation of $P \overset{\alpha?r}{\rightarrow} P'$, where the last rule applied is distinguished. Most of cases are obvious. Here we only consider one of the most interesting case.

- the base case of [Act]: suppose $P = \alpha(r).P' \overset{\alpha?r}{\rightarrow} P'$ and $\Gamma \vdash P, E$.

  By applying Lemma 5.3.18 to $\Gamma \vdash P, E$, there exist S and E' such that

  $$\Gamma \vdash_N P' : S, E'$$
  $$\Gamma \vdash_N P : \alpha(R).S, E' \text{ and } T \equiv \alpha(R).S, r \in R$$

  where $R = rn(\Phi)$ and $E \equiv E'$. Let $T' \triangleq S$ and $T'' \triangleq S$. We have $T \equiv \alpha(R).S \overset{\alpha(R)}{\rightarrow} S = T''$, $r \in R$ and $T' = T''$, hence $T' \preceq T''$. Moreover, $E \equiv E'$ implies that whenever E is satisfied then E' is also satisfied. The required result follows.

- The induction cases of [Par], [Choice], [Cong], [Res], [Local$_1$], [Local$_2$] are straightforward.

□

Example 5.3.29. Let us continue to discuss the scenario of Example 5.3.27. Assume $P_2 = (r, \varphi, \alpha)\{\mathbf{0}\} \parallel \bar{a} \parallel a \parallel \alpha(r) \overset{\alpha?r}{\to} (r, \varphi, \alpha)\{\mathbf{0}\} \parallel \bar{a} \parallel a = P_3$, then we have

$$\frac{\vdots}{\Gamma \vdash P_3 : T_3, E_3},$$

where $T_3 = \bar{a} \parallel a$ and $E_3 = \xi$. Notice that T_2 can perform α as well, i.e. $T_2 \overset{\alpha(r)}{\to} \bar{a} \parallel a = T_2'$ Moreover, $T_2' = T_3$.

Now, the subject reduction theorem can be established.

Theorem 5.3.30. ***(subject reduction)****. Given a judgement $\Gamma \vdash P : T, E$ and $P \overset{l}{\to} P'$. There exist T', E', T'', R such that*

- *if $l = \langle a \rangle$, then $\Gamma \vdash P' : T', E'$ and $T \overset{\langle a \rangle}{\to} T''$.*
- *if $l \in \{\tau, \alpha(r), \overline{\alpha(r)}\}$, then $\Gamma \vdash P' : T', E'$ and $T \overset{\tau}{\to}^* T''$.*
- *if $l = \alpha?r$, then $\Gamma \vdash P' : T', E'$ and $T \overset{\alpha(R)}{\to} T''$, $r \in R$.*

Moreover, $T'{\uparrow_{-}} \preceq T''{\uparrow_{-}}$ and if E is satisfied then so is E'.

Proof. It is straightforward from the subject reduction lemmas. □

Remark 5.3.31. The subject reduction theorem shows that types simulate their processes. In addition, the statement on resource constraints implies that the well-typedness is preserved by evaluation, hence the theorem indeed preserves the correctness of resource usages of well-typed processes.

5.4 Type Inference Algorithm

In this section, we present a type inference algorithm to check whether, given a well-formed context Γ, a process P is well-typed under Γ or not, i.e. there exist T, E such that $\Gamma \vdash P : T, E$.

Before describing the algorithm, we introduce some auxiliary definitions. We use σ to denote a finite map from type variables to closed types. $T\sigma$ denotes the type in result of substituting a type variable X in T with $\sigma(X)$ (if it is defined). Formally, it is defined as follows:

$$\begin{array}{llll}
(X)\sigma & = \sigma(X) & (!a(t).T)\sigma & = (a(t).T)\sigma \\
(\alpha(R).T)\sigma & = \alpha(R).T\sigma & (a(t).T)\sigma & = a(t).T\sigma \\
(\bar{a}.T)\sigma & = \bar{a}.T\sigma & (\tau.T)\sigma & = \tau.T\sigma \\
(T+T')\sigma & = T\sigma + T'\sigma & (T \parallel T')\sigma & = T\sigma \parallel T'\sigma
\end{array}$$

$\sigma \circ \{T/X\}$ is a map σ' such that $\forall Y \in dom(\sigma')$, $\sigma'(Y) = T$ if $Y = X$, otherwise $\sigma'(Y) = \sigma(Y)$. Similarly, for the resource constraint E, $E\sigma$ is defined as expected.

Following the style of [4], the algorithm rests on the definition of a pair of functions, *typeinf* and *solve*, that define a type inference algorithm, in the following sense:

1. If $typeinf(P, \Gamma) = (T, C, R)$ and $solve(\Gamma, C) = \sigma$, then $\Gamma \vdash P : T\sigma$.

2. If $\Gamma \vdash P : T, E$, then there are T', C, E' and σ such that $typeinf(P, \Gamma) = (T', C, R)$ and $solve(C) = \sigma$, $T'\sigma \equiv T$ and $E \equiv \mathsf{E}'\sigma$.

Note that all the bound names in Γ and P are assumed to be distinct from one another and from free names.

The function *typeinf* is defined in Fig. 5.6. In general, the outcome of *typeinf* is a triple composed by: i) an open process type T describing communication behaviour of P; ii) a set of type constraints C, which needs to be solved in order to obtain a closed type from T; iii) a set of resource constraints E whose satisfaction guarantees the correct usage of resources occurring in P. Type constraints have only the form $\langle T = X \parallel Y, y\#X, T_{\uparrow\_} = T_a\rangle$ (ranged over by TC).

To solve a set of type constraints like the above, we make use of the second function *solve*, which in turn use an auxiliary function *split*:

$$split(\langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a\rangle) = \begin{cases} (T_1, T_2) & \text{if } \exists T_1, T_2 \text{ s.t. } T_1 \parallel T_2 \equiv T, \\ & y\#T_1 \text{ and } T_{2\uparrow\_} = T_a\rangle \\ undefined & \text{otherwise.} \end{cases}$$

Finally, the function *solve* is defined as follows:

$$\begin{cases} solve(\emptyset) & = \epsilon \\ solve(TC \cup C) & \text{if } \begin{cases} solve(C\{T_1/X, T_2/Y\}) \circ \{T_1/X, T_2/Y\} & \text{if } split(TC) = (T_1, T_2) \\ undefined & \text{otherwise,} \end{cases} \end{cases}$$

where $TC = \langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a\rangle$.

Elementary reasoning over structural congruence on the languages of types shows that *split* is computable, hence *solve* is computable. More precisely, we define the

```
$typeinf(\mathbf{0}, \Gamma) =$          $(\varnothing, \mathbf{0}, \varnothing)$
$typeinf(x(w).P, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(P, \Gamma)$
                             $\Gamma \vdash a : (w : t)T_a$
                             $X\ is\ fresh$
                         $\mathbf{in}\ (a.(X \parallel Y_{\downarrow\_}), \langle T = X \parallel Y, w\#X, Y_{\uparrow\_} = T_a\rangle \cup C, E)$
$typeinf(\bar{x}\langle b\rangle.P, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(P, \Gamma)$
                             $\Gamma \vdash b : t$
                             $\Gamma \vdash a : (y : t)T_a$
                         $\mathbf{in}\ (T \parallel T_a\{b/y\}, C, E)$
$typeinf(\bar{x}\langle r\rangle.P, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(P, \Gamma)$
                             $\Gamma \vdash r : res(\Phi)$
                             $\Gamma \vdash a : (s : res(\Phi_s)T_a$
                             $\Gamma\Phi \subseteq \Phi_s$
                         $\mathbf{in}\ (T \parallel T_a\{r/s\}, C, E)$
$typeinf(P_1 \parallel P_2, \Gamma) =$
                         $\mathbf{let}\ (T_1, C_1, E_1) = typeinf(P_1, \Gamma)$
                             $(T_2, C_2, E_2) = typeinf(P_2, \Gamma)$
                         $\mathbf{in}\ (T_1 \parallel T_2), C_1 \cup C_2, E_1 \vee E_2)$
$typeinf(\nu x.P, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(P, (\Gamma, z : t))$
                         $\mathbf{in}\ ((\nu z)T, C, E)$
$typeinf(!a(y).P, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(a(y).P, \Gamma)$
                         $\mathbf{in}\ (!T, C, !_{\#}E)$
$typeinf(\alpha(s).P, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(P, \Gamma)$
                             $\Gamma \vdash s : res(\Phi)$ and $R = rn(\Phi)$
                         $\mathbf{in}\ (\alpha(R).T, C, E)$
$typeinf((r, \varphi, \eta)\{P\}, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(P, \Gamma)$
                             $\Gamma \vdash r : res(\Phi)$
                             $\varphi \in \Phi$
                         $\mathbf{in}\ (T_{\uparrow r}, C, E \vee \varphi\langle T_{\downarrow r}\rangle)$
$typeinf(req(s)\{P\}, \Gamma) =$
                         $\mathbf{let}\ (T, C, E) = typeinf(P, \Gamma)$
                             $\Gamma \vdash s : res(\Phi)$ and $R = rn(\Phi)$
                         $\mathbf{in}\ (T_{\uparrow R}, C, E \vee \bigvee_{\langle r, \varphi\rangle \in \Phi} \langle\varphi\rangle T_{\downarrow r})$
```

Figure 5.6: Type inference algorithm.

set $\mathsf{subs}()$ of sub-processes of a closed type T as follows:

$$\begin{array}{ll}
\mathsf{subs}(a(t).T) & = \{a(t).T\} \cup \mathsf{subs}(T) \\
\mathsf{subs}(\bar{a}.T) & = \{\bar{a}.T\} \cup \mathsf{subs}(T) \\
\mathsf{subs}(\tau.T) & = \{\tau.T\} \cup \mathsf{subs}(T) \\
\mathsf{subs}(T + T') & = \{T + T'\} \cup \mathsf{subs}(T) \cup \mathsf{subs}(T') \\
\mathsf{subs}(T \parallel T') & = \{T \parallel T'\} \cup \mathsf{subs}(T) \cup \mathsf{subs}(T') \\
\mathsf{subs}((\nu a : t)T) & = \{(\nu a : t)T\} \cup \mathsf{subs}(T) \\
\mathsf{subs}(T \parallel T') & = \{T \parallel T'\} \cup \mathsf{subs}(T) \cup \mathsf{subs}(T')
\end{array}$$

Notice that the set of sub-processes of a closed type is finite. Therefore, by brute-force reasoning, we can find all possible splits of the given closed type.

Note that *split* can yield to several outcomes. The following example illustrates the possible outcomes of the function *split*:

$$\begin{array}{l}
T = a \parallel \alpha(r).b.\tau \parallel \alpha(r).c \parallel \beta(r).c \\
\mathit{split}(\langle T = X \parallel Y, b\#X, Y_{\uparrow\mathcal{R}} = \tau.b.\tau \parallel \tau.c\rangle) = \\
(a \parallel \alpha(r).c, \alpha(r).b.\tau \parallel \beta(r).c) \textbf{ or } (a \parallel \beta(r).c, \alpha(r).b.\tau \parallel \alpha(r).c)
\end{array}$$

However, the resource information is maintained during the split because of the following lemma.

Lemma 5.4.1. *Given $C = \langle T = X \parallel Y, y\#X, Y_{\uparrow\mathcal{R}} = T_a\rangle$. If $\mathit{split}(C) = (T_1, T_2)$, then $T_{\downarrow_{-}} \equiv (T_1 \parallel T_{2\downarrow_{-}})_{\downarrow_{-}}$. Moreover, if $\mathit{split}(C) = (T_1', T_2')$ with $T_1 \neq T_1', T_2 \neq T_2'$, then $(T_1 \parallel T_{2\downarrow_{-}})_{\downarrow_{-}} \equiv (T_1' \parallel T'_{2\downarrow_{-}})_{\downarrow_{-}}$.*

Proof. Be definition of the hiding operator $\downarrow_R$, we have

$$\begin{array}{ll}
T_{\downarrow_{-}} & \equiv (T_1 \parallel T_2)_{\downarrow_{-}} \\
& = T_{1\downarrow_{-}} \parallel T_{2\downarrow_{-}} \\
& = T_{1\downarrow_{-}} \parallel (T_{2\downarrow_{-}})_{\downarrow_{-}} \\
& = (T_1 \parallel T_{2\downarrow_{-}})_{\downarrow_{-}}
\end{array}$$

This establishes the required result. □

The resulted process type is unique up to structural congruence and the hiding-resources operator as stated in the following lemma.

Lemma 5.4.2. *Given $C = \langle T = X \parallel Y, y\#X, Y_{\uparrow_{-}} = T_a\rangle$. If $\mathit{split}(C) = (T_1, T_2)$, then the split is unique w.r.t structural congruence and the hiding-resources operator, i.e. whenever $\mathit{split}(C) = (T_1', T_2')$, $T_{1\uparrow_{-}} \equiv T'_{1\uparrow_{-}}$.*

Proof. Suppose $\mathit{split}(C) = (T_1, T_2)$ and $\mathit{split}(C) = (T_1', T_2')$. Then, we have $T_{2\uparrow_{-}} = T'_{2\uparrow_{-}} = T_a$ and $T_{\uparrow_{-}} = T_{1\uparrow_{-}} \parallel T_{2\uparrow_{-}} = T'_{1\uparrow_{-}} \parallel T'_{2\uparrow_{-}}$. This implies that $T_{1\uparrow_{-}} = T'_{1\uparrow_{-}}$. □

In summary, resource-based and communication-based behaviours do not depend on the choice of *split* during the resolution of the the set of type constraints generated by the type inference algorithm.

Notice that by induction on the derivation of $\mathit{typeinf}(P, \Gamma) = (E, C, R)$, one can prove that either C is empty or there is at least one pair of the form $\langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a \rangle$, where T is a closed process type. We call such constraints *ground*. This property is preserved after the resolution of each ground constraint, and the application of the resulting substitution.

Lemma 5.4.3. *There is at least one closed type process during the resolution of a set of type constraints with the application of the resulting substitution.*

Proof. Induction on the derivation of $\mathit{typeinf}(P, \Gamma) = (E, C, R)$.

- base case of $\mathit{typeinf}(\mathbf{0}, \Gamma) = (\varnothing, \mathbf{0}, \varnothing)$: it is obvious.
- the case of $\mathit{typeinf}(x(y).P, \Gamma) = a.(X \parallel Y_{\downarrow\_}), \langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a \rangle \cup C, R)$ with $(T, C, R) = \mathit{typeinf}(P), \Gamma \vdash a : (y : t)T_a$ and fresh X, Y: We have consider resolution of $\langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a \rangle \cup C$.

  First, by applying induction hypothesis to $(T, C, R) = \mathit{typeinf}(P)$, we have two cases: either C contains at least one constraint $\langle T' = X \parallel Y, y\#X, Y_{\uparrow\_} = T'_a \rangle$, where T' is closed, or C is empty. In the former case, we can apply induction hypothesis to $(T, C, R) = \mathit{typeinf}(P)$. In the latter case, since we that that if C is satisfied with the resulted substitution σ then $T\sigma$ is a closed type process since the only remaining variables are X, Y and X, Y are not type variables in T. Hence, the required property is obtained from both cases.
- The proof of the remaining cases is easy.

□

It remains to prove that the set C of resource constraints of each policy is satisfied. We proceed in two steps. First, types can be easily translated into BPP processes except for the replication operator. The translation is given as follows:

$$\begin{array}{llll} tran(\mathbf{0}) & = \mathbf{0} & tran(\mu.T) & = \mu.tran(T) \\ tran(T + T') & = tran(T) + tran(T') & tran(T \parallel T') & = tran(T) \parallel tran(T') \\ tran(!\alpha.T) & = X, \text{ where } X \overset{def}{=} \alpha(trans(T) \parallel X) & & \end{array}$$

Lemma 5.4.4 (Soundness of the translation)**.** *Given a type $T \in \mathcal{T}_0$ and a process $P = tran(T) \in \mathcal{P}_{bpp}$. We have $Traces(T) = Traces(P)$.*

Proof. It straightforward by induction of structure of T. □

Second, we apply the technique in [54] to model check BPP processes against LTL formulas.

Remark 5.4.5. As mentioned before in Remark 5.3.3, in the case of interpreting $!_{\#}$ as an infinite concatenation of resource usages, effects would be translated into PA processes, which have the undecidable problem of model checking against LTL formulas (see [79] for details).

Now we are ready to state the type inference theorem.

Theorem 5.4.6. *(**type inference**) Given Γ and P, it is decidable whether P is well-typed under Γ.*

Recall that correctness of the type inference algorithm is based on the two following facts.

- given a process P, a well-formed context Γ, if $\mathit{typeinf}(P,\Gamma) = (T,C,R)$ and $\mathit{solve}(\Gamma,C) = \sigma$, then $\Gamma \vdash P : T\sigma, R\sigma$ is a type derivation. In other words, the outcome of the algorithm forms a correct type derivation.
- if $\Gamma \vdash P : T, E$, the outcome of the inference algorithm complies with T and E, that is, $\mathit{typeinf}(P,\Gamma) = (T',C,R)$ and $\mathit{solve}(C) = \sigma$ implies that $T'\sigma \equiv T$ and $R\sigma \equiv E$.

Formally, the facts are proved by the following theorems.

Lemma 5.4.7. *(**correctness**) If $\mathit{typeinf}(P,\Gamma) = (T,C,R)$ and $\mathit{solve}(\Gamma,C) = \sigma$, then $\Gamma \vdash P : T\sigma, R\sigma$.*

Proof. The proof is straightforward by induction on the last rule applied for deducing $\mathit{typeinf}(P,\Gamma) = (E,C,R)$.

- base case of $\mathit{typeinf}(\mathbf{0},\Gamma) = (\varnothing,\mathbf{0},\varnothing)$: it is obvious.
- the case of $\mathit{typeinf}(a(y).P,\Gamma) = a.(X \parallel Y_{\downarrow\_}), \langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a\rangle \cup C, R)$ with assumptions $(T,C,R) = \mathit{typeinf}(P)$, $\Gamma \vdash a : (y:t)T_a$ and fresh X, Y and $\langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a\rangle \cup C$ is solved, so is C. Since X, Y are fresh with respect to C, we have $\mathit{solve}(C) = \sigma'$ s.t. σ' and σ are the same substitution on $\mathit{dom}(\sigma) \setminus \{X,Y\}$ and X, Y are not type variables in T and E. By applying induction hypothesis to $(T,C,R) = \mathit{typeinf}(P)$, we get $\Gamma \vdash P : T\sigma', E\sigma'$. Since $\langle T = X \parallel Y, y\#X, Y_{\uparrow\_} = T_a\rangle$ is satisfied by σ, $T\sigma = T\sigma' = \sigma(X) \parallel \sigma(Y)$ s.t. $y\#\sigma(X)$ and $\sigma(Y) = T_{a_{\uparrow\_}}$. By applying the rule [ts_input] to $\Gamma \vdash P : T\sigma', E\sigma'$, we get $\Gamma \vdash a(y)P : a.(\sigma(X) \parallel \sigma(Y)_{\downarrow\_}), E\sigma$. This implies the required result $\Gamma \vdash a(y)P : a.(X \parallel Y_{\downarrow\_})\sigma, E\sigma$.
- other cases: it is straightforward.

□

Lemma 5.4.8. ***(completeness)*** *If $\Gamma \vdash P : T, E$, then there are T', C, R and σ such that $typeinf(P, \Gamma) = (T', C, R)$. $solve(C) = \sigma$, $T'\sigma \equiv T$ and $R\sigma \equiv E$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash P : T, E$.

- base case of the empty rule: it is obvious.
- the case of the rule [ts_input]: we are given $\Gamma \vdash a(y).P : T, E$. By the Lemma 5.3.14, we get $\Gamma \vdash_N a(y).P : S, E$ s.t. $S \equiv T$ and by Lemma 5.3.18, $\Gamma, y : t \vdash_N P : T_1 \parallel T_2$, $\Gamma \vdash a : (y : t)T_a$, $y\#T_1$ and $T_{2\uparrow\_} = T_a$ s.t. $S = a.(T_1 \parallel T_{2\downarrow\_})$. By applying induction hypothesis to $\Gamma, y : t \vdash_N P : T_1 \parallel T_2$, there exist T', C, R s.t. $typeinf(P, \Gamma) = (T', C, R)$, $solve(C) = \sigma$, $T'\sigma \equiv T_1 \parallel T_2$ and $R\sigma \equiv E$. Now, we have $typeinf(a(y).P, \Gamma) = (a.(X \parallel Y_{\downarrow\_}), \langle T' \equiv X \parallel Y, y\#X, Y_{\uparrow\_} = T_a\rangle \cup C, R)$ and let $\sigma' = \sigma[T_1/X][T_2/Y]$. This implies that $solve(\langle T' \equiv X \parallel Y, y\#X, Y_{\uparrow\_} = T_a\rangle \cup C) = \sigma'$, $a.(X \parallel Y_{\downarrow\_})\sigma' = a.(T_1 \parallel T_{2\downarrow\_} \equiv T$ and $R\sigma' = R\sigma \equiv E$. This concludes the proof.
- other cases: it is straightforward.

□

For safety properties, as in the policy compliance theorem in Chapter 4, the below theorem ensures the correctness of resource usages, i.e no faulty traces occur if a process is well-typed.

Theorem 5.4.9. *Given a resource r declared with a safety policy φ in P. If P is well-typed, then P complies with φ for r.*

Proof. By the way of contradiction, suppose that P does not comply with φ, e.g. there exist P', P'' such that $P \xrightarrow{\mu}{}^{*} P' \xrightarrow{\overline{\alpha(r)}} P''$, where $\alpha(r)$ is the first violation action occurred in the sequence of transitions. The proof of the theorem amounts to checking that E', given by $\Gamma \vdash P' : T', E'$, is not satisfied, which implies a contradiction due to the subject reduction lemmas. Suppose that $\Gamma \vdash P, E$. By Subject Reduction theorem, we have that $\Gamma \vdash P, E$ implies $\Gamma \vdash P', E'$.

For this we proceed by cases on P' making use of the induction hypothesis. Many cases are straightforward and here we only consider the most interesting cases.

The case of $P' \equiv (r, \varphi, \eta)\{\alpha(r).Q\}$, $\alpha(r).Q \xrightarrow{\alpha?r} Q$ and $\eta.\alpha \not\models \varphi$: it follows that E' contains a resource constraint of the form $\varphi\langle\eta.\alpha.T\rangle$. The safety property of φ implies that $\eta.\alpha.T \not\models \varphi$, that is E' is not satisfied, which is contradiction with the well-typedness of P'. □

5.5 Related Works and Discussions

Behavioural types. Several approaches have exploited type systems to abstract over resource behaviour. The approaches closer to our development are presented in [66, 4, 63]. The work in [63] introduced a generic type system for the π-calculus. By choosing sub-typing relation and a "consistency condition" of types, one can obtain a variety of type systems, such as those ensuring deadlock-freedom [69] or type-based information analysis [68], from the generic type system. Moreover, the type soundness property of the generic type system ensures the soundness of a certain class of its instances.

The limits of behavioural type system, as instances of the generic type system based on the use of simulation as a sub-typing relations, may lead to undecidable type checking, pointed in [72]. A second point is that a "sub-divide" law, $T \equiv T_{\downarrow x} \parallel T_{\uparrow x}$ (i.e. T is split into two parts: T_1 depends on x and T_2 does not), is essentially used in the sub-typing relation in the generic type system. In fact, this law ignores dependencies among names in types, therefore breaks spatial correspondence between processes and types. These issues has been addressed by the work reported in [4] by reducing the flexibility of the generic type system and gaining decidable results for classes of interesting properties. By defining the sub-typing relation as structural congruence and the absence of union types, the *spatial* structure of processes are maintained in process types. This allows the verification of spatial properties of processes through types, using a fragment of spatial logics [40], called *Shallow Logics*. The main difference between the work in [66] and ours lie at the design choice of the model of resources.

The design choice of our approach is motivated by considering type abstractions that maintains the spatial model checking, as the one in [4], and for resource abstraction that enables verification of resource usages. In result, we adopt instead the type system in [4], where the abstraction of resource behaviour is extracted in the form of side effects. The novelty of our approach comes from the separation of resource behaviour from process behaviour through the type and effect system. Our treatment of resources requires a *symmetric* treatment for input/output on resource behaviour. In [66, 4, 63], rules of input/output are asymmetric in the sense that the sender gets information about continuations of receivers. On the contrary, the resource information should be maintained on both sides (receivers and senders). It can be thought as *symmetric* treatment of input and output processes on resource behaviour. The point is that a spatial split $T \equiv T_1 \parallel T_2$, as described in the rule [ts_input_res], also splits "resource behaviour". To handle this situation, we introduce the "resource splitting" operators $\downarrow$ and $\uparrow$ on resource abstraction of types (reminiscent of the law "sub-divide") to properly keep track of resource behaviour during type checking.

The type system in [66] can be considered as derivation of the one in [63] to deal with resource usages, where the sub-typing relation is defined as structural preorder.

More precisely, in this approach, resource usages are obtained by considering the abstract behaviour of the corresponding private names, i.e. private names extended with a set of traces to define the control over resource accesses. Types in [66] can be automatically inferred through a type inference algorithm with verification of safety properties of resource usages. The main difference between this type system and ours lies at the design choice of model of resources. Consequently, we need to introduced primitives to deal with resource management. As a consequence we need a more complicated extension of the π-calculus than the one in [66]. The shared semantics of private names results in concurrent resource accesses by different processes. In a result, a further approximation to reduce dependencies is required in order to verify safety properties of resource usages. In our design, the explicit model of resource results in gaining precision of tracking resource usages of "individual" processes, i.e. resource requesters. The pay-off comes from that fact that we are able to directly model checking resource behaviour against linear temporal properties, whereas an extra step of over-approximation of resource usages is required in [66].

It is worthwhile to mention works on session types or conversation types. Our type system is considered orthogonal to these works since the different issues are addressed. In [42, 41], the focus is on properties for disciplining interactions between parties. In other words, "business logic" issues are tackled by these works. On the contrary, our main concern is on the correctness of resource usages. For example, resource management in a data center requires proper configurations and settings of virtual machines and services running on them. A misconfiguration could lead to a huge impact on the whole infrastructure, which is usually not a primary concern of applications.

Guaranteed resource properties. Verification of resource usages requires a great effort as the complexity of modern distributed systems increases. A number of methods has been proposed to check properties of distributed resources.

In [47], a proof system is presented to ensure the correctness of resource usages, however it is not clear how to model check it. Type-based analysis has proved its usefulness in many works [66, 70, 104, 108]. In [108], a *unique* type capturing safe reconfigurations (*strong update*) over channels has been introduced. The term *unique* means that only a process owning a channel can allocate or deallocate it. This also guarantees the safety of modifying a channel having a unique type, that is, a strong update. The soundness of the type system ensures that evaluation of a well-typed process never get an error. Similarly, in [70], linear usages of channel input/outputs are verified by the type system.

The language of resource behaviours in our approach is a fragment of CCS without hiding and renaming operators, and therefore is less expressive than those in [66]. However, the pay-off comes from the fact that we do not need an extra step to further approximate them into a form suitable to be model checked as it is the case of the techniques of [66]. Verification of resources is based on a translation of side effect into basic parallel processes. Decidability results ensure the effectiveness of BPP model checking [54] against LTL. Liveness properties can be checked by combining the resource type system in [66] and existing type systems [67, 68, 71]. However, it is not clear how such combination works.

As mentioned in Section 4.4, privacy of resources could potentially give a closer view of resource usages. Recall that this view could potentially separate two important aspects of resource usages: i) an internal view, which focuses on states of resources; ii) an external view, which focuses on dependencies among external processes that request resources. Side effects constructed in the our type system reflect the first view. The second view can be seen in the restriction $(\nu r : t)$, where t could be used to specify properties related to external processes. To support this feature, however, we need to carefully consider the treatment of input/output of private resource names.

Chapter 6

Conclusions

In this dissertation, we studied computing models and techniques to deal with the management of distributed resources. Since resource awareness is a central notion, we modelled resources as stateful and independent entities which are available on demand.

We introduced a novel declarative model for Cloud Computing in Chapter 3. The model takes the form of a concurrent λ-calculus enriched with primitive constructs to manage the assembling of services in the cloud, (asynchronous) service invocation, security policies and their enforcement mechanisms. Abstract bisimulation semantics provides the formal basis for compositional reasoning on the behaviour of cloud systems. Our use of the λ-calculus as foundational basis for Cloud Computing is a refinement of the $\lambda^{\square}$ calculus [15, 12], that handles the service business logic in the form of service orchestration, and of the version of $\lambda^{\square}$-calculus presented [16] that focussed on resource usage analysis for functional languages. In our model, we do not address the issues related to service orchestration, instead, we concentrate on linkage among cloud services and resources. Indeed, our main contribution here consists in the management of cloud services via asynchronous invocation and the development of the bisimulation semantics.

In Chapters 4 and 5, we introduced the G-Local π-calculus, an extension of the π-calculus, to managing distributed resources. The model combines the name-passing of the π-calculus with the publish-subscribe paradigm to cope with resource-awareness. We obtain a name passing process calculus with primitives for acquiring and releasing stateful resources. Our research program has provided the formal mechanisms underlying the definition of a resource-aware programming model. The explicit notion of resource gets benefits in several places. First, we argue that it provides an easy way to specify and design properties of resources. Second, resources with embedded structures are stateful and independent entities in the style of publish-subscribe paradigm. Resources are subject to be published, available and not under the control of the applications that request them. Third, we believe that the model of resources can be easily tweaked to be used with other formalisms of usage policies, for instance, constraints on resources using c-semirings [25].

In terms of reasoning mechanisms, we developed two techniques, namely Control Flow Analysis (CFA) and Type and Effect Systems. CFA mainly focuses on reachability properties related to resource usages. It would be interesting to exploit CFA techniques to develop methodologies to instrument the code in order to avoid bad accesses to resources.

The idea of type and effect system is that types are equipped with resource-access actions $\alpha(R)$, where R is a finite set of resource names over which α possibly acts. Basically, types abstract two kinds of behavioural information: communication-based and resource-based. The former describes the abstract behaviour of processes on channels, i.e. the dependencies and interactions among channels, while the latter describes the abstract resource behaviour of processes. The novelty of our approach comes from the separation of resource behaviour from process behaviour through the type and effect system. Resource behaviours are constructively extracted as side effects during type checking. Side effects are basically BPP processes, which lie at the border of many decidability problems in model checking techniques [78]. Thus, regular linear time properties of resource usages, which can be expressed by LTL formulas, can be considered as the *most* powerful properties, that still enjoy decidable results. Also it would be interesting to apply the typing techniques (behavioural types) introduced in [16] to capture a notion of resource contract.

Bibliography

[1] Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the Network and Distributed System Security Symposium (NDSS'03). The Internet Society (2003)

[2] Abowd, G.D., Mynatt, E.D.: Charting past, present, and future research in ubiquitous computing. ACM Transactions on Computer-Human Interaction (TOCHI'00) 7, 29–58 (March 2000)

[3] Abramsky, S., Ong, C.H.L.: Full abstraction in the lazy lambda calculus. Information and Computation 105(2), 159–267 (1993)

[4] Acciai, L., Boreale, M.: Spatial and behavioral types in the pi-calculus. Information and Computation 208, 1118–1153 (October 2010)

[5] Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-oriented programming. In: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09). pp. 1015–1022 (2009)

[6] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009)

[7] Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. The International Journal of Computer and Telecommunications Networking 54(15), 2787–2805 (Oct 2010)

[8] Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)

[9] Banerjee, A., Naumann, D.A.: History-based access control and secure information flow. In: Proceedings of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'05). LNCS 3362, Springer (2005)

[10] Bartoletti, M.: Usage automata. In: Proceedings of Foundations and Applications of Security Analysis, Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09). pp. 32–47. LNCS 4423, Springer (2009)

[11] Bartoletti, M., Degano, P., Ferrari, G.: History-based access control with local policies. In: Proceedings of 8th Foundations of Software Science and Computational Structures Conference (FoSSaCS'05). pp. 316–332. LNCS 3441, Springer (2005)

[12] Bartoletti, M., Degano, P., Ferrari, G.: Planning and verifying service composition. Journal of Computer Security 17 (5) (2009)

[13] Bartoletti, M., Degano, P., Ferrari, G., Zunino, R.: Secure service orchestration. In: Proceedings of Foundations of Security Analysis and Design IV (FOSAD'07). LNCS 4667, Springer (2007)

[14] Bartoletti, M., Degano, P., Ferrari, G., Zunino, R.: Types and effects for resource usage analysis. In: Proceedings of 10th Foundations of Software Science and Computational Structures Conference (FoSSaCS'07). pp. 32–47. LNCS 4423, Springer (2007)

[15] Bartoletti, M., Degano, P., Ferrari, G., Zunino, R.: Semantics-based design for secure web services. IEEE Transactions on Software Engineering (TSE) 34(1), 33–49 (2008)

[16] Bartoletti, M., Degano, P., Ferrari, G., Zunino, R.: Local policies for resource usage analysis. ACM Transactions on Programming Languages and Systems (TOPLAS'09) 31(6) (2009)

[17] Bartoletti, M., Zunino, R.: A calculus of contracting processes. In: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science (LICS'10). pp. 332–341. IEEE Computer Society (2010)

[18] Bartoletti, M., Degano, P., Ferrari, G.L.: Types and effects for secure service orchestration. In: Proceedings of 19th Computer Security Foundations Workshop (CSFW'06) (2006)

[19] Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Model checking usage policies. In: Proceedings of Trustworthy Global Computing (TGC'08). pp. 19–35 (2008)

[20] Ben-Ari, M., Pnueli, A., Manna, Z.: The temporal logic of branching time. Acta Informatica 20, 207–226 (1983)

[21] Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)

[22] Bhargavan, K., Gordon, A., Narasamdya, I.: Service combinators for farming virtual machines. In: Proceedings of the 10th Coordination Models and Languages Conference (COORDINATION'08). LNCS, vol. 5052, pp. 33–49 (2008)

[23] Bhargavan, K., Andrew, D.G.: Getting operations logic right: Types, service-orientation, and static analysis. In: Proceedings of the workshop on "The Rise and Rise of the Declarative Datacentre". pp. 9–11. Microsoft Research (2008)

[24] Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. Journal of the ACM 44, 201–236 (1997)

[25] Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. Journal of the ACM 44(2), 201–236 (1997)

[26] Bodei, C., Degano, P., Nielson, F., Nielson, H.: Static analysis for the pi-calculus with applications to security. Information and Computation 168(1), 68–92 (2001)

[27] Bodei, C., Dinh, D., Ferrari, G.: Safer in the clouds. Tech. Rep. TR-10-15, Dipartimento di Informatica (2010)

[28] Bodei, C., Dinh, V., Ferrari, G.: Safer in the clouds (extended abstract). In: Proceedings of Interaction and Concurrency Experience (ICE'10). EPTCS, vol. 38, pp. 45–49 (2010)

[29] Bodei, C.: A control flow analysis for beta-binders with and without static compartments. Theoretical Computer Science 410(33-34), 3110–3127 (2009)

[30] Bodei, C., Dinh, V.D., Ferrari, G.L.: Predicting global usages of resources endowed with local policies. In: Proceedings of 10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'11). EPTCS, vol. 58, pp. 49–64 (2011)

[31] Bodei, C., Dinh, V.D., Ferrari, G.L.: Predicting global usages of resources endowed with local policies. Science of Computer Programming (SCP) (submitted) (2012)

[32] Bodei, C., Dinh, V.D., Ferrari, G.L.: A g-local π-calculus. In: Proceedings of Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'11) (2011), `http://places11.di.fc.ul.pt/proceedings.pdf/view`

[33] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: Scc: A service centered calculus. In: Web Services and Formal Methods. pp. 38–57 (2006)

[34] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM 31(3), 560–599 (1984)

[35] Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures (2000)

[36] Buscemi, M.G., Montanari, U.: Cc-pi: A constraint-based language for specifying service level agreements. In: Proceedings of 16th European Symposium on Programming (ESOP'07). LNCS, vol. 4421, pp. 18–32. Springer (2007)

[37] Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25, 599–616 (2009)

[38] Caires, L.: Spatial-behavioral types, distributed services, and resources. In: Proceedings of the 2nd international conference on Trustworthy global computing (TGC'06). pp. 98–115. Trustworthy Global Computing 2006, Springer-Verlag (2007)

[39] Caires, L.: Spatial-behavioral types for concurrency and resource control in distributed systems. Theoretical Computer Science 402(2-3), 120–141 (2008)

[40] Caires, L., Cardelli, L.: A spatial logic for concurrency (part i). Information and Computation 186(2), 194–235 (2003)

[41] Caires, L., Vieira, H.T.: Conversation types. Theoretical Computer Science 411(51-52), 4399–4440 (2010)

[42] Carbone, M., Honda, K., Yoshida, N.: A calculus of global interaction based on session types. Electronic Notes in Theoretical Computer Science 171(3), 127–151 (2007)

[43] Cardelli, L.: Brane calculi. In: Computational Methods in Systems Biology (CMSB'04). pp. 257–278 (2004)

[44] Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)

[45] Christensen, S.: Decidability and decomposition in process algebras. Ph.D. Thesis, University of Edinburgh (1993)

[46] Church, A.: A formulation of the simple theory of types. The Journal of Symbolic Logic 5(2), 56–68 (1940)

[47] Collinson, M., Pym, D.: Algebra and logic for access control. Formal Aspects of Computing 22(3-4), 483–484 (2010)

[48] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the Fourth ACM Symposium on Principles of Programming Languages (POPL'77). pp. 238–252 (1977)

[49] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: ACM Symposium on Principles of Programming Languages (POPL'79). pp. 269–282 (1979)

[50] Dimoulas, C., Pucella, R., Felleisen, M.: Future contracts. In: Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'09). pp. 195–206 (2009)

[51] D.Talbot: How secure is cloud computing? Technology Review (Nov 2009), `http://www.technologyreview.com/computing/23951/`

[52] Edjlali, G., Acharya, A., Chaudhary, V.: History-based access control for mobile code. In: Proceedings of the 5th ACM conference on Computer and communications security (CCS'98). pp. 38–48. ACM, New York, NY, USA (1998)

[53] Esparza, J.: On the decidability of model checking for several μ-calculi and petri nets. In: Proceedings of the 19th International Colloquium on Trees in Algebra and Programming (TAP'94). pp. 115–129. Springer-Verlag, London, UK (1994)

[54] Esparza, J.: Decidability of model checking for infinite-state concurrent systems. Acta Informatica 34, 85–107 (1997)

[55] Ferrari, G., Moggi, E., Pugliese, R.: Guardians for ambient-based monitoring. In: F-WAN: Foundations of Wide Area Network Computing, number 66 in ENTCS. Elsevier Science. pp. 141–202. Elsevier (2002)

[56] Fong, P.W.L.: Access control by tracking shallow execution history. In: IEEE Symposium on Security and Privacy. pp. 43–55. IEEE Computer Society Press (2004)

[57] Fournet, C., Gordon, A.D.: Stack inspection: Theory and variants. ACM Transactions on Programming Languages and Systems (TOPLAS'03) 25, 360–399 (May 2003)

[58] Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS'85) 7(1), 80–112 (1985)

[59] Gordon, A.D.: V for virtual. Electronic Notes in Theoretical Computer Science 162, 177–181 (2006)

[60] Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A Calculus for Service Oriented Computing. In: 4th International Conference on Service Oriented Computing (ICSOC'06). Lecture Notes in Computer Science, vol. 4294, pp. 327–338. Springer (2006)

[61] Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Information and Computation 124(2), 103–112 (1996)

[62] Igarashi, A., Kobayashi, N.: Resource usage analysis. In: ACM Symposium on Principles of Programming Languages (POPL'02). pp. 331–342 (2002)

[63] Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. Theoretical Computer Science 311(1-3), 121–163 (2004)

[64] Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Transactions on Programming Languages and Systems (TOPLAS'05) 27, 264–313 (March 2005)

[65] Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Proceedings of the 8th Internatilonal Collogquium on Automata, Languages and Programming (ICALP'81). pp. 114–128 (1981)

[66] Kobayashi, N., Suenaga, K., Wischik, L.: Resource usage analysis for the pi-calculus. Logical Methods in Computer Science 2(3), 1–42 (2006)

[67] Kobayashi, N.: A type system for lock-free processes. Information and Computation 177(2), 122–159 (2002)

[68] Kobayashi, N.: Type-based information flow analysis for the pi-calculus. Acta Informatica 42(4-5), 291–347 (2005)

[69] Kobayashi, N.: A new type system for deadlock-free processes. In: Concurrency Theory (CONCUR'06). pp. 233–247 (2006)

[70] Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems (TOPLAS'99) 21, 914–947 (September 1999)

[71] Kobayashi, N., Saito, S., Sumii, E.: An implicitly-typed deadlock-free process calculus. In: Concurrency Theory (CONCUR'00). pp. 489–503 (2000)

[72] Kobayashi, N., Suto, T.: Undecidability of 2-label bpp equivalences and behavioral type systems for the pi -calculus. In: 34th International Colloquium: Automata, Languages and Programming (ICALP'07). pp. 740–751 (2007)

[73] Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM 47, 312–360 (March 2000)

[74] Landin, P.J.: The Mechanical Evaluation of Expressions. The Computer Journal 6(4), 308–320 (Jan 1964)

[75] Landin, P.J.: Correspondence between algol 60 and church's lambda-notation (part i). Communications of the ACM (CACM) 8(2), 89–101 (1965)

[76] Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: 16th European Symposium on Programming (ESOP'07). pp. 33–47 (2007)

[77] Lucas, P.: Formal definition of programming languages and systems. In: IFIP Congress (1). pp. 291–297 (1971)

[78] Mayr, R.: Decidability and complexity of model checking problems for infinite-state systems. Ph.D. Thesis, TU Mnchen (1997)

[79] Mayr, R.: Model checking pa-processes. In: Concurrency Theory (CONCUR'97). pp. 332–346 (1997)

[80] McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. pp. 21–28 (1962)

[81] Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer (1980)

[82] Milner, R.: Calculi for synchrony and asynchrony. Theoretical Computer Science 25, 267–310 (1983)

[83] Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)

[84] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i. Information and Computation 100(1), 1–40 (1992)

[85] Nguyen, N., Rathke, J.: Typed static analysis for concurrent, policy-based, resource access control (draft) (2006)

[86] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)

[87] Nielson, H.R., Nielson, F., Pilegaard, H.: Flow logic for process calculi (to appear). ACM Computing Surveys (2011)

[88] O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. The Bulletin of Symbolic Logic 5(2), 215–244 (1999)

[89] Oracle-Corproration: The try-with-resources statement. http://docs.oracle.com/javase/7/docs/technotes/guides/language/try-with-resources.html (2011)

[90] Pagani, M., Rocca, S.R.D.: Solvability in resource lambda-calculus. In: Foundations of Software Science and Computational Structures. pp. 358–373 (2010)

[91] Papazoglou, M.P., Traverso, P., Ricerca, I., Tecnologica, S.: Service-oriented computing: State of the art and research challenges. IEEE Computer 40, 2007 (2007)

[92] Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theoretical Computer Science 1(2), 125–159 (1975)

[93] Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming 60-61, 17–139 (2004)

[94] Pnueli, A.: The temporal logic of programs. In: Annual Symposium on Foundations of Computer Science. pp. 46–57 (1977)

[95] Priami, C., Quaglia, P.: Beta binders for biological interactions. In: Computational Methods in Systems Biology (CMSB'04). pp. 20–33 (2004)

[96] Reynolds, J.C.: Automatic computation of data set definitions. In: IFIP Congress (1). pp. 456–461 (1968), `ftp://ftp.cs.cmu.edu/user/jcr/autodataset.pdf`

[97] Sangiorgi, D., Walker, D.: Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA (2001)

[98] Sangiorgi, D.: The lazy lambda calculus in a concurrency scenario. Information and Computation 111(1), 120–153 (1994)

[99] Schneier, B.: Be careful when you come to put your trust in the clouds. `http://www.schneier.com/essay-274.html` (2009)

[100] Shivers, O.: Control-flow analysis in scheme. In: Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation. pp. 164–174 (1988)

[101] Skalka, C., Smith, S.F.: History effects and verification. In: Programming Languages and Systems: Second Asian Symposium (PLS'04). pp. 107–128. LNCS 3302, Springer (2004)

[102] Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. Journal of IEEE Transactions on Software Engineering 12(1), 157–171 (Jan 1986)

[103] Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, É.: First-class state change in plaid. In: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11). pp. 713–732 (2011)

[104] Teller, D.: Recovering resources in the pi-calculus. In: Proceedings of IFIP TCS 2004. pp. 605–618. Kluwer Academic Publishing (2004)

[105] Turon, A., Wand, M.: A resource analysis of the π-calculus. Electronic Notes in Theoretical Computer Science 276, 313–334 (September 2011)

[106] Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: LICS. pp. 332–344 (1986)

[107] Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service-oriented computation. In: 17th European Symposium on Programming (ESOP'08). pp. 269–283 (2008)

[108] de Vries, E., Francalanza, A., Hennessy, M.: Uniqueness typing for resource management in message-passing concurrency. In: Proceedings of First International Workshop on Linearity. EPTCS, vol. 22, pp. 26–37 (2009)

[109] Weiser, M.: The computer for the 21st century. SIGMOBILE Mobile Computing and Communications Review 3(3), 3–11 (Jul 1999)

[110] Youseff, L., Butrico, M., Silva, D.D.: Toward a unified ontology of cloud computing. In: Proceedings of Grid Computing Environments Workshop (GCE'08). pp. 1–10 (2008)