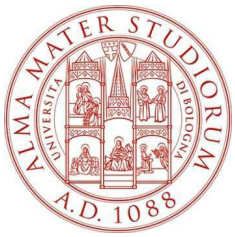

Alma Mater Studiorum - Università di Bologna

Department of Computer Science and Engineering

Type Systems for Distributed Programs: Components and Sessions

A Dissertation in

Computer Science by **Ornela Dardha**

Advisor

Davide Sangiorgi

Coordinator

Maurizio Gabbrielli

May 2014

Thesis Committee

Davide Sangiorgi (advisor)
Department of Computer Science and Engineering
University of Bologna
Bologna, Italy

Fabio Panzieri (member)
Department of Computer Science and Engineering
University of Bologna
Bologna, Italy

Gianluigi Zavattaro (member)
Department of Computer Science and Engineering
University of Bologna
Bologna, Italy

External Reviewers

Ilaria Castellani
INRIA
Sophia Antipolis Méditerranée
Sophia Antipolis, France

Vasco T. Vasconcelos
Department of Informatics
University of Lisbon
Lisbon, Portugal

Abstract

Modern software systems, in particular distributed ones, are everywhere around us and are at the basis of our everyday activities. Hence, guaranteeing their correctness, consistency and safety is of paramount importance. Their complexity makes the verification of such properties a very challenging task. It is natural to expect that these systems are reliable and above all usable.

i) In order to be reliable, compositional models of software systems need to account for consistent *dynamic reconfiguration*, i.e., changing at runtime the communication patterns of a program.

ii) In order to be useful, compositional models of software systems need to account for *interaction*, which can be seen as *communication* patterns among components which collaborate together to achieve a common task.

The aim of the Ph.D. was to develop powerful techniques based on formal methods for the verification of correctness, consistency and safety properties related to dynamic reconfiguration and communication in complex distributed systems. In particular, static analysis techniques based on types and type systems appeared to be an adequate methodology, considering their success in guaranteeing not only basic safety properties, but also more sophisticated ones like, deadlock or livelock freedom in a concurrent setting.

The main contributions of this dissertation are twofold.

i) On the *components* side: we design types and a type system for a concurrent object-oriented calculus to statically ensure consistency of dynamic reconfigurations related to modifications of communication patterns in a program during execution time.

ii) On the *communication* side: we study advanced safety properties related to communication in complex distributed systems like deadlock-freedom, livelock-freedom and progress.

Most importantly, we exploit an encoding of types and terms of a typical distributed language, session π- calculus, into the standard typed π- calculus, in order to understand the expressive power of concurrent calculi with structured communication primitives and how they stand with respect to the standard typed concurrent calculi, namely (variants) of typed π- calculus. Then, we show how to derive in the session π- calculus basic properties, like type safety or complex ones, like progress, by encoding.

Keywords: components, distributed systems, concurrency, π- calculus, session types, encoding, progress, lock-freedom.

Acknowledgments

I am grateful and thankful to my advisor Prof. Davide Sangiorgi, that during my Ph.D. has been of great support and guidance. Thank you for your advice, your presence when I needed your help and in particular for the freedom you gave me during the three years of my Ph.D.

I also want to thank the external reviewers of my Ph.D. dissertation: Ilaria Castellani and Vasco T. Vasconcelos for accepting to review my thesis, in the first place, and for their profound and careful work and their useful feedbacks.

A very special thank goes to Elena Giachino for her help and support, for her scientific, practical and life-related advice she gave me during these last years. I learned a lot from you!

I also want to thank Jorge A. Pérez, for being a very good friend and a very good "older academic brother". Thank you for your prompt response every time I needed your help. You always give me useful tips.

During my one-year-visit at IT University of Copenhagen, I had the pleasure to work with Marco Carbone and Fabrizio Montesi. Thank you for the very nice year at ITU, for being of great support and guidance, for making research a lot fun and for being such good friends.

I also want to thank Giovanni Bernardi for our long emails and vivid discussions on recursion and duality.

An extremely enormous hug goes to all my friends around the world, especially the ones in Rome, Bologna and Copenhagen. Thank you guys for the great time together, for being of inspiration and support and above all, for making me feel home whenever I am visiting you! In particular, a big thank to Elena, Tiziana,

Dora, Marco, Lorena and Juan.

Falenderoj familjen time babin, mamin dhe dy motrat e mia te mrekullueshme, per prezencën, durimin dhe dashurinë e tyre të pakushtëzuar. Ju dua shumë!

Ringrazio la mia (seconda) famiglia, mamma, papi e Titi: il tempo con voi non è mai abbastanza... Vi voglio un mondo di bene!

Contents

List of Figures

Introduction to the Dissertation

History's Worst Software Bugs

Report on Wired News in August 11, 2005

Computer bugs are still with us, and show no sign of going extinct. As the line between software and hardware blurs, coding errors are increasingly playing tricks on our daily lives. Bugs don't just inhabit our operating systems and applications – today they lurk within our cell phones and our pacemakers, our power plants and medical equipment, and in our cars. [...]

July 28, 1962 – Mariner I space probe. *A bug in the flight software for the Mariner 1 causes the rocket to divert from its intended path on launch. Mission control destroys the rocket over the Atlantic Ocean. The investigation into the accident discovers that a formula written on paper and pencil was improperly transcribed into computer code, causing the computer to miscalculate the rocket's trajectory.*

1985-1987 – Therac-25 medical accelerator. *A radiation therapy device malfunctions and delivers lethal radiation doses at several medical facilities. [...] Because of a subtle bug called a "race condition," a quick-fingered typist could accidentally configure the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position. At least five patients die; others are seriously injured.*

June 4, 1996 – Ariane 5 Flight 501. *Working code for the Ariane 4 rocket is reused in the Ariane 5, but the Ariane 5's faster engines trigger a bug in an*

arithmetic routine inside the rocket's flight computer. The error is in the code that converts a 64-bit floating-point number to a 16-bit signed integer. The faster engines cause the 64-bit numbers to be larger in the Ariane 5 than in the Ariane 4, triggering an overflow condition that results in the flight computer crashing [...] and causes the rocket to disintegrate 40 seconds after launch.

The previous text is taken from an article reported in the WIRED magazine in August 11, 2005 [91]. The events above are just a few taken from the long list of software bugs that have caused big havoc. The severity and impact of the bugs grows when dealing with safety critical applications and can result in huge amount of money and time loss or even worse, people lives loss.

This clearly shows the importance and the necessity of *correctness* and *safety* properties in software programs. In addition, the more complex the software systems are and the more difficult it is to ensure such properties. As described in the remainder of the introduction, guaranteeing safety properties for complex distributed software systems is what guides this dissertation.

Problem Description

Complex software systems, in particular distributed ones, are everywhere around us and are at the basis of our everyday activities.

These systems are highly *mobile* and *dynamic*: programs or devices may move and may often execute in networks owned and operated by other parties; new devices or pieces of software may be added; the operating environment or the software requirements may change over time.

These systems are also *heterogeneous* and *open*: the pieces that form a system may be quite different from each other, built by different people or industries, even using different infrastructures or programming languages; the constituents of a system only have a partial knowledge of the overall system, and may only know, or be aware of, a subset of the entities that operate in the system.

These systems are often being thought of and designed as structured composition of computational units often referred to as *components*, which give rise to the name of Component-Based Ubiquitous Systems (CBUS) [59]. These components are supposed to *interact* and *communicate* with each other following some predefined patterns or protocols. The notion of component is widely used also in industry, in particular the following informal definition, from Szyperski [103] is often adopted: "a software component is a unit of composition with contractu-

ally specified interfaces and explicit context dependencies". An *interface* is a set of named operations that can be invoked by clients and *context dependencies* are specifications of what the deployment environment needs to provide, such that the components can properly function.

In order to handle the complexity of distributed systems, it is natural to aim at verification methods and techniques that are *compositional*. On the other hand, compositionality is also useful and can be exploited in dealing with the inherent heterogeneity of software components.

When reasoning about complex distributed systems, their *reliability* and their *usability* are fundamental and necessary requirements.

i) In order to be reliable, compositional models of software systems need to account for *dynamic reconfiguration*, i.e., changing at runtime the communication patterns. This is important because the needs and the requirements of a system may change over time. This may happen because the original specification was incomplete or ambiguous, or because new needs arise that had not been predicted at design time. As designing and deploying a system is costly, it is important for the system to be capable of adapting itself to changes in the surrounding environment. In addition, this is also important when modelling failure recovery.

ii) In order to be useful, compositional models of software systems need to account for *interaction*. Interaction can be seen as *communication* patterns among components which collaborate together to achieve a common task.

As far as *i*) is concerned it is important to understand how correctness and consistency criteria can be enforced. Guaranteeing consistency of dynamic reconfigurations, especially the unplanned ones, is challenging, since it is difficult to ensure that such modifications will not disrupt ongoing computations.

As far as *ii*) is concerned it is important to understand how correctness and safety criteria can be enforced. In the communication setting, the notion of safety comes as a collection of several requirements, including basic properties like *privacy*, guaranteeing that the communication means is owned only by the communicating parties or *communication safety*, guaranteeing that the protocol has the expected structure. Stronger safety properties related to communication may be desirable like *deadlock-freedom*, guaranteeing that the system does not get stuck or *progress*, guaranteeing that every engaged communication or protocol satisfies all the requested interactions. Enforcing each of the previous safety requirements is a difficult task, which becomes even more difficult if one wants to enforce a combination of them. In many distributed systems, in particular, safety critical systems, a combination of these properties is required.

Aim of the Ph.D. and Methodology

The aim of the Ph.D. was to develop powerful techniques based on formal methods for the verification of correctness, consistency and safety properties related to dynamic reconfigurations and communications in complex distributed systems.

In particular, static analysis techniques based on types and type systems appear to be an adequate methodology, as they stand at the formal basis of useful programming tools. Before using them in a practical setting, a rigorous development of such techniques is needed, which is more easily done on models and core languages, such as object-oriented and concurrent calculi.

The reason why we think our methodology is adequate is twofold.

i) Type systems are a very adequate means to guarantee *safety properties*. Their benefits are well-known in sequential programming, starting from early detection of programming errors to facilitating code optimisation and readability. In concurrent and distributed programming the previous benefits still hold and in addition other properties, typical of these systems, can be guaranteed by using types and type systems. In particular, there has been a considerable effort over the last 20-years in the development of types for processes, mainly in the π-calculus [84, 101] or variants of it, which is by all means the calculus mostly used to model concurrent and distributed scenarios. For instance, types have been proposed to ensure termination, so that when we interrogate a well-typed process we are guaranteed that an answer is eventually produced [34, 100, 117], or deadlock-freedom, ensuring that a well-typed process never reaches a deadlocked state, meaning that communications will eventually succeed, unless the whole process diverges [63, 68, 71], or a stronger property, that of lock-freedom [64, 65, 72] ensuring that communication of well-typed processes will succeed, (under fair scheduling), even if the whole process diverges. Types and type systems for guaranteeing safety properties have been successfully adopted also in a more complex setting than the typed π-calculus, that of concurrent component-based systems, to guarantee for example deadlock-freedom [48, 49].

ii) There are several types and type system proposals for *communication*, starting from the standard channel types in the typed π-calculus [70, 84, 98, 101] to the *behavioural types* [17, 21, 36, 55, 89, 104, 109, 112, 118], generally defined for (variants) of the π-calculus. The standard channel types are foundational. They are simple, expressively powerful and robust and they are well-studied in the literature. Moreover, they are at the basis of behavioural types, which were defined later in time. In this dissertation, we concentrate on the standard channel types

and on the *session types*, the latter being a formalism used to describe and model a protocol as a type abstraction. We concentrate on session types because they are designed in such a way that they guarantee several safety properties, like privacy of the communication channel or communication safety and also *session fidelity* stating that the type of data transmitted and the structure of the session type are as expected. However, as previously stated, we are interested in studying also stronger properties, like deadlock-freedom of communicating participants or progress of a session, as we will see in the remainder.

Contribution

The contributions brought by this dissertation are listed in the following.

- We design a type system for a concurrent object-oriented calculus, to statically ensure consistency of dynamic reconfigurations related to modifications of communication patterns in a program. The type system statically tracks runtime information about the objects. It can be seen as a technique which can be applied to other calculi and frameworks, for purposes related to tracking runtime information during compile time.

- We present an encoding of the π-calculus with (dyadic) session types and terms to the standard typed π-calculus, by showing thus that the primitives of the former can be seen as macros for the already existing constructs in the π-calculus. The goal of the encoding is to understand the expressive power of session types and to which extent they are more sophisticated and complex than the standard π-calculus channel types.
  The importance of the encoding is foundational, since

  - The encoding permits us to derive properties and theoretical results of the session π-calculi by exploiting the already well-known theory of the typed π-calculus. Just to mention few of them, properties like Subject Reduction, or Type Safety in session π-calculi are obtained as corollaries of the encoding and of the corresponding properties in the standard typed π-calculus.
  - The encoding is shown to be robust by analysing non trivial extensions like, subtyping, polymorphism, higher-order and recursion. By extending the encoding to the new types and terms constructs that these

features introduce, we show how to derive properties in the π-calculus with sessions by exploiting the corresponding properties in the standard typed π-calculus and the extended encoding.

  - The encoding is an expressivity result for session types, which are interpreted in the standard typed π- calculus. There are many more expressivity results in the untyped settings as opposed to expressivity results in the typed ones.

- We study advanced safety properties related to communication in complex distributed systems. We concentrate on (dyadic) session types and study properties like deadlock-freedom, livelock-freedom and progress. We explain what is the relation among these properties and we present a type system for guaranteeing the progress property by exploiting the encoding previously mentioned.

Structure of the Dissertation

The structure of the dissertation is given in the following. Every part is roughly an extension of the previous one and is self-contained.

- Part I: Safe Dynamic Reconfiguration of Components.
  This part focuses on components and is based on [30]. Chapter 1 introduces the component calculus, which is a concurrent object-oriented language designed for distributed programs. Chapter 2 introduces a type system for the component calculus, which statically guarantees consistency properties related to runtime modifications of communication patterns. Chapter 3 gives the theoretical results and properties that the component type system satisfies, as well as the detailed proofs.

- Part II: Safe Communication by Encoding.
  This part focuses on the encoding of session types and terms and is based on [31]. Chapter 4 presents the typed π-calculus [101]. We give the syntax of types and terms, the operational semantics, and the typing rules. Chapter 5 gives a detailed overview of the notions of sessions and session types, as well as the statics and dynamics of a session calculus [109]. Chapter 6 gives the encoding of session types into *linear channel types* and *variant types* and the encoding of session terms into standard typed π- calculus

terms. In addition, we present the theoretical results and their detailed proofs, that validate our encoding.

- Part III: Advanced Features on Safety by Encoding.
  This part is a continuation of the previous one. It shows the robustness of the encoding by analysing important extensions to session types and by showing yet the validity of our encoding. In particular, Chapter 7 focuses on subtyping; Chapter 8 on polymorphism; Chapter 9 on higher-order and Chapter 10 focuses on recursion. Chapter 11 gives an alternative encoding and hence an alternative way of obtaining session types.

- Part IV: Progress of Communication.
  This part focuses on the progress property for sessions and is based on [19]. Chapter 12 and Chapter 13 give a background on the standard π-calculus typed with usage types and the π-calculus with sessions, respectively, which report few modifications with respect to the ones introduced in Part II. In particular, Chapter 12 focuses on types and the type system for guaranteeing the lock-freedom property. Chapter 14 introduces the notion of progress for the π-calculus with session, by relating it to the notion of lock-freedom for sessions. In addition, it gives a static way for checking progress for sessions, by using the type system for lock-freedom given in Chapter 12.

Part I

Safe Dynamic Reconfiguration of Components

Introduction to Part I

In modern complex distributed systems, unplanned dynamic reconfiguration, i.e., changing at runtime the communication pattern of a program, is challenging as it is difficult to ensure that such modifications will not disrupt ongoing computations. In [75] the authors propose to solve this problem by integrating notions coming from component models [4, 11, 14, 29, 81, 85] within the *Abstract Behavioural Specification* programming language (ABS) [60].

We start this dissertation with a component-extension of the ABS calculus because it has interesting constructs for modelling components, especially reconfigurable components and hence for designing complex distributed systems. The reconfigurable component constructs can be adopted in calculi and languages other than ABS in order to model a component-layer system and to address the (dynamic) reconfiguration problem. The communication-based problems are addressed (in the remainder parts of the dissertation) after a solid system is built.

ABS is an actor-based language and is designed for distributed object-oriented systems. It integrates *concurrency* and *synchronisation* mechanisms with a *functional* language. The concurrency and synchronisation mechanisms are used to deal with data races, whether the functional level is used to deal with data, namely, data structures, data types and functional expressions. Actors, called *concurrent object groups*, *cog*s or simply *group*s, are dynamic collections of collaborating objects. Cogs offer consistency by guaranteeing that at most one method per cog is executing at any time. Within a cog, objects collaborate using (*synchronous*) method calls and *collaborative concurrency* with the **suspend** and **await** operations which can suspend the execution of the current method, and thus allow

another one to execute. Between cogs, collaboration is achieved by means of *asynchronous* method calls that return *future*, i.e., a placeholder where the result of the call is put when its computation finishes. Futures are first-class values. ABS ensures the *encapsulation* principle by using interfaces to type objects and thus by separating the interface from its (possibly) various implementations. For the same reason classes are (possibly) parametrised in a sequence of typed variables. In this way, when creating a new object, the actual parameters initialise the class' formal parameters, differently from other object-oriented languages, where the fields are the one to be initialised. The fields in ABS are initialised by calling a special method *init*(C), or differently one can initialise them in a second step after the object creation by performing an assignment statement. – In the present work we adopt the latter way of initialising an object's fields. –

ABS is a fully-fledged programming language. On top of the implementation of ABS language, the authors in [60] define the Core ABS, a calculus that abstracts from some implementation details. In the remainder of this part, we use the Core ABS calculus. However, without leading to confusion, we often will refer to it as simply the ABS language.

On top of the ABS language, [75] adds the notion of *components*, and more precisely, the notions of *ports*, *bindings* and *safe state* to deal with dynamic reconfiguration. Ports define variability points in an object, namely they define the access points to functionalities provided by the external environment, and can be *rebound* (i.e., modified) from outside the object. On the contrary, fields, which represent the inner state of the object, can only be modified by the object itself. To ensure consistency of the **rebind** operation, [75] enforces two constraints on its application: *i*) it is only possible to rebind an object's port when the object is in a *safe state*; and *ii*) it is only possible to rebind an object's port from *any* object within the *same* cog. Safe states are modelled by annotating methods as **critical**, specifying that while a **critical** method is executing, the object is *not* in a safe state. The resulting language offers a consistent setting for dynamic reconfigurations, which means performing modifications on a program at runtime while still ensuring consistency of its execution.

On the other hand, consistency is based on two constraints: synchronous method calls and rebinding operations must involve two objects in the same cog. These constraints are enforced at *runtime*; therefore, programs may encounter unexpected runtime errors during their execution.

The contribution of Part I of the dissertation, is to *statically* check that synchronous method calls and rebinding operations are consistent. In particular, we

define a type system for the aforementioned component model that ensures the legality of both synchronous method calls and port rebindings, guaranteeing that well-typed programs will always be consistent.

Our approach is based on a static tracking of group membership of the objects. The difficulty in retrieving this kind of information is that cogs as well as objects are dynamic entities. Since we want to trace group information statically, we need a way to identify and track every group in the program. To this aim, we define a technique that associates to each group creation a fresh *group name*. Then, we keep track of which cog an object is allocated to, by associating to each object a *group record*. The type system checks that objects indeed have the specified group record, and uses this information to ensure that synchronous calls and rebindings are always performed locally to a cog. The type system is proven to be sound with respect to the operational semantics. We use this result to show that well-typed programs do not violate consistency during execution.

Roadmap to Part I The rest of Part I is organised as follows. Chapter 1 gives a detailed presentation of the component calculus. We start by introducing the syntax of terms and types, give the operational semantics and we conclude by presenting a running example that illustrates the calculus, its features and the problems we deal with. Chapter 2 presents the main contribution of this Part of the dissertation, namely the type system. We start by explaining the features of types, then we present the subtyping relation and we conclude with the typing rules for the component calculus. In Chapter 3 we present the properties that our type system satisfies and give the complete proofs of these properties.

CHAPTER 1

Background on Components

In this chapter we give an overview of the component calculus, which is an extension of the ABS language. We first present the syntax of terms and types; then the operational semantics and we conclude by giving a running example which illustrates the main features of components.

1.1 Syntax

The calculus we present in the following is an extension of the ABS language [60] and is mainly inspired by the component calculus in [75]. It is a concurrent object-oriented calculus designed for distributed programs. It is roughly composed by a *functional* part, containing data types and data type constructors, pure functional expressions and **case** expressions; and a *concurrent* part, containing object and object/cog creations, synchronous and asynchronous method calls, **suspend**, **await** and **get** primitives. We include the functional part of the language in order to have a complete general-purpose language, which can be used in practice, as ABS. Notice that, the functional part is present in ABS but is not present in the component calculus [75]. On the other hand, we include in the present calculus the component primitives from [75], like **port** and **rebind** and **critical** methods to deal with critical sections. Notice that the component part is not present in the original ABS language. The present calculus differs, from both calculi mentioned above, in the

P	::=	$\overline{Dl}\ \{\ s\ \}$	Program
Dl	::=	$D \mid F \mid I \mid C$	Declarations
T	::=	$\mathtt{V} \mid \mathtt{D}[\langle\overline{T}\rangle] \mid (\mathtt{I}, \mathbb{r})$	Type
$\mathbb{r}$	::=	$\bot \mid \mathtt{G}[\overline{f : T}] \mid \alpha \mid \overline{\mu\alpha.\mathbb{r}}$	Record
D	::=	$\textbf{data}\ \mathtt{D}[\langle\overline{T}\rangle] = \mathtt{Co}[(\overline{T})] \vert \mathtt{Co}[(\overline{T})];$	Data Type
F	::=	$\textbf{def}\ T\ \mathtt{fun}[\langle\overline{T}\rangle](\overline{T\ x}) = e;$	Function
I	::=	$\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}$	Interface
C	::=	$\textbf{class}\ \mathtt{C}[(\overline{T\ x})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl}\ \overline{M}\ \}$	Class
Fl	::=	$[\textbf{port}]\ T\ x$	Field Declaration
S	::=	$[\textbf{critical}]\ (\mathcal{G}, \mathbb{r})\ T\ \mathtt{m}(\overline{T\ x})$	Method Header
M	::=	$S\ \{\ s\ \}$	Method Definition
s	::=	$\textbf{skip} \mid s\ ;\ s \mid T\ x \mid x = z \mid \textbf{await}\ g$	Statement
		$\mid\ \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \{\ s\ \} \mid \textbf{return}\ e$	
		$\mid\ \textbf{rebind}\ e.p = z \mid \textbf{suspend}$	
z	::=	$e \mid \textbf{new}\ [\textbf{cog}]\ \mathtt{C}\ (\overline{e}) \mid e.\mathtt{m}(\overline{e}) \mid e!\mathtt{m}(\overline{e}) \mid \textbf{get}(e)$	Side Effects Expression
e	::=	$v \mid x \mid \mathtt{fun}(\overline{e}) \mid \textbf{case}\ e\ \{\overline{p \Rightarrow e_p}\} \mid \mathtt{Co}[(\overline{e})]$	Pure Expression
v	::=	$\mathtt{true} \mid \mathtt{false} \mid \textbf{null} \mid \mathtt{Co}[(\overline{v})]$	Value
p	::=	$\_ \mid x \mid \textbf{null} \mid \mathtt{Co}[(\overline{p})]$	Pattern
g	::=	$e \mid e? \mid \|e\| \mid g \wedge g$	Guard

Figure 1.1: Component Extension of Core ABS

syntax of types which use group information. Moreover, for the sake of readability, the component calculus we consider lacks the notion of *location*, present in [75]. This notion is orthogonal to the notion of ports and rebinding operations and does not influence the aim of our work. The validity of our approach and of our type system still holds for the full calculus.

The syntax of the component calculus is given in Fig. 1.1. It is based on several categories of names: `I` and `C` range over interface and class names; `V` ranges over type variables for polymorphism; `G` ranges over cog names –which will be explained in details in the remainder; `D`, `Co` and `fun` range respectively over data type, constructor and function names; `m`, f and p range respectively over method, field and port names –often, in order to have a uniform presentation, we will use f for both fields and ports; and x ranges over variables, in particular it ranges over the special variable **this**, indicating the current object, and the special variable **destiny**, indicating the placeholder for the returned value of the method

invocation. We adopt the following notations in the syntax and in the rest of the work: an overlined element corresponds to any finite, possibly empty, sequence of such element; and an element between square brackets is optional.

A program P consists of a sequence of declarations $\overline{Dl}$ ended by a main block, namely a statement s to be executed. We refer to the sequence of declarations in a program as a *Class Table* (CT), the same way as called in [58].

Declarations Dl include data type declarations D, function declarations F, interface declarations I and class declarations C.

A type T can be: a type variable `V`; a data type name `D` which can be a ground type like `Bool`, `Int` or a future $\texttt{Fut}\langle T\rangle$, used to type data structures – we will thoroughly explain future types in the remainder; or a pair consisting of an interface name `I` and a *record* $\mathbb{r}$ to type objects. Records are a new concept and are associated to types in order to track group information. The previous calculi, neither ABS [60] nor its component extension [75] used the notion of records in types. A record can be: $\bot$, meaning that the structure of the object is unknown; $\texttt{G}[\overline{f : T}]$, meaning that the object is in the cog `G` and its fields $\overline{f}$ are typed with $\overline{T}$; or regular terms, using the standard combination of variables α and the μ-binder.

Data types D have at least one constructor, with name `Co`, and possibly a list of type parameters $\overline{T}$. Examples of data types are: **data** `IntList` = `NoInt` | $\texttt{Cons}(\texttt{Int}, \texttt{IntList})$, or parametric data types **data** $\texttt{List}\langle T\rangle$ = Nil | $\texttt{Cons}(T, \texttt{List}\langle T\rangle)$, or predefined data types like **data** `Bool` = `true` | `false` ; or **data** `Int` ; or **data** $\texttt{Fut}\langle T\rangle$; where the names of the predefined data types are used as types, as given by the production T introduced earlier.

Functions F are declared with a return type T, a name `fun`, a list of parameters $\overline{T\ x}$ and a code or body e. Note that both data types and functions can also have in input type parameters for polymorphism.

Interfaces I declare methods and ports that can be modified at runtime.

Classes C implement interfaces; they have a list of fields and ports Fl and implement all declared methods. Classes are possibly parametrised in a sequence of typed variables, $\overline{T\ x}$, as in Core ABS and in its implementation. This respects the *encapsulation* principle, often desired in the object-oriented languages. There is a neat distinction between the *parameters* of the class, which are the ones that the class exhibits, and the inner fields and ports of the class, given by $\overline{Fl}$.

Method headers S are used to declare methods with their classic type annotation, and *i*) the possible annotation **critical** that ensures that no rebinding will be performed on that object during the execution of that method; *ii*) a *method signature* $(\mathcal{G}, \mathbb{r})$ which will be described and used in our type system section.

Method declarations M consist of a header and a body, the latter being a sequential composition of local variables and commands.

Statements s are mainly standard. Statements **skip** and $s_1; s_2$ indicate the empty statement and the composition statement, respectively. Variable declaration $T\ x$, as in many programming languages and also in the implementation of the ABS language, is a statement. Assignment statement $x = z$ assigns an expression with side-effects to variable x. The statement **await** g suspends the execution of the method until the guard g is `true`. Statements **if** e **then** s_1 **else** s_2 and **while** e { s } indicate the standard conditional and loop, respectively. Statement **return** e returns the expression e after a method call. Statement **rebind** $e.p = z$ rebinds the port p of the object e to the value stored in z, and statement **suspend** merely suspends the currently active process.

Expressions are divided in expressions with side effects, produced by z and pure expressions, produced by e. We will often use the term expression to denote both of them, when it does not lead to confusion.

Expressions z include: pure expressions e; **new** `C` $(\overline{e})$ and **new cog** `C` $(\overline{e})$ that instantiate a class `C` and place the object in the current cog and in a new cog, respectively; synchronous $e.\mathtt{m}(\overline{e})$ and asynchronous $e!\mathtt{m}(\overline{e})$ method calls, the latter returning a future that will hold the result of the method call when it will be computed; and **get**(e) which gives the value stored in the future e, or actively waits for it if it is not computed yet.

Pure expressions e include values v, variables x, function call $\mathtt{fun}(\overline{e})$, pattern matching **case** e $\{\overline{p \Rightarrow e_p}\}$ that tests e and executes e_p if it matches p and a constructor expression $\mathtt{Co}[(\overline{e})]$, possibly parametrised in a sequence of expressions.

Values v can be **null** or a constructor value $\mathtt{Co}[(\overline{v})]$, possibly parametrised in a sequence of values. For example, values `true` and `false` are obtained as values from the corresponding constructor, as defined previously by the data type `Bool`.

Patterns p are standard, they include wildcard _ which matches everything, variable x which matches everything and binds it to x, value **null** which matches a null object and value $\mathtt{Co}(\overline{p})$ which matches a value $\mathtt{Co}(\overline{e_p})$ where p matches e_p.

Finally, a guard g can be: an expression e; e? which is `true` when the future e is completed, `false` otherwise; $\|e\|$ which is `true` when the object e is in a safe state, i.e., it is not executing any **critical** method, `false` otherwise; and the conjunction of two guards $g \wedge g$ has the usual meaning.

1.2 Semantics

In this section we present the operational semantics of the component calculus, given in Fig. 1.1. It is defined as a transition system on the runtime configurations. So, we first define the runtime configurations and then give some auxiliary functions on which the operational semantics relies on. We conclude with the operational semantics rules.

1.2.1 Runtime Syntax

The operational semantics is defined over *runtime configurations*, presented in Fig. 1.2 which extend the language with constructs used during execution, namely runtime representations of objects, groups and tasks.

Let o, f and c range over object, future, and cog identifiers, respectively.

A runtime configuration N can be empty, denoted with ϵ, an interface, a class, an associative and commutative union of configurations $N\ N'$, an object, a cog, a future or an invocation message. An object $ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q)$ has a name o; a substitution σ mapping the object's fields, ports and special variables (**this**, **class**, **cog**, **destiny**) to values; a running process $K_{\mathbf{idle}}$, that is **idle** if the object is idle; and a queue of *suspended processes* Q. A process K is $\{\ \sigma \mid s\ \}$ where σ maps the local variables to their values and s is a list of statements. The statements are the ones presented in Fig. 1.1 augmented with the statement **cont**(f), used to control continuation of synchronous calls, and the special sequential composition $s\ ;;\ s$ which will be explained in the operational semantics rules. A substitution σ is a mapping from variable names to values. For convenience, we associate the declared type of the variable with the binding, and we also use substitutions to store: *i*) in case of substitutions directly included in objects, their **this** reference, their **class**, their **cog**, and an integer denoted by $\mathtt{nb}_{cr}$ which counts how many open critical sections the object has; and *ii*) in case of substitution directly included in tasks, **destiny** is associated to future return value. A cog $cog(\mathtt{c}, \mathtt{o}_\varepsilon)$ has a name c and a running object $\mathtt{o}_\varepsilon$, which is ε when no execution is taking place in the cog. A future $fut(\mathtt{f}, v_\perp)$ has a name f and a value $v_\perp$ which is $\perp$ when the value has not been computed yet. Finally, the invocation message $invoc(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})$, which is the initial form of an asynchronous call, consists of the callee identifier o, the name of the future f where the call's result should be returned, the invoked method name m, and the call's actual parameters $\overline{v}$.

The *initial state* of a program is denoted by $ob(start, \epsilon, p, \emptyset)$ where the process

$$
\begin{array}{rcl@{\qquad}rcl}
N & ::= & \epsilon \mid I \mid C \mid N\,N & s & ::= & \mathbf{cont}(\mathtt{f}) \mid s\,;;\,s \mid \cdots \\
 & \mid & ob(\mathtt{o}, \sigma, K_{\mathbf{idle}}, Q) & \sigma & ::= & \varepsilon \mid \sigma; T\ x\ v \mid \sigma; \theta \\
 & \mid & cog(\mathtt{c}, \mathtt{o}_{\varepsilon}) & \theta & ::= & \varepsilon \mid \theta; \mathbf{this}\ \mathtt{o} \\
 & \mid & fut(\mathtt{f}, v_{\perp}) & & \mid & \theta; \mathbf{class}\ \mathtt{C} \mid \theta; \mathbf{cog}\ \mathtt{c} \\
 & \mid & invoc(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}) & & \mid & \theta; \mathbf{destiny}\ v \mid \theta; \mathtt{nb}_{cr}\ v \\
Q & ::= & \epsilon \mid K \mid Q \cup Q & v_{\perp} & ::= & v \mid \perp \\
K & ::= & \{\, \sigma \mid s \,\} & \mathtt{o}_{\varepsilon} & ::= & \mathtt{o} \mid \varepsilon \\
v & ::= & \mathtt{o} \mid \mathtt{f} \mid \cdots & K_{\mathbf{idle}} & ::= & K \mid \mathbf{idle}
\end{array}
$$

Figure 1.2: Runtime Syntax

p is the activation of the main block of the program. We call *execution* of a program a sequence of reductions established by the operational semantics rules starting from the initial state of the program.

1.2.2 Functions and Predicates

In this section we introduce the auxiliary functions and predicates that are used to define the operational semantics of the calculus.

Function bind($\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}, \mathtt{C}$) returns the process being the instantiation of the body of method `m` in class `C` with **this** bound to `o`, **destiny** bound to `f`, and the parameters of the method bound to the actual values $\overline{v}$. If the method is **critical**, then $\mathtt{nb}_{cr}$ is first incremented and then decremented when the method finishes its execution. Instead, if binding does not succeed, then **error** is returned. Since, in the component calculus we have standard and **critical** methods, the bind function is defined differently from the corresponding one in ABS – whereas, the rest of the functions and predicates are defined in the same way. Formally, the bind function is defined by the following two rules, where rule (NM-Bind) applies for a *normal*

method and rule (CM-BIND) applies for a *critical method*:

$$\frac{\begin{array}{c}\text{(NM-BIND)}\\ \textbf{class } \mathtt{C} \ldots \{T\,\mathtt{m}(\overline{T}\,x)\{\overline{T'\ x'}\ s\} \ldots\} \in N\end{array}}{\text{bind}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}, \mathtt{C}) = \{\overline{T\ x = v}; \overline{T'\ x' = \textbf{null}}; \textbf{this} = \mathtt{o} \mid s\}}$$

$$\frac{\begin{array}{c}\text{(CM-BIND)}\\ \textbf{class } \mathtt{C} \ldots \{\textbf{critical } T\,\mathtt{m}(\overline{T}\,x)\{\overline{T'\ x'\ s'}\} \ldots\} \in N\\ s = \mathtt{nb}_{cr} = \mathtt{nb}_{cr} + 1; s'; \mathtt{nb}_{cr} = \mathtt{nb}_{cr} - 1\end{array}}{\text{bind}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}, \mathtt{C}) = \{\overline{T\ x = v}; \overline{T'\ x' = \textbf{null}}; \textbf{this} = \mathtt{o} \mid s\}}$$

Function atts($\mathtt{C}, \overline{v}, \mathtt{o}, \mathtt{c}$) returns the initial state of an instance of class C with its fields, **this** and **cog** mapped to $\overline{v}$, o and c, respectively.

Function select(Q, σ, N) selects from the queue of suspended processes the next process to be active.

Predicate fresh is defined on names of objects o, futures f and names of cogs c and asserts that these names are globally unique. It is defined on a variable x or a sequence of variables $\{x_1 \ldots x_n\}$ and asserts that the variables are globally new.

Function match(p, v) returns a unique substitution σ such that $\sigma(p) = v$ and $\mathsf{dom}(\sigma) = \text{vars}(p)$, otherwise $\text{match}(p, v) = \bot$.

Function vars(p) returns the set of variables in the pattern p and is defined by induction on the structure of p: $\text{vars}(\_) = \text{vars}(\textbf{null}) = \emptyset$, $\text{vars}(x) = \{x\}$ and $\text{vars}(\mathsf{Co}(p_1 \ldots p_n)) = \bigcup_i \text{vars}(p_i)$.

1.2.3 Evaluation of pure and guard expressions

In this section we present the evaluation of pure expressions and guard expressions, before introducing the operational semantics on runtime configurations.

Pure expressions The evaluation of pure expressions is defined by a small-step reduction relation $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$ and is given in Fig. 1.3. Let σ be a substitution and e be a pure expression, then the reduction relation means that expression e in the context σ reduces to expression e' in the context σ'. We use the notation $e[\overline{x} \mapsto \overline{y}]$ to denote the expression e in which every occurrence of variable x_i is substituted by variable y_i. The same holds for $\sigma[\overline{x} \mapsto \overline{y}]$. For simplicity in the reduction rules to follow, we denote with $[\![e]\!]_\sigma$ the evaluation of the expression e

$$\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \le i \le n}{\sigma \vdash \mathtt{Co}(e_1 \dots e_i \dots e_n) \rightsquigarrow \sigma' \vdash \mathtt{Co}(e_1 \dots e_i' \dots e_n)} \text{(RedCons)}$$

$$\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \le i \le n}{\sigma \vdash \mathtt{fun}(e_1 \dots e_i \dots e_n) \rightsquigarrow \sigma' \vdash \mathtt{fun}(e_1 \dots e_i' \dots e_n)} \text{(RedFunExp)}$$

$$\frac{}{\sigma \vdash x \rightsquigarrow \sigma \vdash \sigma(x)} \text{(RedVar)}$$

$$\frac{\mathrm{fresh}(\{y_1 \dots y_n\}) \quad \overline{y} = y_1 \dots y_n \quad |\overline{x}_{\mathtt{fun}}| = n}{\sigma \vdash \mathtt{fun}(\overline{v}) \rightsquigarrow \sigma[\overline{y} \mapsto \overline{v}] \vdash \overline{e}_{\mathtt{fun}}[\overline{x}_{\mathtt{fun}} \mapsto \overline{y}]} \text{(RedFunGround)}$$

$$\frac{\sigma \vdash e \rightsquigarrow \sigma' \vdash e'}{\sigma \vdash \mathbf{case}\ e\ \{\overline{p \Rightarrow e_p}\} \rightsquigarrow \sigma' \vdash \mathbf{case}\ e'\ \{\overline{p \Rightarrow e_p}\}} \text{(RedCase1)}$$

$$\frac{\mathrm{match}(\sigma(p), v) = \bot}{\sigma \vdash \mathbf{case}\ v\ \{p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}}\} \rightsquigarrow \sigma \vdash \mathbf{case}\ v\ \{\overline{p' \Rightarrow e'_p}\}} \text{(RedCase2)}$$

$$\frac{\overline{y} = y_1 \dots y_n \quad \mathrm{fresh}(\{y_1 \dots y_n\}) \quad \overline{x} = x_1 \dots x_n \quad \{x_1 \dots x_n\} = \mathrm{vars}(\sigma(p)) \quad \mathrm{match}(\sigma(p), v) = \sigma'' \quad \sigma' = \sigma[y_i \mapsto \sigma''(x_i)]\ \forall i}{\sigma \vdash \mathbf{case}\ v\ \{p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}}\} \rightsquigarrow \sigma' \vdash e_p[\overline{x} \mapsto \overline{y}]} \text{(RedCase3)}$$

Figure 1.3: The evaluation of pure expressions

in the context σ to its ground value, given by the production v. In particular, when e is a boolean expression, then $[\![e]\!]_\sigma = \mathtt{true}$ and $\neg[\![e]\!]_\sigma = \mathtt{false}$.

Rule (RedCons) states that the expression $\mathtt{Co}(e_1 \dots e_i \dots e_n)$ reduces to $\mathtt{Co}(e_1 \dots e_i' \dots e_n)$ whenever e_i reduces to e_i'. Rule (RedVar) states that variable x in the context σ evaluates to the value assigned by σ, namely $\sigma(x)$, in the same context. Function evaluation is given by rules (RedFunExp) and (RedFunGround). A function `fun` is defined by **def** T $\mathtt{fun}(\overline{T\ x}) = e$, and we denote by $\overline{x}_{\mathtt{fun}}$ the list of formal parameters $\overline{x}$ and by $\overline{e}_{\mathtt{fun}}$ the body e of the function; namely, we use the subscript `fun` to state the belonging to the function having name `fun`. By rule (RedFunGround), the evaluation of a function call $\mathtt{fun}(\overline{v})$ in a context σ reduces to the evaluation of $\overline{e}_{\mathtt{fun}}[\overline{x}_{\mathtt{fun}} \mapsto \overline{y}]$ in $\sigma[\overline{y} \mapsto \overline{v}]$. First of all, in order to get the values $\overline{v}$, the reduction rule(RedFunExp) is applied, where the expressions $\overline{e}$ are evaluated to values $\overline{v}$. In addition, the change in scope in evaluating a function body is obtained by replacing the list of formal parameters $\overline{x}_{\mathtt{fun}}$ by fresh variables

$\overline{y}$ in the body of the function, thus avoiding name capture. There are three reduction rules for case expressions. Rule (RedCase1) states that the case expression **case** $v\ \{p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}}\}$ reduces if its argument e reduces. Case expressions reduce only if the pattern in one of the branches matches. In order to achieve this, we use the function match(p, v), which returns the unique substitution σ such that $\sigma(p) = v$ and $\mathsf{dom}(\sigma) = \text{vars}(p)$, otherwise match$(p, v) = \bot$. Rules (RedCase2) and (RedCase3) check this matching function. In case match$(\sigma(p), v) = \bot$, then the case expression **case** $v\ \{p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}}\}$ reduces to **case** $v\ \{\overline{p' \Rightarrow e'_{p'}}\}$. Otherwise, if match$(\sigma(p), v) \neq \bot$, first variables in p are bound to ground values, given by the substitution σ'' and then, in order to avoid names to be captured, variables in $\overline{x}$ are substituted by fresh variables in $\overline{y}$, which in turn have associated values given by $\sigma''(\overline{x})$. Thus, the context for evaluating the new expression is σ augmented with $\overline{y}$ associated to $\sigma''(\overline{x})$. Then, the case expression reduces to the body e_p of the corresponding branch, where $\overline{x}$ is replaced by $\overline{y}$.

Guard expressions The evaluation of guards is given in Fig. 1.4.

$$\frac{\sigma \vdash e \rightsquigarrow \sigma \vdash \mathtt{f} \quad \mathit{fut}(\mathtt{f}, v) \in N \quad v \neq \bot}{\sigma, N \vdash e? \rightsquigarrow \sigma, N \vdash \mathtt{true}}\ (\text{RedReply1}) \qquad \frac{\sigma \vdash e \rightsquigarrow \sigma \vdash \mathtt{f} \quad \mathit{fut}(\mathtt{f}, \bot) \in N}{\sigma, N \vdash e? \rightsquigarrow \sigma, N \vdash \mathtt{false}}\ (\text{RedReply2})$$

$$\frac{\sigma, N \vdash g_1 \rightsquigarrow \sigma, N \vdash g'_1 \quad \sigma, N \vdash g_2 \rightsquigarrow \sigma, N \vdash g'_2}{\sigma, N \vdash g_1 \wedge g_2 \rightsquigarrow \sigma, N \vdash g'_1 \wedge g'_2}\ (\text{RedConj})$$

$$\frac{\sigma \vdash e \rightsquigarrow \sigma \vdash \mathtt{o} \quad \mathit{ob}(\mathtt{o}, \sigma_\mathtt{o}, K_{\mathbf{idle}}, Q) \in N \quad \sigma_\mathtt{o}(\mathtt{nb}_{cr}) = 0}{\sigma, N \vdash \|e\| \rightsquigarrow \sigma, N \vdash \mathtt{true}}\ (\text{RedCS1}) \qquad \frac{\sigma \vdash e \rightsquigarrow \sigma \vdash \mathtt{o} \quad \mathit{ob}(\mathtt{o}, \sigma_\mathtt{o}, K_{\mathbf{idle}}, Q) \in N \quad \sigma_\mathtt{o}(\mathtt{nb}_{cr}) \neq 0}{\sigma, N \vdash \|e\| \rightsquigarrow \sigma, N \vdash \mathtt{false}}\ (\text{RedCS2})$$

Figure 1.4: The evaluation of guard expressions

Let σ be a substitution and N be a configuration. The evaluation of a guard to a ground value is either `true` or `false`. For simplicity, we denote with $[\![g]\!]^N_\sigma$ the evaluation of a guard g in a context σ, N to `true`. Hence, we denote with $\neg[\![g]\!]^N_\sigma$ the evaluation of a guard g in a context σ, N to `false`.

$$\frac{}{ob(\mathsf{o},\sigma,\{\sigma'|\mathbf{skip};s\},Q) \rightarrow ob(\mathsf{o},\sigma,\{\sigma'|s\},Q)}\ (\textsc{Skip})$$

$$\frac{x \in \mathsf{dom}(\sigma') v = [\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\sigma'|x = e;s\},Q) \rightarrow ob(\mathsf{o},\sigma,\{\sigma'[x \mapsto v]|s\},Q)}\ (\textsc{Assign-Local})$$

$$\frac{x \in \mathsf{dom}(\sigma) \quad v = [\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\sigma'|x = e;s\},Q) \rightarrow ob(\mathsf{o},\sigma[x \mapsto v],\{\sigma'|s\},Q)}\ (\textsc{Assign-Field})$$

$$\frac{[\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\sigma'|\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2;s\},Q) \rightarrow ob(\mathsf{o},\sigma,\{\sigma'|s_1;s\},Q)}\ (\textsc{Cond-True})$$

$$\frac{\neg[\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\sigma'|\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2;s\},Q) \rightarrow ob(\mathsf{o},\sigma,\{\sigma'|s_2;s\},Q)}\ (\textsc{Cond-False})$$

$$\frac{[\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\sigma'|\ \mathbf{while}\ e\ \{\ s\ \};s'\},Q) \rightarrow ob(\mathsf{o},\sigma,\{\sigma'|\ s;;\mathbf{while}\ e\ \{\ s\ \};s'\},Q)}\ (\textsc{While-True})$$

$$\frac{\neg[\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\sigma'|\ \mathbf{while}\ e\ \{\ s\ \};s'\},Q) \rightarrow ob(\mathsf{o},\sigma,\{\sigma'|\ \mathbf{skip};s'\},Q)}\ (\textsc{While-False})$$

$$\frac{}{ob(\mathsf{o},\sigma,\{\sigma'|\ \mathbf{suspend};s\},Q) \rightarrow ob(\mathsf{o},\sigma,\mathbf{idle},Q \cup \{\sigma'|s\})}\ (\textsc{Suspend})$$

$$\frac{\mathsf{c} = \sigma(\mathbf{cog})}{ob(\mathsf{o},\sigma,\ \mathbf{idle},Q)\ cog(\mathsf{c},\mathsf{o}) \rightarrow ob(\mathsf{o},\sigma,\mathbf{idle},Q)\ cog(\mathsf{c},\epsilon)}\ (\textsc{Release-Cog})$$

$$\frac{p = \mathrm{select}(Q,\sigma,N) \quad \mathsf{c} = \sigma(\mathbf{cog})}{ob(\mathsf{o},\sigma,\mathbf{idle},Q)\ cog(\mathsf{c},\epsilon)\ N \rightarrow ob(\mathsf{o},\sigma,\{p\},Q \setminus \{p\})\ cog(\mathsf{c},\mathsf{o})\ N}\ (\textsc{Activate})$$

Figure 1.5: Reduction rules for the concurrent object level of Core ABS (1)

Rules (RedReply1) and (RedReply2) assert that the guard $e?$ evaluates to `true`, whenever the value associated to the evaluation of expression e is ready to be retrieved, namely is different from $\perp$; otherwise, it evaluates to `false`. Rule (RedConj) is trivial and asserts the evaluation of boolean conjunction of guards g_1 and g_2. Rules (RedCS1) and (RedCS2) are the new evaluation rules of the component calculus, with respect to ABS. They state that, when in the object `o` the field $\mathtt{nb}_{cr}$ is different from zero, then it has an open critical section and hence the test $\|e\|$ returns `true`; otherwise, if $\mathtt{nb}_{cr} = 0$ it means that no critical section is open, and $\|e\|$ evaluates to `false`.

$$\frac{\sigma'(\textbf{destiny}) = \mathtt{f} \quad v = [\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathtt{o},\sigma,\{\sigma'|\ \textbf{return}\ e;s\},Q)\ fut(\mathtt{f},\bot) \rightarrow ob(\mathtt{o},\sigma,\{\sigma'|s\},Q)\ fut(\mathtt{f},v)}\ (\textsc{Return})$$

$$\frac{v \neq \bot \quad \mathtt{f} = [\![e]\!]_{(\sigma\circ\sigma')}}{ob(\mathtt{o},\sigma,\{\sigma'\mid x = \textbf{get}(e);s\},Q)\ fut(\mathtt{f},v) \rightarrow ob(\mathtt{o},\sigma,\{\sigma'|x = v;s\},Q)\ fut(\mathtt{f},v)}\ (\textsc{Read-Fut})$$

$$\frac{[\![g]\!]^N_{(\sigma\circ\sigma')}}{ob(\mathtt{o},\sigma,\{\sigma'|\ \textbf{await}\ g;s\},Q)\ N \rightarrow ob(\mathtt{o},\sigma,\{\sigma'|s\},Q)\ N}\ (\textsc{Await-True})$$

$$\frac{\neg[\![g]\!]^N_{(\sigma\circ\sigma')}}{ob(\mathtt{o},\sigma,\{\sigma'|\ \textbf{await}\ g;s\},Q)\ N \rightarrow ob(\mathtt{o},\sigma,\{\sigma'|\ \textbf{suspend};\textbf{await}\ g;s,Q)\ N}\ (\textsc{Await-False})$$

$$\frac{p' = \text{bind}(\mathtt{o},\mathtt{f},\mathtt{m},\overline{v},class(\mathtt{o}))}{ob(\mathtt{o},\sigma,p,Q)\ invoc(\mathtt{o},\mathtt{f},\mathtt{m},\overline{v}) \rightarrow ob(\mathtt{o},\sigma,p,Q\cup p')}\ (\textsc{Bind-Mtd})$$

$$\frac{\overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma'')} \quad \text{fresh}(\mathtt{o}') \quad \sigma' = \text{atts}(\mathtt{C},\overline{v},\mathtt{o}',\mathtt{c})}{ob(\mathtt{o},\sigma,\{\sigma''|x = \textbf{new}\ \mathtt{C}(\overline{e});s\},Q)\ cog(\mathtt{c},\mathtt{o}) \rightarrow ob(\mathtt{o},\sigma,\{\sigma''|x = \mathtt{o}';s\},Q) \quad cog(\mathtt{c},\mathtt{o}) \quad ob(\mathtt{o}',\sigma',\textbf{idle},\varepsilon)}\ (\textsc{New-Object})$$

$$\frac{\overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma'')} \quad \text{fresh}(\mathtt{o}') \quad \text{fresh}(\mathtt{c}') \quad \sigma' = \text{atts}(\mathtt{C},\overline{v},\mathtt{o}',\mathtt{c}')}{ob(\mathtt{o},\sigma,\{\sigma''|x = \textbf{new cog}\ C(\overline{e});s\},Q) \rightarrow ob(\mathtt{o},\sigma,\{\sigma''|x = \mathtt{o}';s\},Q) \quad ob(\mathtt{o}',\sigma',\textbf{idle},\varepsilon) \quad cog(\mathtt{c}',\mathtt{o}')}\ (\textsc{New-Cog-Object})$$

Figure 1.6: Reduction rules for the concurrent object level of Core ABS (2)

1.2.4 Reduction rules

In this section we introduce the operational semantics rules for the concurrent level of ABS language and its component extension. The rules are given in Fig. 1.5, 1.6, 1.7 and 1.8.

Rule (Skip) merely executes the **skip** statement and reduces to the object having only s as part of its active process. Rules (Assign-Local) and (Assign-Field) update the values of the local variables and of the object variables, respectively, where $\sigma,[x \mapsto v]$ denotes the updating of σ with the substitution of x to v. Rules (Cond-True) and (Cond-False) select branch s_1 or branch s_2 of the **if** e **then** s_1 **else** s_2 statement if the evaluation of expression e is `true` or `false`, respectively. Rules (While-True) and (While-False) for loops are similar to the ones for conditionals. In case the evaluation of the expression e is `true`, then the

loop reduces to its body s composed with the loop itself – which is then evaluated again. Otherwise, if the evaluation returns `false` then the loop reduces to **skip**. Notice that the composition used in reduction rule (WHILE-TRUE), denoted with ';;', is a special sequential composition. As long as the operational semantics is concerned, this composition has exactly the same effect and meaning as the standard one ';'. However, we use ;; to distinguish it from ; in the typing rules for runtime configurations shown in the next chapter. Basically, the reason why we want to distinguish it in the typing rules is to handle the set of cogs created in the body of the loop, which is not known ahead of execution. Rule (SUSPEND) simply suspends the currently active process by moving it to the queue Q of suspended process. Rule (RELEASE-COG), after a process is suspended, updates the *cog* configuration, by setting the object part to **idle** meaning that there is no active object in the cog. Rule (ACTIVATE), as opposed to (SUSPEND), selects a task p from the queue of suspended processes and activates it. The process is removed from the queue and the *cog* configuration is updated by letting the object with the newly activated process, be the active object of the cog. Rule (RETURN) assigns the return value to the call's associated future. Rule (READ-FUT) retrieves the value associated to the future `f` when ready ($v \neq \bot$). Rules (AWAIT-TRUE) and (AWAIT-FALSE) define the behaviour of statement **await** g, which depending on the truth value of g either succeeds and lets the current task continue with its execution, or suspends the current task, allowing other tasks to execute (see rule (SUSPEND)), respectively. Rule (BIND-MTD) adds a process p' to the queue of the suspended processes by first letting p' be the process obtained by the bind auxiliary function after the invocation configuration is consumed. The latter provides the arguments to the bind function. Rules (NEW-OBJECT) and (NEW-COG-OBJECT) spawn a new object runtime configuration, bound to the current cog or to a newly created cog, respectively. The names of the object and the cog newly created are globally unique. The object's fields are given default values by applying function atts($C, \overline{v}$, `o'`, `c`). Rule (SELF-SYNC-CALL) looks up for the body of the method using function bind, as previously described. After the reduction, the active task for object `o` will be the body of the method (suitably instantiated with the actual parameters) and the continuation statement s will be put in the queue of the suspended processes. The statement **cont**(`f`), which is a statement added to the runtime syntax of the calculus, is used to resume the execution of s as stated by rule (SELF-SYNC-RETURN-SCHED). Rule (ASYNC-CALL) sends an invocation message to `o'` with a new (unique) future `f`, method name `m` and actual parameters $\overline{v}$. The return value of `f` is undefined (i.e., $\bot$). Rules (COG-SYNC-CALL) and (COG-SYNC-RETURN-SCHED) are specific to synchronous method

calls between objects residing in the same cog. When a method is called synchronously, inside a cog, then the active object in that cog changes, in particular from o to o′, by respecting thus that only one object per cog is active. In (Cog-Sync-Call) the **cont** statement is composed with the statement s' of the newly created process in order to be used to activate the caller in (Cog-Sync-Return-Sched). Rule (Rebind-Local) is applied when an object rebinds one of its own ports. The rule first checks that the object is not in a critical section, by testing if $\mathtt{nb}_{cr}$ is zero, and then updates the port. Rule (Rebind-Global) is applied when an object rebinds a port of another object, within the same group, and follows the same line as the previous one. Rule (Rebind-None) states that when a rebind is attempted on a port that does not exist, then nothing happens and the **rebind** operation is simply ignored and discarded. Intuitively, the reason for this rule is the following: a component can replace another one if the former offers less services, accessed by ports, than the latter – this intuition is respected by the subtyping relation which is defined in the next section. So, during execution a component can be replaced by another one with a smaller number of ports. As a consequence, if a **rebind** is performed on a port not present, this is going merely to be ignored.

1.3 Server and Client Example

In this section we present a running example which gives a better understanding of the ABS language and its component extension. In addition, this example gives a flavour of the motivation behind our type system.

Consider the following typical distributed scenario: suppose that we have several clients working together in a specific workflow and using a central server for their communications. Suppose we want to update the server. Updating the server is a difficult task, as it requires to update its reference in all clients at the same time in order to avoid communication failures.

We first consider how the above task is achieved in ABS . The code is presented in Fig. 1.9. The programmer declares two interfaces `Server` and `Client` and a class `Controller`. Basically, the class `Controller` updates the server in all the clients $\mathtt{c}_i$ by synchronously calling their setter method. All the clients are updated at the same time: since they are in the same cog as the controller they cannot execute until the execution of method `updateServer` has terminated.

However, this code does not ensure that the update is performed when the clients are in a safe state. This can lead to inconsistency issues because clients

that are using the server are not aware of the modification taking place. This problem can be solved by using the notions of **port** and **rebind** as shown in [75]. The solution is presented in Fig. 1.10. In this case, the method `updateServer` first waits for all clients to be in a safe state (**await** statement performed on the conjunction of all clients) and then updates their reference one by one (**rebind** server `s` which is declared to be a **port**).

However, even with the component extension and the presence of critical sections, runtime errors can still occur. For instance, if the clients and the controller are not in the same cog, by following the operational semantics rules, the update will fail. Consider the code in Fig. 1.11. Method `main` instantiates classes `Client` and `Controller` – and possibly other classes, like `Server`, present in the program – by creating objects $c_1, c_2, \ldots, c_n, c$. These objects are created in the same cog by the **new** command, except for client c_1, which is created and placed in a new cog by the **new cog** command. Now, suppose that the code in Fig. 1.10 is executed. At runtime, the program will check if the controller and the client belong to the same cog to respect the consistency constraints for rebinding. In case of c_1 this check will fail by leading to a runtime error.

In the remainder of the present Part, we address the aforementioned problem; namely to avoid these kind of runtime errors and the overhead in dealing with them, while performing runtime modifications. We present our type system which tracks cog membership of objects thus permitting to typecheck only programs where rebinding is consistent.

$$\frac{\begin{array}{c}\text{(Self-Sync-Call)}\\ \mathtt{o} = [\![e]\!]_{(\sigma\circ\sigma')} \quad \overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma')} \quad \sigma'(\mathbf{destiny}) = \mathtt{f}'\\ \text{fresh}(\mathtt{f}) \quad \{\sigma''|s'\} = \text{bind}(\mathtt{o},\mathtt{f},\mathtt{m},\overline{v},class(\mathtt{o}))\end{array}}{\begin{array}{c}ob(\mathtt{o},\sigma,\{\sigma' \mid x = e.\mathtt{m}(\overline{e});s\},Q)\\ \rightarrow ob(\mathtt{o},\sigma,\{\sigma''|s';\mathbf{cont}(\mathtt{f}')\},Q\cup\{\sigma'|\ x = \mathbf{get}(\mathtt{f});s\}) \quad fut(\mathtt{f},\bot)\end{array}}$$

$$\frac{\begin{array}{c}\text{(Self-Sync-Return-Sched)}\\ \sigma'(\mathbf{destiny}) = \mathtt{f}\end{array}}{ob(\mathtt{o},\sigma,\{\sigma''|\ \mathbf{cont}(\mathtt{f})\},Q\cup\{\sigma'|s\}) \rightarrow ob(\mathtt{o},\sigma,\{\sigma'|s\},Q)}$$

$$\frac{\begin{array}{c}\text{(Async-Call)}\\ \text{fresh}(\mathtt{f}) \quad \mathtt{o}' = [\![e]\!]_{(\sigma\circ\sigma')} \quad \overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma')}\end{array}}{\begin{array}{c}ob(\mathtt{o},\sigma,\{\sigma' \mid x = e!\mathtt{m}(\overline{e});s\},Q)\\ \rightarrow ob(\mathtt{o},\sigma,\{\sigma'|x = \mathtt{f};s\},Q)\ invoc(\mathtt{o}',\mathtt{f},\mathtt{m},\overline{v})\ fut(\mathtt{f},\bot)\end{array}}$$

$$\frac{\begin{array}{c}\text{(Cog-Sync-Call)}\\ \mathtt{o}' = [\![e]\!]_{(\sigma\circ\sigma'')} \quad \overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma'')} \quad \text{fresh}(\mathtt{f}) \quad \sigma'(\mathbf{cog}) = \mathtt{c}\\ \mathtt{f}' = \sigma''(\mathbf{destiny})\ \{\sigma'''|s'\} = \text{bind}(\mathtt{o}',\mathtt{f},\mathtt{m},\overline{v},class(\mathtt{o}'))\end{array}}{\begin{array}{c}ob(\mathtt{o},\sigma,\{\sigma''|x = e.\mathtt{m}(\overline{e});s\},Q)\ ob(\mathtt{o}',\sigma',\mathbf{idle},Q')\ cog(\mathtt{c},\mathtt{o})\\ \rightarrow ob(\mathtt{o},\sigma,\mathbf{idle},Q\cup\{\sigma''|x = \mathbf{get}(\mathtt{f});s\})\ fut(\mathtt{f},\bot)\\ ob(\mathtt{o}',\sigma',\{\sigma'''|s';\mathbf{cont}(\mathtt{f}')\},Q')\ cog(\mathtt{c},\mathtt{o}')\end{array}}$$

$$\frac{\begin{array}{c}\text{(Cog-Sync-Return-Sched)}\\ \sigma''(cog) = \mathtt{c} \quad \sigma'''(\mathbf{destiny}) = \mathtt{f}\end{array}}{\begin{array}{c}ob(\mathtt{o},\sigma,\{\sigma'|\mathbf{cont}(\mathtt{f})\},Q)\ cog(\mathtt{c},\mathtt{o})\ ob(\mathtt{o}',\sigma'',\mathbf{idle},Q'\cup\{\sigma'''|s\})\\ \rightarrow ob(\mathtt{o},\sigma,\mathbf{idle},Q)\ cog(\mathtt{c},\mathtt{o}')\ ob(\mathtt{o}',\sigma'',\{\sigma'''|s\},Q')\end{array}}$$

Figure 1.7: Reduction rules for the concurrent object level of Core ABS (3)

$$\frac{\sigma(\mathsf{nb}_{cr}) = 0 \quad \mathsf{o} = [\![e]\!]_{(\sigma\circ\sigma')} \quad v = [\![e']\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\ \sigma' \mid \textbf{rebind}\ e.f = e';\ s\ \},Q) \rightarrow ob(\mathsf{o},\sigma[f \mapsto v],\{\ \sigma' \mid s\ \},Q)}\ (\textsc{Rebind-Local})$$

$$\frac{\sigma(\mathsf{nb}_{cr}) = 0 \quad f \notin ports(\sigma(\textbf{class})) \quad \mathsf{o} = [\![e]\!]_{(\sigma\circ\sigma')} \quad v = [\![e']\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o},\sigma,\{\ \sigma' \mid \textbf{rebind}\ e.f = e';\ s\ \},Q) \rightarrow ob(\mathsf{o},\sigma,\{\ \sigma' \mid s\ \},Q)}\ (\textsc{Rebind-None})$$

$$\frac{\sigma_{\mathsf{o}}(\mathsf{nb}_{cr}) = 0 \quad \sigma_o(\textbf{cog}) = \sigma_{o'}(\textbf{cog}) \quad \mathsf{o} = [\![e]\!]_{(\sigma_{\mathsf{o}'}\circ\sigma'_{\mathsf{o}'})} \quad v = [\![e']\!]_{(\sigma_{\mathsf{o}'}\circ\sigma'_{\mathsf{o}'})}}{ob(\mathsf{o},\sigma_{\mathsf{o}},K_{\textbf{idle}},Q)\ ob(\mathsf{o}',\sigma_{\mathsf{o}'},\{\ \sigma'_{\mathsf{o}'} \mid \textbf{rebind}\ e.f = e';\ s\ \},Q') \rightarrow ob(\mathsf{o},\sigma_{\mathsf{o}}[f \mapsto v],K_{\textbf{idle}},Q)\ ob(\mathsf{o}',\sigma_{\mathsf{o}'},\{\ \sigma'_{\mathsf{o}'} \mid s\ \},Q')}\ (\textsc{Rebind-Global})$$

Figure 1.8: Reduction rules for **rebind**

```
interface Server { ... }
interface Client { Unit setServer(Server s); ... }

class Controller {
 Client c1, c2, ... cn;

 Unit updateServer(Server s2) {
  c1.setServer(s2);
  c2.setServer(s2);
      ...
  cn.setServer(s2);
 }
}
```

Figure 1.9: Workflow in ABS

```
interface Server { ... }
interface Client { port Server s; ... }

class Controller {
Client c1, c2, ... cn;
        ...
 Unit updateServer(Server s2) {
  await ||c1|| ∧ ||c2|| ∧ ... ∧ ||cn||;
  rebind c1.s = s2;
  rebind c2.s = s2;
        ...
  rebind cn.s = s2;
 }
}
```

Figure 1.10: Workflow using the Component Model

```
Unit main () {      ...
        Client c1 = new cog Client (s);
        Client c2 = new Client (s);
                    ...
        Client cn = new Client (s);
        Controller c = new Controller (c1, c2, ... cn);
}
```

Figure 1.11: Client and Controller objects creation

CHAPTER 2

A Type System for Components

In this chapter we present our type system for the component model. We first give a thorough explanation of the types we adopt and how the type system achieves the tracking of cog membership. Then, we introduce the subtyping relation, we present the auxiliary functions and predicates that the type system relies on, and we conclude with the typing rules.

2.1 Typing Features

In this section we give the intuition behind the types and the records used in the typing rules, the latter being a new concept not adopted either in ABS or in its component extension [75]. We explain also the meaning of the method signature and how the type system addresses the problem of consistent rebindings and consistent synchronous method calls.

Cog Names The goal of our type system is to statically check if rebindings and synchronous method calls are performed locally to a cog. Since cogs and objects are entities created at runtime, we cannot know statically their identity. The interesting and also difficult part in designing our type systems is how to statically track cogs identity and hence membership to a cog. We address this issue by using a *linear* type system on names of cogs, which range over $\mathsf{G}, \mathsf{G}', \mathsf{G}''$,

in a way that abstracts the runtime identity of cogs. The type system associates to every cog creation a unique cog name, which makes it possible to check if two objects are in the same cog or not.

Precisely, we associate objects to their cogs using records $\mathbb{r}$, having the form $\mathsf{G}[\overline{f : T}]$, where G denotes the cog in which the object is located and $[\overline{f : T}]$ maps any object's fields in $\overline{f}$ to its type in $\overline{T}$. In fact, in order to correctly track cog membership of each expression, we also need to keep information about the cog of the object's fields in a record. This is needed, for instance, when an object stored in a field is accessed within the method body and then returned by the method; in this case one needs a way to bind the cog of the accessed field to the cog of the returned value.

Cog Sets In order to deal with linearity of cogs created, and to keep track of them after their creation, our type system, besides the standard typing context Γ – which is formally defined in the next section – uses a set of cogs, ranged over by $\mathcal{G}, \mathcal{G}', \mathcal{G}''$, that keeps track of the cogs created so far and uses the operator $\uplus$ to deal with the disjoint union of set, namely $\mathcal{G} \uplus \mathcal{G}'$ etc, where the empty set acts as the neutral element, namely $\mathcal{G} \uplus \emptyset = \emptyset \uplus \mathcal{G} = \mathcal{G}$. In Section 2.4, we will show how the set of cogs is used in typing the terms of the calculus and how the disjoint union of cogsets works.

Method Signature Let us now explain the method signature $(\mathcal{G}, \mathbb{r})$ used in annotating a method header. The record $\mathbb{r}$ is used as the record of **this** during the typing of the method, i.e., $\mathbb{r}$ is the binder for the cog of the object **this** in the scope of the method body, as we will see in the typing rules in the following. The set of cog names $\mathcal{G}$ is used to keep track of the fresh cogs that the method creates. In particular, when we deal with recursive method calls, the set $\mathcal{G}$ gathers the fresh cogs of every call, which is then returned to the main execution. Moreover, when it is not necessary to keep track of cog information about an object, because the object is not going to take part in any synchronous method call or any rebind operation, it is possible to associate to this object the *unknown* record $\bot$. This special record does not keep any information about the cog where the object or its fields are located, and it is to be considered different from any other cog, thus to ensure the soundness of our type system.

Finally, note that data types also can contain records: for instance, a list of objects is typed with $\mathrm{List}\langle T \rangle$ where T is the type of the objects in the list and it includes also the records of the objects.

$$\frac{\forall i \quad T_i \leq T_i'}{D\langle \overline{T} \rangle \leq D\langle \overline{T}' \rangle}\ (\text{S-Data}) \qquad \frac{}{(L, \mathbb{r}) \leq (L, \bot)}\ (\text{S-Bot}) \qquad \frac{L \leq L' \in CT}{(L, \mathbb{r}) \leq (L', \mathbb{r})}\ (\text{S-Type})$$

$$\frac{\forall i \quad T_i \leq T_i' \qquad f \notin ports(L)}{(L, \mathtt{G}[f : T; \overline{f : T}]) \leq (L, \mathtt{G}[\overline{f : T'}])}\ (\text{S-Fields}) \qquad \frac{\forall i \quad T_i \leq T_i' \qquad f \in ports(L)}{(L, \mathtt{G}[\overline{f : T}]) \leq (L, \mathtt{G}[f : T; \overline{f : T'}])}\ (\text{S-Ports})$$

Figure 2.1: Subtyping Relation

2.2 Subtyping Relation

The subtyping relation $\leq$ on the syntactic types is a preorder and is presented in Fig. 2.1. For readability, we let L be either a class name `C` or an interface name `I`. We distinguish between two forms of subtyping, which is typical of object-oriented languages: *structural* and *nominal* subtyping. In a language where subtyping is nominal, A is a subtype of B if and only if it is declared to be so, meaning if class (or interface) A extends (or implements) class (or interface) B; these relations must be defined by the programmer and are based on the names of classes and interfaces declared. Java programmers are used to nominal subtyping, but other languages [42, 50, 79, 80, 93] are based on the structural approach. In the latter, subtyping relation is established only by analysing the structure of a class, i.e., its fields and methods: class (or interface) A is a subtype of class (or interface) B if and only if the fields and methods of A are a superset of the fields and methods of B, and their types in A are subtypes of their types in B. On the other hand, in [32] the authors integrate both the nominal and the structural subtyping in Featherweight Java-like calculus.

Rule (S-Data) states that data types are covariant in their type parameters. Rule (S-Bot) states that every record $\mathbb{r}$ is a subtype of the unknown record $\bot$. Rules (S-Fields) and (S-Ports) use structural subtyping on records. Fields, like methods, are what the object provides, hence it is sound to forget about the existence of a field in an object. This is why the rule (S-Fields) allows to remove fields from records. Ports on the other hand, model the *dependencies* the objects have on their environment, hence it is sound to consider that an object may have more

dependencies than it actually has during its execution. This is why the rule (S-Ports) allows to add ports to records. So, in case of fields, one object can be substituted by another one if the latter has at least the same fields; on the contrary, in case of ports, one object can be substituted by another one if the latter has at most the same ports. Notice that in the standard object-oriented setting this rule would not be sound, since trying to access a non-existing attribute would lead to a null pointer exception. Therefore, to support our vision of port behaviour, we add a (Rebind-None) reduction rule to the component calculus semantics which simply permits the rebind to succeed without modifying anything if the port is not available. Finally, rule (S-Type) adopts nominal subtyping between classes and interfaces and subtyping is again covariant.

2.3 Functions and Predicates

In this section we define the auxiliary functions and predicates that are used in the typing rules.

We first start with the lookup functions *params, ports, fields, ptype, mtype, heads* shown in Fig. 2.2. These functions are similar and are inspired by the corresponding ones in Featherweight Java [58]. For readability reasons, the lookup functions are written in italics, whether the auxiliary functions and predicates are not. Function *params* returns the sequence of typed parameters of a class. Function *ports* returns the sequence of typed ports. Instead, function $fields$ returns all the fields of the class it is defined on, namely the inner state and the ports too. Functions *ptype* and *mtype* return the declared type of respectively the port and the method they are applied to. Function *heads* returns the headers of the declared methods. Except function $fields$ which is defined only on classes, the rest of the lookup functions is defined on both classes and interfaces.

The auxiliary functions and predicates are shown in Fig. 2.3. Function tmatch returns a substitution σ of the formal parameters to the actual ones. It is defined both on types and on records. The matching of a type T to itself, or of a record $\mathbb{r}$ to itself, returns the identity substitution *id*; the matching of a type variable $\mathtt{V}$ to a type T returns a substitution of $\mathtt{V}$ to T; the matching of data type D parametrized on formal types $\overline{T}$ and on actual types $\overline{T'}$ returns the union of substitutions that correspond to the matching of each type T_i with T'_i in such a way that substitutions coincide when applied to the same formal types, the latter being expressed by $\forall i, j\ \sigma_{i|\mathsf{dom}(\sigma_j)} = \sigma_{j|\mathsf{dom}(\sigma_i)}$; the matching of records follows the same idea as

$$\frac{\textbf{class}\ \mathtt{C}\ (\overline{T\ x})\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl};\ \overline{M}\ \}}{params(\mathtt{C}) = \overline{T\ x}}$$

$$\frac{\textbf{class}\ \mathtt{C}\ [(\overline{T''\ x''})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{T'\ x'};\ \ \overline{M}\ \}}{ports(\mathtt{C}) = \overline{T\ x}}$$

$$\frac{\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}}{ports(\mathtt{I}) = \overline{T\ x}}$$

$$\frac{\textbf{class}\ \mathtt{C}\ [(\overline{T''\ x''})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{T'\ x'};\ \ \overline{M}\ \}}{fields(C) = \overline{T\ x}; \overline{T'\ x'}}$$

$$\frac{\textbf{class}\ \mathtt{C}\ [(\overline{T''\ x''})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{T'\ x'};\ \ \overline{M}\ \}}{ptype(p, \mathtt{C}) = \overline{T}}$$

$$\frac{\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}}{ptype(p, \mathtt{I}) = \overline{T}}$$

$$\frac{\begin{array}{c}\textbf{class}\ \mathtt{C}\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl}\ \ \overline{M}\ \} \\ [\textbf{critical}]\ (\mathcal{G}, \mathbb{r})\ T\ \mathtt{m}(\overline{T\ x})\{\ s\ \} \in \overline{M}\end{array}}{mtype(\mathtt{m}, \mathtt{C}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \rightarrow T}$$

$$\frac{\begin{array}{c}\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \} \\ [\textbf{critical}]\ (\mathcal{G}, \mathbb{r})\ T\ \mathtt{m}(\overline{T\ x}) \in \overline{S}\end{array}}{mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \rightarrow T}$$

$$\frac{\textbf{class}\ \mathtt{C}\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{\mathtt{I}}]\ \{\ \overline{Fl}\ \overline{M}\ \} \qquad \overline{M} = \overline{S\ \{s\}}}{heads(\mathtt{C}) = \overline{S}}$$

$$\frac{\textbf{interface}\ \mathtt{I}\ [\textbf{extends}\ \overline{\mathtt{I}}]\ \{\ \overline{\textbf{port}\ T\ x}; \overline{S}\ \}}{heads(\mathtt{I}) = \overline{S}}$$

Figure 2.2: Lookup Functions

that of data types. Finally, tmatch applied on types $(\mathtt{I}, \mathbb{r}), (\mathtt{I}, \mathbb{r}')$ returns the same substitution obtained by matching $\mathbb{r}$ with $\mathbb{r}'$. Function pmatch, performs matchings on patterns and types by returning a typing environment Γ. In particular, pmatch returns an empty set when the pattern is _ or **null**, or $x : T$ when applied on a variable x and a type T. Otherwise, if applied to a constructor expression $\mathtt{Co}(\overline{p})$ and a type T'' it returns the union of typing environments corresponding to patterns in $\overline{p}$. Function crec asserts that $(\mathtt{I}, \mathtt{G}[\sigma \uplus \sigma'(\overline{f : (\mathtt{I}, \mathbb{r})})])$ is a member of $\mathrm{crec}(\mathtt{G}, \mathtt{C}, \sigma)$ if class C implements interface I and σ' and σ are substitutions defined on disjoint sets of names. Predicate coloc states the equality of two cog names. Predicates implements and extends check when a class implements an interface and an interface extends another one properly. A class C implements an interface I if the ports of C are at *most* the ones of I. This follows the intuition, as already stated in the subtyping section: since ports indicate services then an object has at most the services declared in its interface. Then, any port in C has a type being a subtype of the type of the respective port in I. Instead, for methods, C may define at *least* the methods declared in I having the same signature. The extends predicate states when an interface I properly extends another interface I′ and it is defined similarly to the implements predicate.

2.4 Typing Rules

A *typing context* Γ is a partial function from names to typings, which assigns types T to variables, a pair $(\mathtt{C}, \mathbb{r})$ to **this**, and arrow types $\overline{T} \to T'$ to function symbols like Co or fun, namely:

$$\Gamma ::= \emptyset \mid x : T, \Gamma \mid \textbf{this} : (\mathtt{C}, \mathbb{r}), \Gamma \mid \mathtt{Co} : \overline{T} \to T', \Gamma \mid \mathtt{fun} : \overline{T} \to T', \Gamma$$

As usual $\mathsf{dom}(\Gamma)$ denote the domain of the typing context Γ, $\Gamma(\mathit{name})$ denotes the type of the *name* being x, **this**, Co or fun. We define the *composition* of typing contexts, by using the operator $\circ$ namely $\Gamma \circ \Gamma'$, as follows: $\Gamma \circ \Gamma'(x) = \Gamma'(x)$ if $x \in \mathsf{dom}(\Gamma')$, and $\Gamma \circ \Gamma'(x) = \Gamma(x)$ otherwise. We say that a typing context Γ' *extends* a typing context Γ, denoted with $\Gamma \subseteq \Gamma'$ if $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for all $x \in \mathsf{dom}(\Gamma)$. Typing judgements use a typing context Γ and possibly a cogset $\mathcal{G}$ which indicates the set of new cogs created by the term being typed. Typing judgements have the following forms: $\Gamma \vdash g : \mathtt{Bool}$ for guards, $\Gamma \vdash e : T$ for pure expressions, $\Gamma, \mathcal{G} \vdash z : T$ for expressions with side effects, $\Gamma, \mathcal{G} \vdash s$ for statements, $\Gamma, \mathcal{G} \vdash M$ for method declarations $\Gamma, \mathcal{G} \vdash C$ for class

declarations $\Gamma, \emptyset \vdash I$ for interface declarations and $\Gamma, \mathcal{G} \vdash P$ for programs.

Pure Expressions and Declarations The typing rules for pure expressions and their declarations are given in Fig. 2.4. Rule (T-VAR/FIELD) states that a variable is of type the one assumed in the typing context. Rule (T-NULL) states that the value **null** is of type any interface `I` declared in the CT (class table) and any record $\mathbb{r}$. Rule (T-WILD) states that the wildcard _ is of any type T. Rule (T-CONSEXP) states that the application of the constructor `Co` to a list of pure expressions $\overline{e}$ is of type $\sigma(T')$ whenever the constructor is of a functional type $\overline{T} \to T'$ and the pure expressions are of type $\overline{T'}$, and the the auxiliary function tmatch applied on the formal types $\overline{T}$ and the actual ones $\overline{T'}$ returns the substitution σ. Rule (T-FUNEXP) for function expressions is the same as the previous one for constructor expressions, namely, the application of the function `fun` to a list of pure expressions $\overline{e}$ is of type $\sigma(T')$ whenever the function is of a functional type $\overline{T} \to T'$ and the pure expressions are of type $\overline{T'}$, and the the auxiliary function tmatch applied on the formal types $\overline{T}$ and the actual ones $\overline{T'}$ returns the substitution σ. Rule (T-CASE) states that if all branches in $\overline{p \Rightarrow e_p}$ are well-typed with the same type, then the case expression is also well-typed with the return type of the branches. Rule (T-BRANCH) states that a branch $p \Rightarrow e_p$ is well-typed with an arrow type $T \to T'$ if the pattern p is well-typed with T and the expression e_p is well-typed with type T' in the composition of Γ with typing assertions for the pattern obtained by the function pmatch, previously defined. We comment now on the typing rules for the functional declarations. Rule (T-DATADECL) states that the data type declaration is well-typed whenever the constructors of the data type are of type the ones in the declaration. On the other hand, rule (T-CONSDECL) states that a constructor applied to a sequence of types $\overline{T}$ is of type T', if to the constructor name is assigned type $\overline{T} \to T'$ in the the typing context. Rule (T-FUNDECL) states that a function declaration is well-typed whenever the function name `fun` has type the one declared in the function declaration and the body of the function has type the one returned by the function. Rule (T-SUB) is the standard subsumption rule, where the subtyping relation is defined in Section 2.2.

Guards The typing rules for guards are given at the end of Fig. 2.4. Some comments follow. Rule (T-FUTGUARD) states that if a variable x has type `Fut`$\langle T \rangle$, the guard x? has type `Bool`. Rule (T-CRITICGUARD) states that $\|x\|$ has type `Bool` if x is an object, namely having type $(\mathtt{I}, \mathbb{r})$. Rule (T-CONJGUARD) states that if each g_i has type `Bool` for $i = 1, 2$ then the conjunction $g_1 \wedge g_2$ has also type `Bool`.

Expressions with Side Effects The typing rules for expressions with side effects are given in Fig. 2.5. As already stated at the beginning of the section, these typing rules are different wrt the typing rules for pure expressions, as they keep track of the new cogs created. Rule (T-Exp) is a weakening rule which asserts that a pure expression e is well-typed in a typing context Γ and an empty set of cogs, if it is well-typed in Γ. Rule (T-Get) states that **get**(e) is of type T, if expression e is of type $\texttt{Fut}\langle T\rangle$. Rule (T-New) assigns type T to the object **new** $\texttt{C}(\overline{e})$ if the actual parameters have types compatible with the formal ones, by applying function tmatch; the new object and **this** have the same cog $\texttt{C}$ and the type T belongs to the crec($\texttt{G}, \texttt{C}, \sigma$) predicate, which means that T is of the form $(\texttt{I}, \texttt{G}[\overline{f : \sigma(\texttt{I}, \mathbb{r})}])$ and implements($\texttt{C}, \texttt{I}$) and σ is obtained by the function tmatch. Rule (T-Cog) is similar to the previous one, except for the creation of a new cog $\texttt{G}$ where the new object is placed, and hence the group of object **this** is not checked. Rules (T-SCall) and (T-ACall) type synchronous and asynchronous method calls, respectively. Both rules use function *mtype* to obtain the method signature as well as the method's typed parameters and the return type, i.e., $(\mathcal{G}, \mathbb{r})(\overline{T\ x}) \rightarrow T$. The group record $\mathbb{r}$, the parameters types and the return type of the method are the "formal" ones. In order to obtain the "actual" ones, we use the substitution σ that maps formal cog names to actual cog names. Consequently, the callee e has type $(\texttt{I}, \sigma(\mathbb{r}))$ and the actual parameters $\overline{e}$ have types $\overline{\sigma(T)}$. Finally, the invocations are typed in the substitution $\sigma(T)$, where T is the "formal" return type. The rules differ in that the former also checks whether the group of **this** and the group of the callee coincide, by using the auxiliary function coloc, and also the types of the returned value are $\sigma(T)$ and $\texttt{Fut}\langle\sigma(T)\rangle$, respectively.

Statements The typing rules for statements are presented in Fig. 2.6. Rules (T-Skip) and (T-Suspend) state that **skip** and **suspend** are always well-typed. Rule (T-Decl) states that $T\ x$ is well-typed if variable x is of type T in Γ. Rule (T-Comp) states that, if s_1 and s_2 are well-typed in the same typing context and, like in linear type systems, they use distinct sets of cogs, then their composition is well-typed and uses the disjoint union $\uplus$ of the corresponding cogsets. Rule (T-Assign) asserts the well-typedness of the assignment $x = z$ if both x and z have the same type T and the set of cogs is the one corresponding to the expression z. Rule (T-Await) asserts that **await** g is well-typed whenever the guard g has type $\texttt{Bool}$. Rules (T-Cond) and (T-While) are quite standard, except for the presence of the linear set of cog names: in typing the conditional statement, we use a set $\mathcal{G}$ of cogs in typing both its branches, as only one of the statements will be executed;

and in the loop statement we use the set of cogs corresponding to the loop body. It is important to notice that the set of cogs associated to the **while** statement is the same as the set of cogs associated to its body s. At first, this may seem surprising since the statement s during runtime is executed a number of times depending on the truth value of the loop's guard, and this truth value is not know before the execution takes places. However, it does not matter how many new cogs are created and assigned inside the loop because what matters is the last execution of the statement and the last cogs assignment, which overwrites all the previous ones. Rule (T-RETURN) asserts that **return** e is well-typed if expression e has type T whether the variable **destiny** has type $\texttt{Fut}\langle T\rangle$. Finally, rule (T-REBIND) well-types the statement **rebind** $e.p = z$ by checking that: *i*) p is a port of the right type, *ii*) z has the same type as the port, and *iii*) the object stored in e and the current one **this** are in the same cog, by using the predicate $\text{coloc}(\mathbb{r}, \Gamma(\textbf{this}))$.

Declarations The typing rules for declarations of methods, classes, interfaces and programs are presented in Fig. 2.7. Rule (T-METHOD) states that method $\texttt{m}$ is well-typed in class $\texttt{C}$ if the method's body s is well-typed in a typing context augmented with the method's typed parameters $\overline{x : T}$; **destiny** being of type $\texttt{Fut}\langle T\rangle$ and the current object **this** being of type $(\texttt{C}, \mathbb{r})$; and cog names as specified by the method signature. Rule (T-CLASS) states that a class $\texttt{C}$ is well-typed when it implements all the interfaces $\overline{\texttt{I}}$ and all its methods are well-typed. The set of cogs produced by the method declarations is propagated to the class declaration. Rule (T-INTERFACE) states that an interface $\texttt{I}$ is well-typed if it extends all interfaces in $\overline{\texttt{I}}$. Finally, rule (T-PROGRAM) states that a program is well-typed if all the declarations in the program are well-typed and also the body s is well-typed. Again, the set of cogs of the program is the disjoint union of all sets of cogs present in the declarations and in the body s.

Remark The typing rule for assignment requires the group of the variable and the group of the expression being assigned to be the same. This restriction applies to rule for rebinding, as well. To see why this is needed let us consider a sequence of two asynchronous method invocations $x!\texttt{m}(); x!\texttt{n}()$, both called on the same object and both modifying the same field. Say $\texttt{m}$ does $\textbf{this}.\texttt{f} = \texttt{z}_1$ and $\texttt{n}$ does $\textbf{this}.\texttt{f} = \texttt{z}_2$. Because of asynchronicity, there is no way to know the order in which the updates will take place at runtime. A similar example may be produced for the case of rebinding. Working statically, we can either force the two expressions $\texttt{z}_1$ and $\texttt{z}_2$ to have the same group as $\texttt{f}$, or keep track of all the different possibilities,

thus the type system must assume for an expression a set of possible objects it can reduce to. In this work we adopt the former solution, we let the exploration of the latter as a future work. We plan to relax this restriction following a similar idea to the one proposed in [49], where a set of groups can be associated to a variable instead of just only one group.

Example Revisited We now recall the example of the workflow given in Fig. 1.10 and Fig. 1.11. We show how the type system works on this example: by applying the typing rule for **rebind** we have the derivation in Fig. 2.8 for any clients from $\mathtt{c}_2$ to $\mathtt{c}_n$. Let us now try to typecheck client $\mathtt{c}_1$. If we try to typecheck the rebinding operation, we would have the following typing judgement in the premise of (T-Rebind):

$$\Gamma(\textbf{this}) = (\mathtt{Controller}, \mathtt{G}[...]) \qquad \Gamma, \emptyset \vdash \mathtt{c}_1 : (\mathtt{Client}, \mathtt{G}'[\ldots, s : (\mathtt{Server}, \mathbb{r})])$$

But then, the predicate $\mathrm{coloc}(\mathtt{G}'[\ldots, s : (\mathtt{Server}, \mathbb{r})], \Gamma(\textbf{this}))$ is false, since $\mathrm{equals}(\mathtt{G}, \mathtt{G}')$ is false. Then, one cannot apply the typing rule (T-Rebind), by thus not typechecking **rebind** $\mathtt{c}_1.\mathtt{s} = \mathtt{s2}$, exactly as we were aiming to.

2.5 Typing Rules for Runtime Configurations

In this section we present the typing rules for runtime configurations, the latter have been introduced in Section 1.2. In order to prove theoretical results about our type system, such as the subject reduction property, typing rules for runtime configurations are needed. We present them in Fig. 2.9.

Runtime typing judgements are of the form $\Delta, \mathcal{G} \vdash_R N$ meaning that the configuration N is well-typed in the typing context Δ by using a set $\mathcal{G}$ of new cogs. The (runtime) typing context Δ is an *extension* of the (compile time) typing context Γ with runtime information about objects, futures and cogs and is formally defined as follows:

$$\Delta ::= \emptyset \mid \Gamma, \Delta \mid \mathtt{o} : (\mathtt{C}, \mathbb{r}), \Delta \mid \mathtt{f} : \mathtt{Fut}\langle T \rangle, \Delta \mid \mathtt{c} : \textbf{cog}, \Delta$$

An object identifier $\mathtt{o}$ is given type $(\mathtt{C}, \mathbb{r})$ where $\mathtt{C}$ is the class the object is instantiating and $\mathbb{r}$ is the group record containing group information about the object itself and the object's fields. Notice that $(\mathtt{C}, \mathbb{r})$ is not a type produced by the grammar of types given in Section 1.1, however it is needed to type object identifiers. (As we

will see in the subject reduction theorem that follows, C is the class implementing I, the latter being the type of the expression which evaluates to o.) A future value f is assigned type future Fut⟨T⟩ and a cog identifier c is merely assigned the keyword **cog**, since cog names have no types.

Rules (T-Weak1), (T-Weak2) and (T-Weak3) state respectively that when an expression is of type T in some typing context Γ, then it has the same type in Δ, which is an extension of Γ; and whenever a statement s or a declaration Dl is well-typed in Γ, then it is also well-typed in Δ, which is an extension of Γ. Rule (T-State) asserts that the substitution of variable x with value v is well-typed when x and v have the same type T. Rule (T-Cont) asserts that the statement **cont**(f), which is a new statement added to the runtime syntax, is well-typed whenever f is a future. As previously stated, we adopt two different sequential compositions, denoted by ';' for the standard sequential composition and ';;' for the loop composition. Rule (T-OtherComp) is basically the typing rule for the **while** composition. It states that the composition s ;; s' is well-typed under a set $\mathcal{G}$ of cogs, if the first statement s uses $\mathcal{G}$ and the second statement s' uses some other set $\mathcal{G}'$. Intuitively, this typing rule "forgets" about the creation of new cogs in the second statement, and typechecks the composition by "remembering" only the cogs created by the first statement. When typechecking the loop statement, no matter how many times the loop is executed, the last execution and hence the last creation of cogs overwrites all the previous ones. Rule (T-Future1) states that the configuration *fut*(f, v) is well-typed if the future f has type Fut⟨T⟩ where T is the type of v. Instead, rule (T-Future2) states that *fut*(f, $\perp$) is well-typed whenever f is a future. Rule (T-Process-Queue) states that the union of two queues is well-typed if both queues are well-typed and the set of cogs is obtained as a disjoint union of the two sets of cogs corresponding to each queue. Rule (T-Process) states that a task or a process is well-typed if its local variables $\overline{x}$ are well-typed and statement s is well-typed in a typing context augmented with typing information about the local variables and the set of cogs $\mathcal{G}$. Rule (T-Config) states that the composition $N\ N'$ of two configurations is well-typed whenever N and N' are well-typed using disjoint set of cog names. Rule (T-Cog) asserts that a group configuration *cog*(c, o_ϵ) is well-typed if c is declared to be a **cog** in Δ. Rules (T-Empty) and (T-Idle) assert the well-typedness of the ϵ configuration and **idle** process, respectively. Rule (T-Object) states that an object is well-typed whenever: *i*) the declared group record of o corresponds to its actual structure; *ii*) all its fields are well-typed; and *iii*) its running process and process queue are also well-typed. Finally, (T-Invoc) states that *invoc*(o, f, m, $\overline{v}$) is well-typed under some substitution σ when the fol-

lowing premises hold: *i*) callee `o` is assigned type $(\mathtt{C}, \sigma(\mathbb{r}))$; *ii*) future `f` is of type $\mathtt{Fut}\langle\sigma(T)\rangle$, and *iii*) values $\overline{v}$ are typed accordingly to the type and signature by applying substitution σ, namely $\overline{\sigma(T)}$.

$$\text{tmatch}(T, T) = id \qquad \text{tmatch}(\mathbb{r}, \mathbb{r}) = id \qquad \text{tmatch}(\mathtt{V}, T) \triangleq [\mathtt{V} \mapsto T]$$

$$\frac{\forall i \quad \text{tmatch}(T_i, T_i') = \sigma_i \qquad \forall i, j \quad \sigma_{i|\mathsf{dom}(\sigma_j)} = \sigma_{j|\mathsf{dom}(\sigma_i)}}{\text{tmatch}(D\langle\overline{T}\rangle, D\langle\overline{T'}\rangle) \triangleq \bigcup_i \sigma_i}$$

$$\frac{\text{tmatch}(\mathbb{r}, \mathbb{r}') = \sigma}{\text{tmatch}((\mathtt{I}, \mathbb{r}), (\mathtt{I}, \mathbb{r}')) \triangleq \sigma}$$

$$\frac{\forall i \quad \text{tmatch}(T_i, T_i') = \sigma_i \qquad \forall i, j \quad \sigma_{i|\mathsf{dom}(\sigma_j)} = \sigma_{j|\mathsf{dom}(\sigma_i)} \qquad \sigma(\mathtt{G}) \in \{\mathtt{G}, \mathtt{G}'\}}{\text{tmatch}(\mathtt{G}[\overline{f : T}], \mathtt{G}'[\overline{f : T'}]) \triangleq [\mathtt{G} \mapsto \mathtt{G}'] \bigcup_i \sigma_i}$$

$$\text{pmatch}(\_, T) \triangleq \emptyset \qquad \text{pmatch}(x, T) \triangleq \emptyset; x : T \qquad \text{pmatch}(\textbf{null}, (\mathtt{I}, \mathbb{r})) \triangleq \emptyset$$

$$\frac{\begin{array}{c}\Gamma(\mathtt{Co}) = \overline{T} \to T' \\ \text{tmatch}(T', T'') = \sigma \qquad \forall i \quad \text{pmatch}(p_i, \sigma(T_i)) = \Gamma_i\end{array}}{\text{pmatch}(\mathtt{Co}(\overline{p}), T'') \triangleq \biguplus_i \Gamma_i}$$

$$\frac{\begin{array}{c}\mathtt{C} \le \mathtt{I} \in CT \qquad \mathsf{dom}(\sigma') \cap \mathsf{dom}(\sigma) = \emptyset \\ \mathit{fields}(\mathtt{C}) = \overline{(\mathtt{I}, \mathbb{r})\ f}; \overline{\mathtt{D}(\ldots)\ f'}\end{array}}{(\mathtt{I}, \mathtt{G}[\overline{f : \sigma \circ \sigma'(\mathtt{I}, \mathbb{r})}]) \in \text{crec}(\mathtt{G}, \mathtt{C}, \sigma)}$$

$$\frac{\text{equals}(\mathtt{G}, \mathtt{G}')}{\text{coloc}(\mathtt{G}[\ldots], (\mathtt{C}, \mathtt{G}'[\ldots]))}$$

$$\frac{\begin{array}{c}\mathit{ports}(\mathtt{C}) \subseteq \mathit{ports}(\mathtt{I}) \text{ and } \forall p \in \mathit{ports}(\mathtt{C}).\ \mathit{ptype}(p, \mathtt{C}) \le \mathit{ptype}(p, \mathtt{I}) \\ \mathit{heads}(\mathtt{I}) \subseteq \mathit{heads}(\mathtt{C}) \text{ and } \forall \mathtt{m} \in \mathtt{I}.\ \mathit{mtype}(\mathtt{m}, \mathtt{I}) = \mathit{mtype}(\mathtt{m}, \mathtt{C})\end{array}}{\text{implements}(\mathtt{C}, \mathtt{I})}$$

$$\frac{\begin{array}{c}\mathit{ports}(\mathtt{I}) \subseteq \mathit{ports}(\mathtt{I}') \text{ and } \forall p \in \mathit{ports}(\mathtt{I}).\ \mathit{ptype}(p, \mathtt{I}) \le \mathit{ptype}(p, \mathtt{I}') \\ \mathit{heads}(\mathtt{I}') \subseteq \mathit{heads}(\mathtt{I}) \text{ and } \forall \mathtt{m} \in \mathtt{I}'.\ \mathit{mtype}(\mathtt{m}, \mathtt{I}) = \mathit{mtype}(\mathtt{m}, \mathtt{I}')\end{array}}{\text{extends}(\mathtt{I}, \mathtt{I'})}$$

Figure 2.3: Auxiliary Functions and Predicates

$$\text{(T-Var/Field)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-Null)}\ \frac{\textbf{interface}\ \mathtt{I}\ [\cdots]\ \{\ \cdots\ \} \in CT}{\Gamma \vdash \textbf{null} : (\mathtt{I}, \mathbb{r})} \qquad \text{(T-Wild)}\ \frac{}{\Gamma \vdash \_ : T}$$

$$\text{(T-ConsExp)}\ \frac{\Gamma(\mathtt{Co}) = \overline{T} \to T' \quad \text{tmatch}(\overline{T}, \overline{T'}) = \sigma \quad \Gamma \vdash \overline{e} : \overline{T'}}{\Gamma \vdash \mathtt{Co}(\overline{e}) : \sigma(T')} \qquad \text{(T-FunExp)}\ \frac{\Gamma(\mathtt{fun}) = \overline{T} \to T' \quad \text{tmatch}(\overline{T}, \overline{T'}) = \sigma \quad \Gamma \vdash \overline{e} : \overline{T'}}{\Gamma \vdash \mathtt{fun}(\overline{e}) : \sigma(T')}$$

$$\text{(T-Case)}\ \frac{\Gamma \vdash e : T \quad \Gamma \vdash \overline{p \Rightarrow e_p} : T \to T'}{\Gamma \vdash \textbf{case}\ e\ \{\overline{p \Rightarrow e_p}\} : T'} \qquad \text{(T-Branch)}\ \frac{\Gamma \vdash p : T \quad \Gamma \circ \text{pmatch}(p, T) \vdash e_p : T'}{\Gamma \vdash p \Rightarrow e_p : T \to T'}$$

$$\text{(T-DataDecl)}\ \frac{\Gamma \vdash \mathtt{Co}[(\overline{T})] | \overline{\mathtt{Co}[(\overline{T})]} : \mathtt{D}[\langle \overline{T} \rangle]}{\Gamma \vdash \textbf{data}\ \mathtt{D}[\langle \overline{T} \rangle] = \mathtt{Co}[(\overline{T})] | \overline{\mathtt{Co}[(\overline{T})]};} \qquad \text{(T-ConsDecl)}\ \frac{\Gamma(\mathtt{Co}) = \overline{T} \to T'}{\Gamma \vdash \mathtt{Co}(\overline{T}) : T'}$$

$$\text{(T-FunDecl)}\ \frac{\Gamma(\mathtt{fun}) = \overline{T} \to T' \quad \Gamma, \overline{x} : \overline{T} \vdash e : T'}{\Gamma \vdash \textbf{def}\ T\ \mathtt{fun}[\langle \overline{T} \rangle](\overline{T\ x}) = e;} \qquad \text{(T-Sub)}\ \frac{\Gamma \vdash e : T \quad T \leq T'}{\Gamma \vdash e : T'}$$

$$\text{(T-FutGuard)}\ \frac{\Gamma \vdash x : \mathtt{Fut}\langle T \rangle}{\Gamma \vdash x? : \mathtt{Bool}} \qquad \text{(T-CriticGuard)}\ \frac{\Gamma \vdash x : (\mathtt{I}, \mathbb{r})}{\Gamma \vdash \|x\| : \mathtt{Bool}} \qquad \text{(T-ConjGuard)}\ \frac{\Gamma \vdash g_1 : \mathtt{Bool} \quad \Gamma \vdash g_2 : \mathtt{Bool}}{\Gamma \vdash g_1 \wedge g_2 : \mathtt{Bool}}$$

Figure 2.4: Typing Rules for the Functional Level

$$\text{(T-Exp)}\ \frac{\Gamma \vdash e : T}{\Gamma, \emptyset \vdash e : T} \qquad \text{(T-Get)}\ \frac{\Gamma \vdash e : \texttt{Fut}\langle T \rangle}{\Gamma \vdash \textbf{get}(e) : T}$$

$$\text{(T-New)}\ \frac{\Gamma(\textbf{this}) = (\texttt{C}', \texttt{G}[\ldots]) \qquad \mathit{params}(\texttt{C}) = \overline{T\ x} \qquad \Gamma \vdash \overline{e : T'} \qquad \text{tmatch}(\overline{T}, \overline{T'}) = \sigma \qquad T \in \text{crec}(\texttt{G}, \texttt{C}, \sigma)}{\Gamma \vdash \textbf{new}\ \texttt{C}(\overline{e}) : T}$$

$$\text{(T-Cog)}\ \frac{\mathit{params}(\texttt{C}) = \overline{T\ x} \qquad \Gamma \vdash \overline{e : T'} \qquad \text{tmatch}(\overline{T}, \overline{T'}) = \sigma \qquad T \in \text{crec}(\texttt{G}, \texttt{C}, \sigma)}{\Gamma, \{\texttt{G}\} \vdash \textbf{new cog}\ \texttt{C}\ (\overline{e}) : T}$$

$$\text{(T-SCall)}\ \frac{\mathit{mtype}(\texttt{m}, \texttt{I}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \to T \qquad \Gamma \vdash e : (\texttt{I}, \sigma(\mathbb{r})) \qquad \Gamma \vdash \overline{e} : \overline{\sigma(T)} \qquad \text{coloc}(\sigma(\mathbb{r}), \Gamma(\textbf{this}))}{\Gamma \vdash e.m(\overline{e}) : \sigma(T)}$$

$$\text{(T-ACall)}\ \frac{\mathit{mtype}(\texttt{m}, \texttt{I}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \to T \qquad \Gamma \vdash e : (\texttt{I}, \sigma(\mathbb{r})) \qquad \Gamma \vdash \overline{e} : \overline{\sigma(T)}}{\Gamma \vdash e!m(\overline{e}) : \texttt{Fut}\langle \sigma(T) \rangle}$$

Figure 2.5: Typing Rules for Expressions with Side Effects

$$\text{(T-Skip)}\ \frac{}{\Gamma, \emptyset \vdash \textbf{skip}} \qquad \text{(T-Suspend)}\ \frac{}{\Gamma, \emptyset \vdash \textbf{suspend}} \qquad \text{(T-Decl)}\ \frac{\Gamma(x) = T}{\Gamma, \emptyset \vdash T\ x} \qquad \text{(T-Comp)}\ \frac{\Gamma, \mathcal{G}_1 \vdash s_1 \qquad \Gamma, \mathcal{G}_2 \vdash s_2}{\Gamma, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash s_1; s_2}$$

$$\text{(T-Assign)}\ \frac{\Gamma(x) = T \qquad \Gamma, \mathcal{G} \vdash z : T}{\Gamma, \mathcal{G} \vdash x = z} \qquad \text{(T-Await)}\ \frac{\Gamma \vdash g : \texttt{Bool}}{\Gamma, \emptyset \vdash \textbf{await}\ g}$$

$$\text{(T-Cond)}\ \frac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma, \mathcal{G} \vdash s_1 \qquad \Gamma, \mathcal{G} \vdash s_2}{\Gamma, \mathcal{G} \vdash \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2} \qquad \text{(T-While)}\ \frac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma, \mathcal{G} \vdash s}{\Gamma, \mathcal{G} \vdash \textbf{while}\ e\ \{\ s\ \}}$$

$$\text{(T-Return)}\ \frac{\Gamma \vdash e : T \qquad \Gamma(\textbf{destiny}) = \texttt{Fut}\langle T \rangle}{\Gamma, \emptyset \vdash \textbf{return}\ e} \qquad \text{(T-Rebind)}\ \frac{T\ p \in \mathit{ports}(\texttt{I}) \qquad \Gamma \vdash e : (\texttt{I}, \mathbb{r}) \qquad \Gamma, \mathcal{G} \vdash z : T \qquad \mathrm{coloc}(\mathbb{r}, \Gamma(\textbf{this}))}{\Gamma, \mathcal{G} \vdash \textbf{rebind}\ e.p = z}$$

Figure 2.6: Typing Rules for Statements

$$\text{(T-Method)}\ \frac{\Gamma, \overline{x : T}, \textbf{destiny} : \texttt{Fut}\langle T \rangle, \textbf{this} : (\texttt{C}, \mathbb{r}), \mathcal{G} \vdash s}{\Gamma, \mathcal{G} \vdash [\textbf{critical}]\ (\mathcal{G}, \mathbb{r})\ T\ \mathtt{m}(\overline{T\ x})\{\ s\ \}\ \mathit{in}\ \texttt{C}}$$

$$\text{(T-Class)}\ \frac{\forall \texttt{I} \in \overline{\texttt{I}}.\ \mathrm{implements}(\texttt{C}, \texttt{I}) \qquad \Gamma, \overline{x} : \overline{T}, \overline{\mathcal{G}} \vdash \overline{M}\ \mathit{in}\ \texttt{C}}{\Gamma, \overline{\mathcal{G}} \vdash \textbf{class}\ \texttt{C}\ (\overline{T\ x})\ \textbf{implements}\ \overline{\texttt{I}}\ \{\ \overline{Fl}\ \overline{M}\ \}}$$

$$\text{(T-Interface)}\ \frac{\forall \texttt{I}' \in \overline{\texttt{I}}.\ \mathrm{extends}(\texttt{I}, \texttt{I}')}{\Gamma, \emptyset \vdash \textbf{interface}\ \texttt{I}\ \textbf{extends}\ \overline{\texttt{I}}\ \{\ \overline{\textbf{port}\ T\ x; S}\ \}} \qquad \text{(T-Program)}\ \frac{\Gamma, \overline{\mathcal{G}} \vdash \overline{Dl} \qquad \Gamma, \mathcal{G} \vdash s}{\Gamma, \overline{\mathcal{G}} \uplus \mathcal{G} \vdash \overline{Dl}\ \{s\}}$$

Figure 2.7: Typing Rules for Declarations

$$\text{(T-Rebind)}\quad \frac{\Gamma(\textbf{this}) = (\texttt{Controller}, \texttt{G}[\ldots]) \qquad (\texttt{Server}, \mathbb{r})\ s \in \mathit{ports}(\texttt{Client}) \quad \forall i = 2, ..., n \quad \Gamma \vdash \texttt{c}_i : (\texttt{Client}, \texttt{G}[\ldots, s : (\texttt{Server}, \mathbb{r})]) \quad \Gamma, \emptyset \vdash s2 : (\texttt{Server}, \mathbb{r}) \qquad \mathrm{coloc}(\texttt{G}[\ldots, s : (\texttt{Server}, \mathbb{r})], \Gamma(\textbf{this}))}{\forall i\ \Gamma, \emptyset \vdash \textbf{rebind}\ \texttt{c}_i.s = s2}$$

Figure 2.8: Typing the Workflow Example

$$\text{(T-Weak1)}\quad \frac{\Gamma, \mathcal{G} \vdash z : T \qquad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R z : T} \qquad \text{(T-Weak2)}\quad \frac{\Gamma, \mathcal{G} \vdash s \qquad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R s} \qquad \text{(T-Weak3)}\quad \frac{\Gamma, \mathcal{G} \vdash Dl \qquad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R Dl} \qquad \text{(T-State)}\quad \frac{\Delta(x) = T \qquad \Delta \vdash_R v : T}{\Delta, \emptyset \vdash_R T\ x\ v} \qquad \text{(T-Cont)}\quad \frac{\Delta(\texttt{f}) = \texttt{Fut}\langle T \rangle}{\Delta, \emptyset \vdash_R \textbf{cont}(\texttt{f})}$$

$$\text{(T-OtherComp)}\quad \frac{\Delta, \mathcal{G} \vdash_R s \qquad \Delta, \mathcal{G}' \vdash_R s'}{\Delta, \mathcal{G} \vdash_R s\ ;;\ s'} \qquad \text{(T-Future1)}\quad \frac{\Delta(\texttt{f}) = \texttt{Fut}\langle T \rangle \qquad \Delta \vdash_R v : T}{\Delta, \emptyset \vdash_R \mathit{fut}(\texttt{f}, v)} \qquad \text{(T-Future2)}\quad \frac{\Delta(\texttt{f}) = \texttt{Fut}\langle T \rangle}{\Delta, \emptyset \vdash_R \mathit{fut}(\texttt{f}, \bot)} \qquad \text{(T-Process-Queue)}\quad \frac{\Delta, \mathcal{G} \vdash_R Q \qquad \Delta, \mathcal{G}' \vdash_R Q'}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R Q \cup Q'}$$

$$\text{(T-Process)}\quad \frac{\Delta, \emptyset \vdash_R \overline{T}\ \overline{x}\ \overline{v} \qquad \Delta, \overline{x} : \overline{T}, \mathcal{G} \vdash_R s}{\Delta, \mathcal{G} \vdash_R \{\ \overline{T}\ \overline{x}\ \overline{v} \mid s\ \}} \qquad \text{(T-Config)}\quad \frac{\Delta, \mathcal{G} \vdash_R N \qquad \Delta, \mathcal{G}' \vdash_R N'}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R N\ N'} \qquad \text{(T-Cog)}\quad \frac{\Delta(\texttt{c}) = \textbf{cog}}{\Delta, \emptyset \vdash_R \mathit{cog}(\texttt{c}, \texttt{o}_\varepsilon)} \qquad \text{(T-Empty)}\quad \frac{}{\Delta, \emptyset \vdash_R \epsilon}$$

$$\text{(T-Idle)}\quad \frac{}{\Delta, \emptyset \vdash_R \textbf{idle}} \qquad \text{(T-Object)}\quad \frac{\Delta(\texttt{o}) = (\texttt{C}, \texttt{G}[\overline{f : T}]) \quad \mathit{fields}(\texttt{C}) = \overline{T}\ \overline{f} \qquad \Delta, \overline{f} : \overline{T}, \emptyset \vdash_R \overline{T}\ \overline{f}\ \overline{v} \quad \Delta, \overline{f} : \overline{T}, \mathcal{G} \vdash_R K_{\textbf{idle}} \qquad \Delta, \overline{f} : \overline{T}, \mathcal{G}' \vdash_R Q}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R \mathit{ob}(\texttt{o}, \overline{T}\ \overline{f}\ \overline{v};\ \theta, K_{\textbf{idle}}, Q)}$$

$$\text{(T-Invoc)}\quad \frac{\mathit{mtype}(\mathtt{m}, \texttt{C}) = (\mathcal{G}, \mathbb{r})(\overline{T\ x}) \to T \qquad \Delta(\texttt{o}) = (\texttt{C}, \sigma(\mathbb{r})) \quad \Delta(\texttt{f}) = \texttt{Fut}\langle \sigma(T) \rangle \qquad \Delta \vdash_R \overline{v} : \overline{\sigma(T)}}{\Delta, \emptyset \vdash_R \mathit{invoc}(\texttt{o}, \texttt{f}, \mathtt{m}, \overline{v})}$$

Figure 2.9: Typing Rules for Runtime Configurations

CHAPTER 3

Properties of the Type System

3.1 Properties of the type system

In this section we present the main properties of our type system. We start with Type Preservation of the evaluation of expressions, then we present Subject Reduction of the reduction of configurations and we conclude with the Correction Theorem, the main result of this part, intuitively stating that well-typed programs do not perform illegal rebinding or synchronous method calls.

A substitution σ is well-typed in a typing context Γ, denoted by $\Gamma \vdash \sigma$, if $\Gamma \vdash \sigma(x) : \Gamma(x)$ for all $x \in \mathsf{dom}(\sigma)$. Recall that a typing context Γ' *extends* a typing context Γ, denoted with $\Gamma \subseteq \Gamma'$ if $\mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for all $x \in \mathsf{dom}(\Gamma)$.

Lemma 3.1.1 (Type preservation)**.** *Let Γ be a typing environment and σ a substitution such that $\Gamma \vdash \sigma$. If $\Gamma \vdash e : T$ and $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$, then there is a typing environment Γ' such that $\Gamma \subseteq \Gamma$, $\Gamma' \vdash \sigma'$ and $\Gamma' \vdash e' : T$.*

The types system is proven correct in a Wright-Felleisen style [116], namely we prove Subject Reduction property stating that if a well-typed configuration N reduces to some configuration N' then, the latter configuration is also well-typed.

Theorem 3.1.2 (Subject Reduction)**.** *If $\Delta, \mathcal{G} \vdash_R N$ and $N \rightarrow N'$ then $\exists\, \Delta'$, $\mathcal{G}'$ such that $\Delta' \supseteq \Delta$, $\mathcal{G}' \subseteq \mathcal{G}$ and $\Delta', \mathcal{G}' \vdash_R N'$.*

Theorem 3.1.3 (Correction of rebinding)**.** *If* $\Delta, \mathcal{G} \vdash_R N$*, then for all objects* $ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q) \in N$ *and* $s \equiv$ **rebind** $e.f_i = e'; s'$ *there exists an object* $ob(\mathsf{o}', \sigma'', K_{\mathbf{idle}}, Q') \in N$ *such that* $[\![e]\!]_{(\sigma \circ \sigma')} = \mathsf{o}'$ *and* $\sigma(\mathbf{cog}) = \sigma''(\mathbf{cog})$.

Theorem 3.1.4 (Correction of method call)**.** *If* $\Delta, \mathcal{G} \vdash_R N$*, then for all objects* $ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q) \in N$ *and* $s \equiv x = e.\mathtt{m}(\overline{e}); s'$ *there exists an object* $ob(\mathsf{o}', \sigma'', K_{\mathbf{idle}}, Q') \in N$ *such that* $[\![e]\!]_{(\sigma \circ \sigma')} = \mathsf{o}'$ *and* $\sigma(\mathbf{cog}) = \sigma''(\mathbf{cog})$.

As a consequence of the previous results, rebinding and synchronous method calls are always performed between objects of the same cog:

Corollary 3.1.5. *Well-typed programs do not perform i) an illegal rebinding or ii) a synchronous method call outside the cog.*

3.2 Proofs of properties

In this section we give the detailed proofs of the previous lemmas and theorems that validate our type system. We state the following auxiliary lemma needed to prove the former properties.

Lemma 3.2.1 (Weakening)**.** *If* $\Delta, \mathcal{G} \vdash_R N$*, then* $\Delta', \mathcal{G} \vdash_R N$*, where* $\Delta \subseteq \Delta'$.

Proof. The proof follows immediately by the definition of Δ and and the typing judgements $\Delta, \mathcal{G} \vdash_R N$. □

Lemma 3.2.2 (Type preservation)**.** *Let* Γ *be a typing environment and* σ *a substitution such that* $\Gamma \vdash \sigma$*. If* $\Gamma \vdash e : T$ *and* $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$*, then there is a typing environment* Γ' *such that* $\Gamma \subseteq \Gamma$*,* $\Gamma' \vdash \sigma'$ *and* $\Gamma' \vdash e' : T$.

Proof. The proof is done by induction on the reduction rules for the pure expressions, given in Fig. 1.4.

- Case (RedVar): By assumption $\Gamma \vdash \sigma$ and $\Gamma \vdash x : T$ and $\sigma \vdash x \rightsquigarrow \sigma \vdash \sigma(x)$. Since σ is well-typed $\Gamma \vdash \sigma(x) : \Gamma(x)$, so, $\Gamma \vdash \sigma(x) : T$.

- Case (RedCons): By induction hypothesis $\Gamma \vdash e_i : T_i$ and since $\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \leq i \leq n$, the $\Gamma' \vdash e_i' : T_i$ and $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \sigma'$. By assumption $\Gamma \vdash \mathsf{Co}(e_1 \ldots e_i \ldots e_n) : T$. Since $\Gamma \subseteq \Gamma'$, the $\Gamma' \vdash \mathsf{Co}(e_1 \ldots e_i' \ldots e_n) : T$.

- Case (RedFunExp): This case follows exactly the same line as (RedCons). By induction hypothesis $\Gamma \vdash e_i : T_i$ and since $\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \leq i \leq n$, the $\Gamma' \vdash e_i' : T_i$ and $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \sigma'$. By assumption $\Gamma \vdash \mathtt{fun}(e_1 \ldots e_i \ldots e_n) : T$. Since $\Gamma \subseteq \Gamma'$, the $\Gamma' \vdash \mathtt{fun}(e_1 \ldots e_i' \ldots e_n) : T$.

- Case (RedFunGround): By assumption $\Gamma \vdash \sigma$ and $\Gamma \vdash \mathtt{fun}(\overline{v}) : T$ and by (T-FunExp) we have $\Gamma \vdash \overline{v} : \overline{T}$ and $\Gamma(\mathtt{fun}) = \overline{T'} \to T'$, and there is a type substitution ρ such that $\overline{T} = \rho(\overline{T'})$ and $T = \rho(T')$. Obviously, $\Gamma, \overline{x}_{\mathtt{fun}} : \rho(\overline{T'}) \vdash \overline{x}_{\mathtt{fun}} : \overline{T'}$. By rule (T-FunDecl) we have $\Gamma, \overline{x}_{\mathtt{fun}} : \overline{T'} \vdash e_{\mathtt{fun}} : T'$. Since, typing is preserved by substitution, this means $\Gamma, \overline{x}_{\mathtt{fun}} : \rho(\overline{T'}) \vdash e_{\mathtt{fun}} : \rho(T')$. This is the same as $\Gamma, \overline{x}_{\mathtt{fun}} : \overline{T} \vdash e_{\mathtt{fun}} : T$. Let $\Gamma' = \Gamma, \overline{y} : \overline{T}$ where a renaming of variables has occurred. Then, $\Gamma' \vdash e_{\mathtt{fun}}[\overline{x}_{\mathtt{fun}} \mapsto \overline{y}] : T$. Since $\mathrm{fresh}(\{y_1 \ldots y_n\})$, then $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \sigma$, so $\Gamma' \vdash \sigma'$.

- Case (RedCase1): By assumption $\Gamma \vdash \sigma$ and $\Gamma \vdash \mathbf{case}\ e\ \{\overline{p \Rightarrow e_p}\} : T'$. By induction hypothesis $\Gamma \vdash e : T$, $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \sigma'$ and $\Gamma' \vdash e' : T$. Then, since $\Gamma \subseteq \Gamma'$ we have $\Gamma' \vdash \mathbf{case}\ e'\ \{\overline{p \Rightarrow e_p}\} : T'$.

- Case (RedCase2): By assumption $\Gamma \vdash \mathbf{case}\ v\ \{p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}}\} : T$, then also $\mathbf{case}\ v\ \{\overline{p' \Rightarrow e'_{p'}}\} : T$.

- Case (RedCase3): By assumption we have that $\Gamma \vdash \sigma$ and $\Gamma \vdash \mathbf{case}\ v\ \{p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}}\} : T'$ and $\mathrm{match}(\sigma(p), v) = \sigma''$ which implies that $\mathrm{vars}(\sigma(p)) \cap \mathsf{dom}(\sigma) = \emptyset$. By rule (T-Case) we have that $\Gamma \vdash v : T$ and $\Gamma \vdash p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}} : T \to T'$ for some type T. By rule (T-Branch) we have that $\Gamma'' = \Gamma \circ \mathrm{pmatch}(\sigma(p), T)$ and $\Gamma'' \vdash \sigma(p) : T$, $\Gamma'' \vdash e_p : T'$, and let $\rho = \mathrm{pmatch}(\sigma(p), T)$. Since, $\mathsf{dom}(\rho) \cap \mathsf{dom}(\sigma) = \emptyset$, then $\Gamma \circ \rho \vdash \sigma \circ \sigma''$. By renaming the variable in $\sigma(p)$ we let $\Gamma' = \Gamma, \overline{y} : \Gamma''(\overline{x})$ and $\Gamma \subseteq \Gamma'$. Then we get $\Gamma' \vdash \sigma'$ and $\Gamma' \vdash e_p[\overline{x} \mapsto \overline{y}] : T'$.

□

Theorem 3.2.3 (Subject Reduction)**.** *If $\Delta, \mathcal{G} \vdash_R N$ and $N \to N'$ then $\exists\, \Delta', \mathcal{G}'$ such that $\Delta \subseteq \Delta'$, $\mathcal{G} \supseteq \mathcal{G}'$ and $\Delta', \mathcal{G}' \vdash_R N'$.*

Proof. The proof is done by induction over the operational semantics rules. We assume that class definitions are well-typed and for simplicity we omit them from the runtime syntax.

- Case (Skip): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \mathbf{skip}; s\}, Q)$; but then also $\Delta, \mathcal{G} \vdash ob(\mathsf{o}, \sigma, \{\sigma' | s\}, Q)$.

- Case *Assignment*: By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | x = e; s\}, Q)$, and $x \in \mathsf{dom}(\sigma')$ and $v = [\![e]\!]_{(\sigma \circ \sigma')}$, and let $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma' = \overline{T'\ x'\ v}; \theta'$. Let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. Then, by rules (T-Object) and (T-Process) and Lemma 4.5.3, we have $\Delta', \mathcal{G}_1 \vdash_R x = v; s$, such that $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$ and $\Delta', \mathcal{G}_2 \vdash_R Q$. The derivation $\Delta', \mathcal{G}_1 \vdash_R x = v; s$ implies that $\Delta', \emptyset \vdash_R v : \Delta'(x)$, by rule (T-Assign) being v a value, and $\Delta', \mathcal{G}_1 \vdash_R s$. By rule (Assign-Local) we have $ob(\mathsf{o}, \sigma, \{\sigma' | x = v; s\}, Q) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma'[x \mapsto v] | s\}, Q)$. By applying rule (T-Object) we obtain $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma'[x \mapsto v] | s\}, Q)$.
  Case (Assign-Field) follows the same line as case (Assign-Local). Since $ob(\mathsf{o}, \sigma, \{\sigma' | x = v; s\}, Q) \rightarrow ob(\mathsf{o}, \sigma[x \mapsto v], \{\sigma' | s\}, Q)$, then we derive $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma[x \mapsto v], \{\sigma' | s\}, Q)$.

- Case *Conditionals*: By assumption we have that $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s\}, Q)$ and $[\![e]\!]_{(\sigma \circ \sigma')} = \texttt{true}$. There exists some Δ' which extends Δ with typing assumptions present in σ and σ'; namely $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma' = \overline{T'\ x'\ v}; \theta'$, and $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. By assumption $\Delta', \emptyset \vdash_R x : \texttt{Bool}$, $\Delta', \mathcal{G}_1 \vdash_R s_1$, $\Delta', \mathcal{G}_1 \vdash_R s_2$, $\Delta', \mathcal{G}_2 \vdash_R s$, and $\Delta', \mathcal{G}_3 \vdash_R Q$ where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2 \uplus \mathcal{G}_3$. Then, by rule (T-Comp) we have that $\Delta', \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash_R s_1; s$. By rule (Cond-True) we obtain $ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s\}, Q) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma' | s_1; s\}, Q)$ and by (T-Object) we conclude $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | s_1; s\}, Q)$. The case (Cond-False) follows the same line as case (Cond-True), where $[\![e]\!]_{(\sigma \circ \sigma')} = \texttt{false}$ and hence $ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s\}, Q) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma' | s_2; s\}, Q)$, then, we derive $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | s_2; s\}, Q)$.

- Case *Loops*: By assumption we have that $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{ while } e \ \{ \ s \ \}; s'\}, Q)$, and $[\![e]\!]_{(\sigma \circ \sigma')} = \texttt{true}$. There exists some Δ' which extends Δ with typing assumptions present in σ and σ'; namely $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma' = \overline{T'\ x'\ v}; \theta'$, and $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. By assumption $\Delta', \mathcal{G}_1 \vdash_R \textbf{ while } e \ \{ \ s \ \}; s'$, and $\Delta', \mathcal{G}_2 \vdash_R Q$, where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By (T-Comp) we have $\Delta', \mathcal{G}'_1 \vdash_R \textbf{ while } e \ \{ \ s \ \}$, which by (T-While) means $\Delta', \emptyset \vdash_R e : \texttt{Bool}$ and $\Delta', \mathcal{G}'_1 \vdash_R s$, and $\Delta', \mathcal{G}''_1 \vdash_R s'$ where $\mathcal{G}_1 = \mathcal{G}'_1 \uplus \mathcal{G}''_1$. By rule (While-True) we have $ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{ while } e \ \{ \ s \ \}; s'\}, Q) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma' | \ s; ; \textbf{while } e \ \{ \ s \ \}; s'\}, Q)$. Since we have $\Delta', \mathcal{G}'_1 \vdash_R s$ and $\Delta', \mathcal{G}'_1 \vdash_R \textbf{ while } e \ \{ \ s \ \}$ then by applying (T-OtherComp) we obtain $\Delta', \mathcal{G}'_1 \vdash_R s \ ; ; \textbf{while } e \ \{ \ s \ \}$. By (T-Comp) we have $\Delta', \mathcal{G}_1 \vdash_R s \ ; ; \textbf{while } e \ \{ \ s \ \}; s'$. We conclude by (T-Object). Now let $[\![e]\!]_{(\sigma \circ \sigma')} = \texttt{false}$. By rule (While-False) we have

$ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{while}\ e\ \{\ s\ \};\ s'\}, Q) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{skip};\ s'\}, Q)$. By (T-Skip) and Lemma 3.2.1 we have $\Delta', \emptyset \vdash_R \textbf{skip}$. By rules (T-Comp) and (T-Object) we have $\Delta', \mathcal{G}_1'' \uplus \mathcal{G}_2 \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{skip};\ s'\}, Q)$ and $\mathcal{G}_1'' \uplus \mathcal{G}_2 \subseteq \mathcal{G}$.

- Case (Suspend): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{suspend};\ s\}, Q)$ and by the transition rule $ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{suspend};\ s\}, Q) \rightarrow ob(\mathsf{o}, \sigma, \textbf{idle}, Q \cup \{\sigma' | s\})$. Let $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma' = \overline{T'\ x'\ v}; \theta'$, then $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. By (T-Object) we have $\Delta', \mathcal{G}_1 \vdash_R \textbf{suspend}; s$ and $\Delta', \mathcal{G}_2 \vdash_R Q$ and $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By (T-Process) and (T-Suspend) we have $\Delta', \emptyset \vdash_R \textbf{suspend}$ and $\Delta', \mathcal{G}_1 \vdash_R s$. By (T-Process), (T-Idle) and (T-Object) we conclude that $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \textbf{idle}, Q \cup \{\sigma' | s\})$.

- Case (Release-Cog): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \textbf{idle}, Q)\ cog(\mathsf{c}, \mathsf{o})$ and by the transition rule $ob(\mathsf{o}, \sigma, \textbf{idle}, Q)\ cog(\mathsf{c}, \mathsf{o}) \rightarrow ob(\mathsf{o}, \sigma, \textbf{idle}, Q)\ cog(\mathsf{c}, \epsilon)$. By (T-Config) we have $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \textbf{idle}, Q)$ and $\Delta, \emptyset \vdash_R cog(\mathsf{c}, \mathsf{o})$. By assumption $\mathsf{c} = \sigma(cog)$, and σ is well-typed in Δ, then $\Delta(\mathsf{c}) = \textbf{cog}$. By applying (T-Cog) we have $\Delta, \emptyset \vdash_R cog(\mathsf{c}, \epsilon)$.

- Case (Activate): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \textbf{idle}, Q)\ cog(\mathsf{c}, \epsilon)\ N$, which by (T-Config) means $\Delta, \mathcal{G}_1 \vdash_R ob(\mathsf{o}, \sigma, \textbf{idle}, Q)$ and $\Delta, \emptyset \vdash_R cog(\mathsf{c}, \epsilon)$ and $\Delta, \mathcal{G}_2 \vdash_R N$, where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By the transition rule $ob(\mathsf{o}, \sigma, \textbf{idle}, Q)\ cog(\mathsf{c}, \epsilon)\ N \rightarrow ob(\mathsf{o}, \sigma, \{p\}, Q \setminus \{p\})\ cog(\mathsf{c}, \mathsf{o})\ N$. Suppose $\sigma = \overline{T\ x\ v}; \theta$ and let $\Delta' = \Delta, \overline{x} : \overline{T}$. Then, by (T-Object) Q is well-typed in Δ'. By assumption $p = \text{select}(Q, \sigma, N)$, hence $p \in Q$, namely p is well-typed in Δ'. Then, $\Delta, \mathcal{G}_1 \vdash_R ob(\mathsf{o}, \sigma, \{p\}, Q \setminus \{p\})$. By assumption $\mathsf{c} = \sigma(\textbf{cog})$ and since σ and o are well-typed in Δ, by (T-Cog) we have $\Delta, \emptyset \vdash_R cog(\mathsf{c}, \mathsf{o})$.

- Case *Awaits*: By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{await}\ g; s\}, Q)\ N$. By (Await-True), since $[\![g]\!]^N_{(\sigma \circ \sigma')}$, then $ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{await}\ g; s\}, Q)\ N \rightarrow ob(\mathsf{o}, \sigma, \{\sigma' | s\}, Q)\ N$. Trivially, $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | s\}, Q)\ N$. By (Await-False), since $\neg[\![g]\!]^N_{(\sigma \circ \sigma')}$, then $ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{await}\ g; s\}, Q)\ N \rightarrow ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{suspend}; \textbf{await}\ g; s, Q)\ N$. By (T-Suspend), $\Delta, \emptyset \vdash_R \textbf{suspend}$. Then, by (T-Comp), $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{suspend}; \textbf{await}\ g; s, Q)\ N$.

- Case (Return): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{return}\ e; s\}, Q)$ and $\Delta, \emptyset \vdash_R fut(\mathsf{f}, \bot)$ and by the transition rule $\sigma'(\textbf{destiny}) = \mathsf{f}$ and $v = [\![e]\!]_{(\sigma \circ \sigma')}$ and $ob(\mathsf{o}, \sigma, \{\sigma' |\ \textbf{return}\ e; s\}, Q)\ fut(\mathsf{f}, \bot) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma' | s\}, Q)\ fut(\mathsf{f}, v)$. Trivially, $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | s\}, Q)$. By the premises of (T-Return) we have $\Delta \vdash_R e : T$ and $\Delta(\textbf{destiny}) = \mathtt{Fut}\langle T \rangle$. By assumption $\sigma'(\textbf{destiny}) = \mathsf{f}$,

hence $\Delta(\mathtt{f}) = \mathtt{Fut}\langle T\rangle$. By assumption $v = [\![e]\!]_{(\sigma\circ\sigma')}$, then by applying Lemma 4.5.3 we have $\Delta \vdash_R v : T$. By applying (T-Future1) we have $\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, v)$. We conclude by applying (T-Config).

- Case (Read-Fut): By assumption $\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' \mid x = \mathbf{get}(e); s\}, Q)$ and $\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, v)$, where $v \neq \bot$ and $\mathtt{f} = [\![e]\!]_{(\sigma\circ\sigma')}$. By the transition rule $\mathit{ob}(\mathtt{o}, \sigma, \{\sigma' \mid x = \mathbf{get}(e); s\}, Q)\ \mathit{fut}(\mathtt{f}, v) \rightarrow \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' | x = v; s\}, Q)\ \mathit{fut}(\mathtt{f}, v)$. By (T-Future1) $\Delta(\mathtt{f}) = \mathtt{Fut}\langle T\rangle$ and $\Delta \vdash_R v : T$. Since by assumption $\mathtt{f} = [\![e]\!]_{(\sigma\circ\sigma')}$, consequently $\Delta \vdash_R \mathbf{get}(e) : T$. Then, $\Delta, \emptyset \vdash_R x = v$ and hence $\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' | x = v; s\}, Q)$.

- Case (Bind-Mtd): By assumption $\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, p, Q)$ and $\Delta, \emptyset \vdash_R \mathit{invoc}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})$ and by the transition rule $\mathit{ob}(\mathtt{o}, \sigma, p, Q)\ \mathit{invoc}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}) \rightarrow \mathit{ob}(\mathtt{o}, \sigma, p, Q \cup p')$. By assumption $p' = \mathrm{bind}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}, \mathit{class}(\mathtt{o}))$ and let $\mathit{class}(\mathtt{o}) = \mathtt{C}$. The bind function returns a process $p' = \{\overline{T\ x = v}; \overline{T'\ x' = \mathbf{null}}; \mathbf{this} = \mathtt{o} \mid s\}$ where either (NM-Bind) or (CM-Bind) is applied, depending on whether the method $\mathtt{m}$ is **critical** or not. Let $\sigma = \overline{T\ x\ w}; \theta$ and let $\Delta' = \Delta, \overline{x} : \overline{T}$. Then, process p' is well-typed in Δ augmented with $\mathit{fields}(\mathtt{C})$, namely $\Delta', \emptyset \vdash_R p'$. Then, by (T-Object) and (T-Process-Queue) $\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, p, Q \cup p')$.

- Case (New-Object): By assumption $\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, \{\ \sigma'' \mid x = \mathbf{new}\ \mathtt{C}\ (\overline{e})\ \}; s, Q)$ and $\Delta, \emptyset \vdash_R \mathit{cog}(\mathtt{c}, \mathtt{o})$ and by transition rule $\mathit{ob}(\mathtt{o}, \sigma, \{\ \sigma'' \mid x = \mathbf{new}\ \mathtt{C}\ (\overline{e})\ \}; s, Q)\ \mathit{cog}(\mathtt{c}, \mathtt{o}) \rightarrow \mathit{ob}(\mathtt{o}, \sigma, \{\sigma'' | x = \mathtt{o}'; s\}, Q)\ \mathit{cog}(\mathtt{c}, \mathtt{o})\ \mathit{ob}(\mathtt{o}', \sigma', \mathbf{idle}, \varepsilon)$. By assumption $\overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma'')}$ $\mathrm{fresh}(\mathtt{o}')$ $\sigma' = \mathrm{atts}(\mathtt{C}, \overline{v}, \mathtt{o}', \mathtt{c})$. Suppose $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma'' = \overline{T'\ x'\ v}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. By (T-Object) and (T-Process) we have that $\Delta', \mathcal{G}_1 \vdash_R x = \mathbf{new}\ \mathtt{C}\ (\overline{e}); s$ where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$ and $\mathcal{G}_2$ is the set of cogs in Q. By (T-Comp) $\Delta', \emptyset \vdash_R x = \mathbf{new}\ \mathtt{C}\ (\overline{e})$ and $\Delta', \mathcal{G}_1 \vdash_R s$. By rule (T-Assign) we have that $\Delta'(x) = T$ and $\Delta', \emptyset \vdash_R \mathbf{new}\ \mathtt{C}\ (\overline{e}) : T$. By the premises of the typing rule we have that $\mathit{fields}(\mathtt{C}) = \overline{T\ f}$, $\Delta' \vdash \overline{x : T'}$, $\mathrm{tmatch}(\overline{T}, \overline{T'}) = \pi$ and $T \in \mathrm{crec}(\mathtt{G}, \mathtt{C}, \pi)$, and let $\pi = \sigma \circ \sigma''$. Then, by the definition of the auxiliary function crec, it means that $T = (\mathtt{I}, \mathtt{G}[\pi \uplus \rho(\overline{f : (\mathtt{I}, \mathbb{r})})])$ and $\mathrm{implements}(\mathtt{C}, \mathtt{I})$. We notate $\mathbb{r} = \mathtt{G}[\pi \uplus \rho(\overline{f : (\mathtt{I}, \mathbb{r})})]$. Since $\mathit{fresh}(\mathtt{o}')$, then let $\Delta'' = \Delta, \mathtt{o}' : (\mathtt{C}, \mathbb{r})$. Then, $\Delta'', \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash_R \mathit{ob}(\mathtt{o}, \sigma, \{\sigma'' | x = \mathtt{o}'; s\}, Q)$, by applying (T-Process), (T-Comp), and (T-Object). Trivially, $\Delta'', \emptyset \vdash_R \mathit{cog}(\mathtt{c}, \mathtt{o})$. By assumption, function $\mathrm{atts}(\mathtt{C}, \overline{v}, \mathtt{o}', \mathtt{c})$ returns a substitution σ' that is well-typed

in Δ''. So, $\Delta'', \emptyset \vdash_R ob(\mathsf{o}', \sigma', \mathbf{idle}, \varepsilon)$. Then, by (T-Config) we have $\Delta'', \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma''|x = \mathsf{o}'; s\}, Q)\ cog(\mathsf{c}, \mathsf{o})\ ob(\mathsf{o}', \sigma', \mathbf{idle}, \varepsilon)$.

- Case (New-Cog-Object): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma''|x = \mathbf{new\ cog}\ \mathsf{C}\ (\overline{e}); s\}, Q)$ and $\Delta, \emptyset \vdash_R cog(\mathsf{c}, \mathsf{o})$, and by the transition rule we have $ob(\mathsf{o}, \sigma, \{\sigma''|x = \mathbf{new\ cog}\ \mathsf{C}\ (\overline{e}); s\}, Q)\ cog(\mathsf{c}, \mathsf{o}) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma''|x = \mathsf{o}'; s\}, Q)\ ob(\mathsf{o}', \sigma', \mathbf{idle}, \varepsilon)\ cog(\mathsf{c}', \mathsf{o}')$. By assumption $\overline{v} = \overline{[\![e]\!]_{(\sigma \circ \sigma'')}}$, fresh($\mathsf{o}'$), $\sigma' = \text{atts}(\mathsf{C}, \overline{v}, \mathsf{o}', \mathsf{c})$. Suppose $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma'' = \overline{T'\ x'\ v}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. By rules (T-Object) and (T-Process) we have that $\Delta', \mathcal{G}_1 \vdash_R x = \mathbf{new\ cog}\ \mathsf{C}\ (\overline{e}); s$ where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$ and $\mathcal{G}_2$ is the set of cogs in Q. By (T-Comp), (T-Cog) and (T-Assign) $\Delta', \{\mathsf{G}\} \vdash_R x = \mathbf{new\ cog}\ \mathsf{C}\ (\overline{e})$ and $\Delta', \mathcal{G}_1 \setminus \{\mathsf{G}\} \vdash_R s$. By (T-Assign), $\Delta'(x) = T$ and $\Delta', \{\mathsf{G}\} \vdash_R \mathbf{new\ cog}\ \mathsf{C}\ (\overline{e}) : T$. By the premises of (T-Cog), $fields(\mathsf{C}) = \overline{T\ f}$, $\Delta' \vdash \overline{x : T'}$, $\text{tmatch}(\overline{T}, \overline{T'}) = \pi$ and $T \in \text{crec}(\mathsf{G}, \mathsf{C}, \pi)$, and let $\pi = \sigma \circ \sigma''$. Then, by definition of the auxiliary function crec, it means that $T = (\mathtt{I}, \mathsf{G}[\pi \uplus \rho(\overline{f : (\mathtt{I}, \mathbb{r})})])$ and implements(C, $\mathtt{I}$). We notate $\mathbb{r} = \mathsf{G}[\pi \circ \rho(\overline{f : (\mathtt{I}, \mathbb{r})})]$. Since $fresh(\mathsf{o}')$, and $fresh(\mathsf{c}')$, let $\Delta'' = \Delta, \mathsf{o}' : (\mathsf{C}, \mathbb{r}), \mathsf{c}' : \mathbf{cog}$. Then, $\Delta'', \mathcal{G}_1 \setminus \{\mathsf{G}\} \uplus \mathcal{G}_2 \vdash_R ob(\mathsf{o}, \sigma, \{\sigma''|x = \mathsf{o}'; s\}, Q)$, by applying (T-Process), (T-Comp), and (T-Object). Trivially, by Lemma 3.2.1 $\Delta'', \emptyset \vdash_R cog(\mathsf{c}', \mathsf{o}')$. By assumption, function atts($\mathsf{C}, \overline{v}, \mathsf{o}', \mathsf{c}'$) returns a substitution σ' that is well-typed in Δ''. So, $\Delta'', \emptyset \vdash_R ob(\mathsf{o}', \sigma', \mathbf{idle}, \varepsilon)$. Then, by (T-Config) we have $\Delta'', \mathcal{G} \setminus \{\mathsf{G}\} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma''|x = \mathsf{o}'; s\}, Q)\ ob(\mathsf{o}', \sigma', \mathbf{idle}, \varepsilon)\ cog(\mathsf{c}', \mathsf{o}')$.

- Case (Self-Sync-Call): By assumption let $\Delta, \mathcal{G}' \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' \mid x = e.\mathtt{m}(\overline{e}); s\}, Q)$ and $\mathsf{o} = [\![e]\!]_{(\sigma \circ \sigma')}$, $\overline{v} = \overline{[\![e]\!]_{(\sigma \circ \sigma')}}$, $\sigma'(\mathbf{destiny}) = \mathtt{f}'$, fresh($\mathtt{f}$), and $\{\sigma''|s'\} = \text{bind}(\mathsf{o}, \mathtt{f}, \mathtt{m}, \overline{v}, class(\mathsf{o}))$ and let $class(\mathsf{o}) = \mathsf{C}$. Since, by assumption class C is well-typed in Δ, by (T-Class) this means that all methods in C are well-typed, in particular method $\mathtt{m}$ is well-typed in C. The auxiliary function bind returns a process $\{\sigma''|s'\}$, which contains the body s' of the method $\mathtt{m}$, which in turn by (T-Method) is well-typed. By the transition rule $ob(\mathsf{o}, \sigma, \{\sigma' \mid x = e.\mathtt{m}(\overline{e}); s\}, Q) \rightarrow ob(\mathsf{o}, \sigma, \{\sigma''|s'; \mathbf{cont}(\mathtt{f}')\}, Q \cup \{\sigma' \mid x = \mathbf{get}(\mathtt{f}); s\})\ fut(\mathtt{f}, \bot)$. Suppose $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma' = \overline{T'\ x'\ w'}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. By (T-Object) and (T-Process) we have that $\Delta', \mathcal{G}_1 \vdash_R x = e.\mathtt{m}(\overline{e}); s$ and $\Delta', \mathcal{G}_2 \vdash_R Q$, where $\mathcal{G}' = \mathcal{G}_1 \uplus \mathcal{G}_2$. From the first judgement, by (T-Comp), we have that $\Delta', \emptyset \vdash_R x = e.\mathtt{m}(\overline{e})$ and $\Delta', \mathcal{G}_1 \vdash_R s$. By the premises of (T-SCall) we have that $mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}'', \mathbb{r})(\overline{T\ x}) \rightarrow T$, $\Delta' \vdash e : (\mathtt{I}, \rho(\mathbb{r}))$, $\Delta' \vdash \overline{e} : \overline{\rho(T)}$, coloc($\rho(\mathbb{r}), \Delta'(\mathbf{this})$). By assumption we have that $\mathsf{o} = [\![e]\!]_{(\sigma \circ \sigma')}$ then $\Delta'(\mathsf{o}') : (\mathsf{C}, \rho(\mathbb{r}))$, such that implements($\mathsf{C}$, $\mathtt{I}$),

and $mtype(\mathtt{m}, \mathtt{C}) = mtype(\mathtt{m}, \mathtt{I})$. Let $\sigma'' = \overline{T''\ x''\ w''}; \theta''$, then by assumption and by (T-Method) we have $\Delta', \overline{x'' : T''}, \mathcal{G}'' \vdash s'$, hence $\Delta', \mathcal{G}'' \vdash_R \{\sigma''|s'\}$ (note that **destiny** : $\mathtt{Fut}\langle T\rangle$, **this** : $(\mathtt{C}, \mathbb{r})$ are part of $\sigma \circ \sigma'$) and by (T-Config) $\mathcal{G} = \mathcal{G}' \uplus \mathcal{G}''$. Since $\sigma'(\mathbf{destiny}) = \mathtt{f}'$, then $\Delta', \mathcal{G}'' \vdash_R \{\sigma''|s'; \mathbf{cont}(\mathtt{f}')\}$. Since $fresh(\mathtt{f})$, let $\Delta'' = \Delta, \mathtt{f} : \rho(T)$, then $\Delta'', \emptyset \vdash_R x = \mathbf{get}(\mathtt{f})$. Then, $\Delta'', \mathcal{G}' \uplus \mathcal{G}'' \vdash_R ob(\mathtt{o}, \sigma, \{\sigma''|s'; \mathbf{cont}(\mathtt{f}')\}, Q \cup \{\sigma'|\ x = \mathbf{get}(\mathtt{f}); s\})$. By (T-Future2) we have $\Delta'', \emptyset \vdash_R fut(\mathtt{f}, \bot)$. We conclude by (T-Config).

- Case (Self-Sync-Return-Sched): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma''|\mathbf{cont}(\mathtt{f})\}, Q \cup \{\sigma'|s\})$ and by the transition rule $ob(\mathtt{o}, \sigma, \{\sigma''|\mathbf{cont}(\mathtt{f})\}, Q \cup \{\sigma'|s\}) \rightarrow ob(\mathtt{o}, \sigma, \{\sigma'|s\}, Q)$, since $\sigma'(\mathbf{destiny}) = \mathtt{f}$. Suppose $\sigma = \overline{T\ x\ v}; \theta$, and let $\Delta' = \Delta, \overline{x} : \overline{T}$. By (T-Object) we have that $\Delta', \mathcal{G} \vdash_R Q \cup \{\sigma'|s\}$, by (T-Process-Queue) $\Delta', \mathcal{G}_1 \vdash_R Q$ and $\Delta', \mathcal{G}_2 \vdash_R \{\sigma'|s\}$, where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By (T-Object) we have $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'|s\}, Q)$.

- Case (Async-Call): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma' \mid x = e!\mathtt{m}(\overline{e}); s\}, Q)$ and by the transition rule $ob(\mathtt{o}, \sigma, \{\sigma' \mid x = e!\mathtt{m}(\overline{e}); s\}, Q) \rightarrow ob(\mathtt{o}, \sigma, \{\sigma'|x = \mathtt{f}; s\}, Q)$ $invoc(\mathtt{o}', \mathtt{f}, \mathtt{m}, \overline{v})$ $fut(\mathtt{f}, \bot)$. Suppose $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma' = \overline{T'\ x'\ v}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. By (T-Object) and (T-Process) we have that $\Delta', \mathcal{G}_1 \vdash_R x = e!\mathtt{m}(\overline{e}); s$ where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$ and $\mathcal{G}_2$ is the set of cogs in Q. By (T-Comp) we have $\Delta', \emptyset \vdash_R x = e!\mathtt{m}(\overline{e})$ and $\Delta', \mathcal{G}_1 \vdash_R s$. For the first judgement, by (T-Assign) and (T-ACall), we have that $\Delta'(x) = \mathtt{Fut}\langle\rho(T)\rangle$ and $\Delta' \vdash_R e!\mathtt{m}(\overline{e}) : \mathtt{Fut}\langle\rho(T)\rangle$. By the premises of (T-ACall), we have that $mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}'', \mathbb{r})(\overline{T\ x}) \rightarrow T$, $\Delta' \vdash_R e : (\mathtt{I}, \rho(\mathbb{r}))$ and $\Delta' \vdash_R \overline{e} : \overline{\rho(T)}$. By the premises of (Async-Call) we have $\mathtt{o}' = [\![e]\!]_{(\sigma\circ\sigma')}$, $\overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma')}$, and since substitutions are well-typed in Δ' and by Lemma 4.5.3 it means $\Delta' \vdash_R \mathtt{o}' : (\mathtt{C}, \rho(\mathbb{r}))$ for a class $\mathtt{C}$ such that implements($\mathtt{C}$, $\mathtt{I}$), since by assumption class definitions are well-typed; and $mtype(\mathtt{m}, \mathtt{C}) = mtype(\mathtt{m}, \mathtt{I})$. Also, by Lemma 4.5.3 $\Delta' \vdash_R \overline{v} : \overline{\rho(T)}$. Since, by assumption $fresh(\mathtt{f})$, let $\Delta'' = \Delta', \mathtt{f} : \mathtt{Fut}\langle\rho(T)\rangle$, hence $\mathtt{f}$ can be safely added. By applying (T-Assign) we have $\Delta'' \vdash_R x = \mathtt{f}$, and by (T-Object) we have $\Delta'', \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'|x = \mathtt{f}; s\}, Q)$. By applying (T-Invoc) we have $\Delta'', \emptyset \vdash_R invoc(\mathtt{o}', \mathtt{f}, \mathtt{m}, \overline{v})$. By applying (T-Future2) we have $\Delta'', \emptyset \vdash_R fut(\mathtt{f}, \bot)$. Then, we conclude by applying (T-Config).

- Case (Cog-Sync-Call): By assumption $\mathtt{o}' = [\![e]\!]_{(\sigma\circ\sigma'')}$, $\overline{v} = [\![\overline{e}]\!]_{(\sigma\circ\sigma'')}$, fresh($\mathtt{f}$), $\sigma'(\mathbf{cog}) = \mathtt{c}$, $\mathtt{f}' = \sigma''(\mathbf{destiny})$ and $\{\sigma'''|s'\} =$

bind($\mathtt{o}', \mathtt{f}, \mathtt{m}, \overline{v}, class(\mathtt{o}')$) and let $class(\mathtt{o}') = \mathtt{C}$. Also, $\Delta, \mathcal{G}' \vdash_R ob(\mathtt{o}, \sigma, \{\sigma''|x = e.\mathtt{m}(\overline{e}); s\}, Q)$ $ob(\mathtt{o}', \sigma', \mathbf{idle}, Q')$ $cog(\mathtt{c}, \mathtt{o})$. Since, by assumption class C is well-typed in Δ, by (T-Class) this means that all methods in C are well-typed, in particular method m is well-typed in C. The auxiliary function bind returns a process $\{\sigma'''|s'\}$, which contains the body s' of the method m, which in turn by (T-Method) is well-typed. By the transition rule $ob(\mathtt{o}, \sigma, \{\sigma''|x = e.\mathtt{m}(\overline{e}); s\}, Q)$ $ob(\mathtt{o}', \sigma', \mathbf{idle}, Q')$ $cog(\mathtt{c}, \mathtt{o}) \rightarrow ob(\mathtt{o}, \sigma, \mathbf{idle}, Q \cup \{\sigma''|x = \mathbf{get}(\mathtt{f}); s\})$ $fut(\mathtt{f}, \bot)$ $ob(\mathtt{o}', \sigma', \{\sigma'''|s'; \mathbf{cont}(\mathtt{f}')\}, Q')$ $cog(\mathtt{c}, \mathtt{o}')$. By (T-Config) it means that $\Delta, \mathcal{G}'_1 \vdash_R ob(\mathtt{o}, \sigma, \{\sigma''|x = e.\mathtt{m}(\overline{e}); s\}, Q)$ and $\Delta, \mathcal{G}'_2 \vdash_R ob(\mathtt{o}', \sigma', \mathbf{idle}, Q')$ and $\Delta \vdash_R cog(\mathtt{c}, \mathtt{o})$, where $\mathcal{G}' = \mathcal{G}'_1 \uplus \mathcal{G}'_2$. Suppose $\sigma = \overline{T\ x\ w}; \theta$, $\sigma' = \overline{T'\ x'\ w'}; \theta'$, and $\sigma'' = \overline{T''\ x''\ w''}; \theta$ and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}, \overline{x'} : \overline{T''}$. From the first judgement, by (T-Assign) and (T-Process) we have $\Delta'(x) = \rho(T)$, $\Delta' \vdash_R e.\mathtt{m}(\overline{e}) : \rho(T)$, $\Delta', \mathcal{G}''_1 \vdash_R s$, $\Delta', \mathcal{G}'''_1 \vdash_R Q$ where $\mathcal{G}'_1 = \mathcal{G}''_1 \uplus \mathcal{G}'''_1$. By the premises of (T-SCall) we know that $mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}'', \mathbb{r})(\overline{T\ x}) \rightarrow T$, $\Delta' \vdash_R e : (\mathtt{I}, \rho(\mathbb{r}))$, $\Delta' \vdash_R \overline{e} : \overline{\rho(T)}$. By assumption, we have $\mathtt{o}' = [\![e]\!]_{(\sigma \circ \sigma'')}$ and $\overline{v} = [\![\overline{e}]\!]_{(\sigma \circ \sigma'')}$ then trivially, $\Delta'(\mathtt{o}') : (\mathtt{C}, \rho(\mathbb{r}))$ implements(C, I). Then $mtype(\mathtt{m}, \mathtt{C}) = mtype(\mathtt{m}, \mathtt{I})$, and $\Delta' \vdash_R \overline{v} : \overline{\rho(T)}$. Since $fresh(\mathtt{f})$, let $\Delta'' = \Delta, \mathtt{f} : \mathtt{Fut}\langle\rho(T)\rangle$. By applying (T-Get) we obtain $\Delta'' \vdash_R \mathbf{get}(\mathtt{f}) : \rho(T)$. By (T-Object) and (T-Idle) and (T-Process-Queue) we have $\Delta'', \mathcal{G}'_1 \vdash_R ob(\mathtt{o}, \sigma, \mathbf{idle}, Q \cup \{\sigma''|x = \mathbf{get}(\mathtt{f}); s\})$. By applying rule (T-Future2) we have $\Delta'', \emptyset \vdash_R fut(\mathtt{f}, \bot)$. By assumption we have that $\{\sigma'''|s'\} = \text{bind}(\mathtt{o}', \mathtt{f}, \mathtt{m}, \overline{v}, class(\mathtt{o}'))$ where s' is the body of method m. Let $\sigma''' = \overline{T''\ x''\ w''}; \theta''$; by assumption and by (T-Method) we have $\Delta'', \overline{x'' : T''}, \mathcal{G}'' \vdash s'$, hence $\Delta'', \mathcal{G}'' \vdash_R \{\sigma''|s'\}$ (note that **destiny** : $\mathtt{Fut}\langle T\rangle$, **this** : $(\mathtt{C}, \mathbb{r})$ are part of $\sigma \circ \sigma''$), and by (T-Config) $\mathcal{G} = \mathcal{G}' \uplus \mathcal{G}''$. In addition, $\mathtt{f}' = \sigma''(\mathbf{destiny})$, meaning $\mathtt{f}' \in \mathsf{dom}(\Delta'')$ so, $\Delta'', \emptyset \vdash_R \mathbf{cont}(\mathtt{f}')$, by applying (T-Cont). Then, $\Delta'', \mathcal{G}'' \uplus \mathcal{G}'_2 \vdash_R ob(\mathtt{o}', \sigma', \{\sigma'''|s'; \mathbf{cont}(\mathtt{f}')\}, Q')$. By applying rule (T-Cog), $\Delta'', \emptyset \vdash_R cog(\mathtt{c}, \mathtt{o}')$. By (T-Config) $\Delta'', \mathcal{G}'_1 \uplus \mathcal{G}'_2 \uplus \mathcal{G}'' \vdash_R ob(\mathtt{o}, \sigma, \mathbf{idle}, Q \cup \{\sigma''|x = \mathbf{get}(\mathtt{f}); s\})$ $fut(\mathtt{f}, \bot)$ $ob(\mathtt{o}', \sigma', \{\sigma'''|s'; \mathbf{cont}(\mathtt{f}')\}, Q')$ $cog(\mathtt{c}, \mathtt{o}')$.

- Case (Cog-Sync-Return-Sched): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'|\mathbf{cont}(\mathtt{f})\}, Q)$ $cog(\mathtt{c}, \mathtt{o})$ $ob(\mathtt{o}', \sigma'', \mathbf{idle}, Q' \cup \{\sigma'''|s\})$ and by the transition rule $ob(\mathtt{o}, \sigma, \{\sigma'|\mathbf{cont}(\mathtt{f})\}, Q)$ $cog(\mathtt{c}, \mathtt{o})$ $ob(\mathtt{o}', \sigma'', \mathbf{idle}, Q' \cup \{\sigma'''|s\}) \rightarrow ob(\mathtt{o}, \sigma, \mathbf{idle}, Q)$ $cog(\mathtt{c}, \mathtt{o}')$ $ob(\mathtt{o}', \sigma'', \{\sigma'''|s\}, Q')$. By (T-Config) it means that $\Delta, \mathcal{G}_1 \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'|\mathbf{cont}(\mathtt{f})\}, Q)$ and

$\Delta, \emptyset \vdash_R cog(\mathtt{c}, \mathtt{o})$ and $\Delta, \mathcal{G}_2 \vdash_R ob(\mathtt{o}', \sigma'', \mathbf{idle}, Q' \cup \{\sigma'''|s\})$, where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. Trivially, $\Delta, \mathcal{G}_1 \vdash_R ob(\mathtt{o}, \sigma, \mathbf{idle}, Q)$, since by (T-Cont) **cont**(f) uses an empty set of cogs. By (T-Cog), since $\Delta(\mathtt{c}) = \mathbf{cog}$, then $\Delta, \emptyset \vdash_R cog(\mathtt{c}, \mathtt{o}')$. By the third typing judgement assumption, we have that $\Delta, \overline{x''} : \overline{T''}, \mathcal{G}_2 \vdash_R Q' \cup \{\sigma'''|s\}$, where we let $\sigma'' = \overline{T'' \; x'' \; w''}; \theta$, then trivially $\Delta, \mathcal{G}_2 \vdash_R ob(\mathtt{o}', \sigma'', \{\sigma'''|s\}, Q')$. We conclude by applying (T-Config).

- Case (Rebind-Local): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\ \sigma' \mid \mathbf{rebind}\ e.f = e'; s\ \}, Q)$ and by the transition rule $ob(\mathtt{o}, \sigma, \{\ \sigma' \mid \mathbf{rebind}\ e.f = e'; s\ \}, Q) \rightarrow ob(\mathtt{o}, \sigma[f \mapsto v], \{\ \sigma' \mid s\ \}, Q)$ and $\sigma(\mathtt{nb}_{cr}) = 0$, $\mathtt{o} = [\![e]\!]_{(\sigma \circ \sigma')}$, and $v = [\![e']\!]_{(\sigma \circ \sigma')}$. Suppose $\sigma = \overline{T\ x\ w}; \theta$ and $\sigma' = \overline{T'\ x'\ v}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. Then, $\Delta', \mathcal{G}_1 \vdash_R \mathbf{rebind}\ e.f = e'; s$ and where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$ and $\mathcal{G}_2$ is the set of cogs in Q. By (T-Rebind) $\Delta' \vdash_R e : (\mathtt{I}, \mathbb{r})$ and $\Delta', \mathcal{G}_1 \vdash_R e' : T$ and $T\ f \in ports(\mathtt{I})$ and coloc($\mathbb{r}, \Delta'(\mathbf{this})$) stating the belonging to the same cog. Since $v = [\![e']\!]_{(\sigma \circ \sigma')}$, then by Lemma 4.5.3 $\Delta' \vdash_R v : T$. Then, by (T-Object) $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma[f \mapsto v], \{\ \sigma' \mid s\ \}, Q)$.

- Case (Rebind-None): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\ \sigma' \mid \mathbf{rebind}\ e.f = e'; s\ \}, Q)$ and by the transition rule $ob(\mathtt{o}, \sigma, \{\ \sigma' \mid \mathbf{rebind}\ e.f = e'; s\ \}, Q) \rightarrow ob(\mathtt{o}, \sigma, \{\ \sigma' \mid s\ \}, Q)$. Then, trivially $\Delta, \mathcal{G}' \vdash_R ob(\mathtt{o}, \sigma, \{\ \sigma' \mid s\ \}, Q)$ and $\mathcal{G}' \subseteq \mathcal{G}$ where $\Delta, \mathcal{G} \setminus \mathcal{G}' \vdash_R \mathbf{rebind}\ e.f = e'$.

- Case (Rebind-Global): By assumption we have $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma_\mathtt{o}, K_{\mathbf{idle}}, Q)\ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid \mathbf{rebind}\ e.f = e'; s\ \}, Q')$. By the transition rule we have $ob(\mathtt{o}, \sigma_\mathtt{o}, K_{\mathbf{idle}}, Q)\ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid \mathbf{rebind}\ e.f = e'; s\ \}, Q') \rightarrow ob(\mathtt{o}, \sigma_\mathtt{o}[f \mapsto v], K_{\mathbf{idle}}, Q)\ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid s\ \}, Q')$. By (T-Config) it means that $\Delta, \mathcal{G}_1 \vdash_R ob(\mathtt{o}, \sigma_o, K_{\mathbf{idle}}, Q)$ and $\Delta, \mathcal{G}_2 \vdash_R ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid \mathbf{rebind}\ e.f = e'; s\ \}, Q')$ and $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. Suppose $\sigma_{\mathtt{o}'} = \overline{T\ x\ w}; \theta$ and $\sigma'_{\mathtt{o}'} = \overline{T'\ x'\ w'}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. Then, $\Delta', \mathcal{G}'_2 \vdash_R \mathbf{rebind}\ e.f = e'; s$ and $\mathcal{G}_2 = \mathcal{G}'_2 \uplus \mathcal{G}''_2$ and $\mathcal{G}''_2$ is the set of cogs in Q'. By (T-Rebind) $\Delta' \vdash_R e : (\mathtt{I}, \mathbb{r})$ and $\Delta', \mathcal{G}''_2 \vdash_R e' : T$ and $T\ f \in ports(\mathtt{I})$ and coloc($\mathbb{r}, \Delta'(\mathbf{this})$) stating the belonging to the same cog. By assumption $\mathtt{o} = [\![e]\!]_{(\sigma_{\mathtt{o}'} \circ \sigma'_{\mathtt{o}'})}$ and $v = [\![e']\!]_{(\sigma_{\mathtt{o}'} \circ \sigma'_{\mathtt{o}'})}$, then by Lemma 4.5.3 we have that $\Delta' \vdash_R v : \Delta'(x') = T$. Then, trivially $\Delta, \mathcal{G}_1 \vdash_R ob(\mathtt{o}, \sigma_\mathtt{o}[f \mapsto v], K_{\mathbf{idle}}, Q)$ and $\Delta, \mathcal{G}_2 \vdash_R ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid s\ \}, Q')$. We conclude by (T-Config).

□

Theorem 3.2.4 (Correction of rebindings)**.** *If* $\Delta, \mathcal{G} \vdash_R N$*, then for all objects* $ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q) \in N$ *and* $s = (\textbf{rebind}\ e.f = e'; s')$ *there exists an object* $ob(\mathsf{o}', \sigma'', K_{\textbf{idle}}, Q')$ *such that* $[\![e]\!]_{(\sigma \circ \sigma')} = \mathsf{o}'$ *and* $\sigma(\textbf{cog}) = \sigma''(\textbf{cog})$.

Proof. The proof is done by induction on the structure of N.
Let $N = ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q)$ and $s = (\textbf{rebind}\ e.f = e'; s')$. By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\ \sigma' \mid \textbf{rebind}\ e.f = e'; s'\ \}, Q)$. Suppose $\sigma = \overline{T\ x\ v}; \theta$ and $\sigma' = \overline{T'\ x'\ v'}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. Notice that, by the well-typedness of the configuration we also have that $\Delta', \emptyset \vdash_R \sigma$ and $\Delta', \emptyset \vdash_R \sigma'$. By the definition of substitution we have that $\sigma(\textbf{this}) = \mathsf{o}$ and let $\sigma(\textbf{cog}) = \mathsf{c}$. By (T-Object) $\Delta', \mathcal{G}_1 \vdash_R \textbf{rebind}\ e.f = e'; s'$ and $\Delta', \mathcal{G}_2 \vdash_R Q$ where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By rules (T-Weak2), (T-Comp) and (T-Rebind) it means that $\Delta', \mathcal{G}'_1 \vdash_R \textbf{rebind}\ e.f = e'$ and $\Delta', \mathcal{G}''_1 \vdash_R s'$ where $\mathcal{G}_1 = \mathcal{G}'_1 \uplus \mathcal{G}''_1$. By the premise of (T-Rebind) and (T-Weak1) and (T-Exp) we have that $\Delta', \emptyset \vdash_R e : (\mathtt{I}, \mathbb{r})$ and f is a port of $\mathtt{I}$. Let $[\![e]\!]_{(\sigma \circ \sigma')} = v$ where v is a value produced by the runtime syntax, since we are dealing with runtime configurations. By type preservation lemma this means that $\Delta', \emptyset \vdash_R v : (\mathtt{I}, \mathbb{r})$. By analysing the syntax of values and by the fact that v is well-typed with $(\mathtt{I}, \mathbb{r})$ the only possible case is v being an object identifier o'. But then, since there exists the object identifier, by the operational semantics rules (New-Object) or (New-Cog-Object), it means that the object is already created, and in addition it is well-typed. Let it have a general object structure denoted with $ob(\mathsf{o}', \sigma'', K_{\textbf{idle}}, Q')$.

We distinguish the following two cases:

- $\mathsf{o}' = \mathsf{o}$: this means that the object is rebinding its own port. Trivially, the cog is the same.
- $\mathsf{o}' \neq \mathsf{o}$: this means that the object o is rebinding the port f of another object o'. By typing rule (T-Rebind) and (T-Weak1) we have that the predicate coloc is true. Namely, $\text{coloc}(\mathbb{r}, \Delta'(\textbf{this}))$, which by the premise of coloc we have that the cog of $\mathbb{r}$ is the same as the cog of **this**, namely c. This means that $\sigma(\textbf{cog}) = \sigma''(\textbf{cog})$.

The inductive case for $N = ob(\mathsf{o}, \sigma, \{\ \sigma' \mid \textbf{rebind}\ e.f = e'; s'\ \}, Q)\ N'$ follows by the base case and by applying (T-Config).

□

Theorem 3.2.5 (Correction of synchronous method calls)**.** *If* $\Delta, \mathcal{G} \vdash_R N$*, then for all objects* $ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q) \in N$ *and* $s = (x = e.\mathtt{m}(\overline{e}); s')$ *there exists an object* $ob(\mathsf{o}', \sigma'', K_{\textbf{idle}}, Q')$ *such that* $[\![e]\!]_{(\sigma \circ \sigma')} = \mathsf{o}'$ *and* $\sigma(\textbf{cog}) = \sigma''(\textbf{cog})$.

Proof. The proof is done by induction over the structure of N.

Let $N = ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q)$ and $s = (x = e.\mathtt{m}(\overline{e}); s')$. By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\ \sigma' \mid x = e.\mathtt{m}(\overline{e}); s'\ \}, Q)$. Suppose $\sigma = \overline{T\ x\ v}; \theta$ and $\sigma' = \overline{T'\ x'\ v'}; \theta'$, and let $\Delta' = \Delta, \overline{x} : \overline{T}, \overline{x'} : \overline{T'}$. Notice that, by the well-typedness of the configuration we also have that $\Delta', \emptyset \vdash_R \sigma$ and $\Delta', \emptyset \vdash_R \sigma'$. By the definition of substitution we have that $\sigma(\mathbf{this}) = \mathsf{o}$ and let $\sigma(\mathbf{cog}) = \mathsf{c}$. By (T-Object) $\Delta', \mathcal{G}_1 \vdash_R x = e.\mathtt{m}(\overline{e}); s'$ and $\Delta', \mathcal{G}_2 \vdash_R Q$ where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By rules (T-Weak2), (T-Comp) and (T-Rebind) it means that $\Delta', \mathcal{G}'_1 \vdash_R x = e.\mathtt{m}(\overline{e})$ and $\Delta', \mathcal{G}''_1 \vdash_R s'$ where $\mathcal{G}_1 = \mathcal{G}'_1 \uplus \mathcal{G}''_1$. By (T-Assign) we have $\Delta', \mathcal{G}'_1 \vdash_R e.\mathtt{m}(\overline{e}) : T$ and $\Delta'(x) = T$ for some type T. By (T-SCall) we have that $T = \rho(T')$ and $\mathcal{G}'_1 = \emptyset$ where ρ is the substitution of the formal return type T' to the actual return type and T. Since the synchronous method call is well-typed, by the premise of (T-SCall) and (T-Weak1) we have that $\Delta' \vdash_R e : (\mathtt{I}, \rho(\mathbb{I}))$. Let $[\![e]\!]_{(\sigma \circ \sigma')} = v$ where v is a value produced by the runtime syntax, since we are dealing with runtime configurations. By type preservation lemma this means that $\Delta', \emptyset \vdash_R v : (\mathtt{I}, \mathbb{I})$. By analysing the syntax of values and by the fact that v is well-typed with $(\mathtt{I}, \rho(\mathbb{I}))$ the only possible case is v being an object identifier o'. But then, since there exists the object identifier, by the operational semantics rules (New-Object) or (New-Cog-Object), it means that the object is already created, and in addition it is well-typed. Let it have a general object structure denoted with $ob(\mathsf{o}', \sigma'', K_{\mathbf{idle}}, Q')$. The rest of the proof follows exactly the same line as the correction of rebinding proof where again by the premise of (T-SCall) we have that the predicate $\mathrm{coloc}(\rho(\mathbb{I}), \Delta'(\mathbf{this}))$ is true.

The inductive case for $N = ob(\mathsf{o}, \sigma, \{\ \sigma' \mid x = e.\mathtt{m}(\overline{e}); s'\ \}, Q)\ N'$ follows by the base case and by applying (T-Config).

□

Conclusions, Related and Future Work for Part I

In Part I we presented a type system for a component-based calculus. The calculus we adopt is inspired by [75], the latter being an extension of the Abstract Behavioural Specification (ABS) language [60]. This extension consists of the notions of **port**s and **rebind** operations.

Ports and fields differ in a conceptual meaning: ports are the access points to the functionalities provided by the environment whether fields are used to save the inner state of an object. Fields are modified freely by an assignment, only by the object that owns them, whilst ports are modified by a **rebind** operation by *any* object in the same cog.

There are two consistency issues involving ports: *i*) ports cannot be modified while in use; this problem is solved in [75] by combining the notions of ports and critical section; *ii*) it is forbidden to modify a port of an object outside the cog; this problem is solved in the present dissertation by designing a type system that guarantees the above requirement. The type system tracks an object's membership to a certain cog by adopting group records. Rebind statement is well-typed if there is compatibility of groups between objects involved in the operation.

In the remainder we discuss the related works by dividing them in three separate paragraphs respectively for, ABS, component extension and type systems. We conclude with future work.

ABS Language Related Work Actor-based ABS language is designed for distributed object-oriented systems. It integrates *concurrency* and *synchronisation* mechanisms with a *functional* language. Actors, called concurrent object groups

*cog*s, are dynamic collections of collaborating objects. Cogs offer consistency by guaranteeing that at most one method per cog is executing at any time.

There are several concurrent object-oriented models that integrate concurrent objects and actors, the same as cogs in ABS language, which adopt asynchronous communication and usage of futures as first-class values, like [1, 5, 13, 25, 52, 61, 114]. As stated in [60], the concurrency model of ABS is a generalisation of the concurrency model of Creol [62] passing from one concurrent object to concurrent groups of objects, implemented in JCoBox [102], which is its Java extension. Creol is based on asynchronous communication and hence future values are present. Futures are adopted in particular in [13, 114] whereas asynchronous calculi for distributed systems are adopted in [1, 5, 25, 61] and in [2] which is mostly oriented in verification of various properties.

Despite the concurrency basically performed by the communication among different cogs, an important and typical feature of ABS is its synchronisation mechanism inside one cog, that permits only one object at a time to be active. The cooperation of objects inside the cog is similar to the so called *cooperative scheduling* in Creol where the concurrent cogs are merely the concurrent objects. As stated in [60] cogs in ABS can be compared to *monitors* in [53]. However, differently from monitors, there is no explicit signalling. It is possible to encode monitors in the language, as stated in [61].

The concurrent object calculus in [12] adopts both synchronous and asynchronous method calls, having different semantics. This is similar to the component extension and differs from ABS where a synchronous method call between two different cogs reduces into an asynchronous one, whether in the component model it is not defined which means it reduces to `error`.

Components Related Work Most component models [4, 7, 11, 14, 29, 77, 81, 88] have a notion of component different from that of object. The resulting language is structured in two separate layers, one using objects for the main execution of the program and the other using components for the dynamic reconfiguration. This separation makes it harder for the (unplanned) dynamic reconfiguration requests to be integrated in the program's workflow. For example, models like Click [73] do not allow runtime reconfigurations at all, whether OSGi model [4] allows addition of classes and objects but does not allow modification of components, whether the Fractal model [14] allows modifications by performing new bindings, which allow addition of components.

However, there are other component models that go towards integrating the

notions of objects and components, thus having a more homogeneous semantics. For example, models like Oz/K [77] and COMP [76] offer a more unified way of presenting objects and components. However, both Oz/K and COMP deal with dynamic reconfigurations in a very complex way.

The component model we adopt in the present work, inspired by [75], has a unified description of objects and components by exploiting the similarities between them. This brings in several benefits with respect to previous component models: *i*) the integration of components and objects strongly simplifies the reconfiguration requests handling, *ii*) the separation of concepts (component and object, port and field) makes it easier to reason about them, for example, in static analysis, and *iii*) ports are useful in the deployment phase of a system by facilitating, for example, the connection to local communication.

Type Systems Related Work The type system for components presented in Chapter 2 is an extension of the type system of ABS which is an extension of the type system for Featherweight Java [58], which is *nominal*. However, differently from both FJ and ABS, we also adopt the *structural* approach, in particular in the subtyping relation defined in Section 2.2. Differently from FJ and similarly to ABS, objects are typed with interfaces, and not classes, by thus having a neat distinction between the two concepts which enables abstraction and encapsulation. Creol's type system has more characteristic in common with our type system. It tracks types which are implicitly associated to untyped futures by using an effect system as in [78]. This allows more flexibility in reusing future variables with different return type. This feature is not present either in ABS or in our type system, where future variables have explicit future types.

Various other type systems have been designed for components. The type system in [115] categorises references to be either Near (i.e., in the same cog), Far (i.e., in another cog) or Somewhere (i.e., we don't know). The goal is to automatically detect the distribution pattern of a system by using the inference algorithm, and also control the usage of synchronous method calls. It is more flexible than our type system since the assignment of values of different cogs is allowed, but it is less precise than our analysis: consider two objects o_1 and o_2 in a cog c_1, and another one o_3 in c_2; if o_1 calls a method of o_3 which returns o_2, the type system will not be able to detect that the reference is Near. In [3] the authors present a tool to statically analyse concurrency in ABS. Typically, it analyses the concurrency structure, namely the cogs, but also the synchronisation between method calls. The goal is to get tools that analyse concurrency for actor-based

concurrency model, instead of the traditional thread-based concurrency model. The relation to our work is in the analysis of the cog structure.

On the other hand, our type system has some similarities with the type system in [23] which is designed for a process calculus with *ambients* [24], the latter roughly corresponding to the notion of components in a distributed scenario. The type system is based on the notion of group which tracks communication between ambients as well as their movement. However, groups in [23] are a "flat" structure whether in our framework we use group records defined recursively; in addition, the underlying language is a process calculus, whether ours is a concurrent object-oriented one. As object-oriented languages are concerned, another similar work to ours is the one on *ownership types* [26], where basically, a type consists of a class name and a context representing object ownership: each object owns a context and is owned by the context it resides in. The goal of the type system is to provide alias control and invariance of aliasing properties, like role separation, restricted visibility etc. [54].

Future Work Our type system can be seen as a technique for tracking membership of a component to a group or a context or to similar notions. Hence it can also be applied to other component-based languages [4, 11, 14, 29] to deal with dynamic reconfiguration and rebindings. Or, more specifically, in business processes and web-services languages [87, 92] to check (dynamic) binding of input or output ports and guarantee consistencies of operations, or in [74] to deal with dynamic reconfiguration of connectors which are created from primitive channels and resemble to ports in our setting. In addition, the group-based type system can be applied not only to tracking membership of an object to a cog, but also to detect misbehaviours, like deadlock, as shown in [48, 49]. So, first of all we want to explore the various areas in which the type system can be applied. Second, as discussed in 2.4 our current approach imposes a restriction on assignments, namely, it is possible to assign to a variable/field/port only an object belonging to the same cog. We plan to relax this restriction following a similar idea to the one proposed in [49], where instead of having just one group associated to a variable, it is possible to have a set of groups. Third, we want to deal with runtime misbehaviours, like deadlocks or livelocks. The idea is to use group information to analyse dependencies between groups. We take inspiration from [48].

Part II

Safe Communication by Encoding

Introduction to Part II

In complex distributed systems, participants willing to communicate should previously agree on a protocol to follow. The specified protocol describes the *types* of messages that are exchanged as well as their *direction*. In this context, *session types* came into play. Session types are a formalism proposed as a foundation to describe and model structured-communication based programming. They were originally introduced nearly 20-years ago in [104] and later in [55] for a polyadic π- calculus, which is the most successful setting. After that, session types have been developed for various paradigms, like (multi-threaded) functional programming [47, 108, 111], component-based object systems [107], object-oriented languages [18, 38–40, 46], Web Services and Contracts, WC3-CDL a language for choreography [21, 89] etc.

Since their appearance, many extensions have been made to session types. An important research direction is the one that brings from *dyadic* sessions types [36, 55, 104, 109, 118], describing communication between only two participants, to *multiparty* session types [56], where the number of participants can be greater than just two, or where the number of participants can be variable, namely participants can dynamically join and leave [35] or to choreographies and global types [21,89]. In dyadic session types, different typing features have been added. Subtyping relation for (recursive) session types is added in [43]. Bounded polymorphism is added in [44] as a further extension to subtyping, and parametric polymorphism is added in [15]. The authors in [90] add higher-order primitives in order to allow not only the mobility of channels but also the mobility of processes.

Session types are an "ad hoc" means to describe a *session*, which is defined

in [36] as: “a session is a logical unit of data that are exchanged between two or more interacting participants”. Session types describe a protocol as a type abstraction, guaranteeing *privacy* as well as *communication safety* and *session fidelity*. Privacy property requires the session channel to be owned only by the communicating parties. Communication safety is an extension to a structured sequence of interactions of the standard type safety property in the (polyadic) π-calculus: it is the requirement that the exchanged values have the expected type. Instead, session fidelity is a typical property of sessions and is the requirement that the session channel has the expected structure.

Session types are defined as a sequence of input and output operations, explicitly indicating the types of messages being transmitted. This structured *sequentiality* of operations is a feature that makes session types suitable to model protocols in distributed scenarios.

However, they offer more flexibility than just performing inputs and outputs: they permit *choice*, internal and external one. Branch and select are typical type (and also term) constructs in the theory of session types, the former being the offering of a set of alternatives and the latter being the selection of one of the possible options being offered.

As mentioned above, session types guarantee privacy, communication safety and session fidelity. Privacy is guaranteed since session channels are known only to the agents involved in the communication. Such communication proceeds without any mismatch of direction and of message type, which guarantees session fidelity. In order to achieve communication safety, a binary session channel is split by giving rise to two opposite endpoints, each of which is owned by one of the agents. These endpoints are required to have dual behaviour and thus have dual types. So, *duality* is a fundamental concept in the theory of session types as it is the ingredient that guarantees communication safety and session fidelity. In order to better understand session types and the notion of duality, let us consider a simple example: a client and a server communicating over a session channel. The endpoints x and y of the channel are owned by the client and the server exclusively and should have dual types. If the type of channel endpoint x is

$$?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

– meaning that the process listening on channel x receives an integer value followed by another integer value and then sends back a boolean value – then the

type of channel endpoint y should be

$$!\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

– meaning that the process listening on channel y sends an integer value followed by another integer value and then waits to receive back a boolean value – which is exactly the dual type.

There is a precise moment at which a session is established between two agents. It is called *connection*, when a fresh (private) session channel is created and its endpoints are bound to each communicating process. The connection is also the moment when the duality, hence compliance of two session types, is verified. In order to perform a connection, primitives for session channel creation, like `accept/request` or (νxy), are added to the syntax of processes [55, 104, 109, 118].

Another important issue concerning session types is that of session channel transmission, namely *delegation*. If one participant in a session wants to delegate a subtask to an agent in another session, he sends his session endpoint to this agent. This is transparent to the other participant. Generally, primitives like `throw/catch` are added to the syntax of processes [55, 104, 118].

Session types and session primitives are added to the syntax of standard π-calculus types and terms, respectively. In doing so, sessions give rise to additional separate syntactic categories. Hence, the syntax of types need to be split into separate syntactic categories, one for session types and the other for standard π-calculus types [43, 55, 104, 118] (this often introduces a duplication of type environments, as well).

In this part we try to understand to which extent this redundancy is necessary, in the light of the following similarities between session constructs and standard π-calculus constructs.

Consider the session type previously mentioned $?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$. This type assigned to a session channel (actually, as we said above, to one of its endpoints) describes a structured sequence of inputs and outputs by specifying the type of messages that it can transmit. This recalls the *linearised* channels [70], which are channels used multiple times for communication but *only* in a sequential manner. Linearised types can be encoded, as shown in [70], into linear types –i.e., channel types used *exactly once* and recursive types. However, linearised channels seem not to be as expressive as the session channels, since they have the same carried type and the same direction of communication.

Let us now consider branch and select. These constructs added on both the syntax of types and of processes, give more flexibility by offering and selecting a range of possibilities. This brings in mind an already existing type construct in the typed π- calculus, namely the *variant* type and the *case* process [101].

Other analogies between session types and π-types concern connection and duality. Connection can be seen as the *restriction* construct, since both are used to create and bind a new private session channel to the communication parties. As mentioned above, duality is checked when connection takes place. Duality describes the split of behaviour of session channel endpoints. This reminds us of the split of *capabilities*: once a new channel is created, it can be used by two communicating processes owning the opposite capability each.

The goal of this work is to investigate further the relation between session types and standard π-types and to understand the expressive power of session types and to which extent they are more sophisticated and complex than standard π- calculus channel types. There is a plethora of papers on session types in which session types are always taken as primitives. However, by following Kobayashi [66, 69], we define an interpretation of session types into standard π-types and by exploiting this encoding, session types and all their theory are shown to be derivable from the well-known theory of the typed π-calculus. For instance, basic properties such as Subject Reduction and Type Safety in session types become straightforward corollaries of the encoding and the corresponding properties in the typed π-calculus.

Intuitively, a session channel is interpreted as a linear channel transmitting a pair consisting of the original message and a new linear channel which is going to be used for the continuation of the communication. The contribution of this encoding is meant to be foundational: we show that it does permit to derive session types and their basic properties; and in the next Part of the dissertation, we show that it is robust, by examining some extensions of session types.

While the encoding first introduced by Kobayashi was generally known, its strength, robustness, and practical impact were not. Probably, the reasons for this are the following:

(a) Kobayashi did not prove any properties of the encoding and did not investigate its robustness;

(b) as certain key features of session types do not clearly show up in the encoding, the faithfulness of the encoding was unclear.

A good example for (b) is duality. In ordinary π-calculus, in contrast, there is no notion of duality on types. Indeed, in the encoding, dual session types for example, the branch type and the select type, are mapped using the same type for example, the variant type. Basically, dual session types will be mapped onto linear types that are identical except for the outermost I/O tag – duality on session types boils down to the duality between input and output capability of channels.

The contribution of this work does not imply that session types are useless, as they are very useful from a programming point of view. The work just tells us that, at least for the dyadic sessions and their properties, session types and session primitives may be taken as macros.

Roadmap to Part II The rest of Part II is structured as follows. Chapter 4 gives a detailed overview on the standard π-calculus. Chapter 5 gives a detailed overview on the π-calculus with sessions. These chapters introduce both the statics and the dynamics of the calculi. Chapter 6 presents the encoding of both session types and session processes and gives the theoretical results that follow from the encoding.

CHAPTER 4

Background on π-Types

The π-calculus [84, 101] is an evolution of the Calculus of Communicating Systems (CCS) invented by Robin Milner in the late '70s [82, 83] which is a seminal work in the theory of concurrency and is at the basis of the process calculi.

In this chapter we present the typed polyadic π-calculus. We start by giving the syntax of the terms and the operational semantics, then we introduce the syntax of types and the typing rules.

4.1 Syntax

The syntax of standard (polyadic) π-calculus is given in Fig. 4.1. Let P, Q range over processes, x, y over variables, l over labels and v over values, i.e., variables, boolean values (and possibly other ground values like integers, strings etc.) and variant values, which are labelled values. For our purposes, we adopt the polyadic π-calculus where a tuple of values denoted by $\tilde{v}$ can be sent and replaces a tuple of variables $\tilde{y}$. We denote with $\tilde{\cdot}$ an ordered sequence of elements "$\cdot$".

The output process $x!\langle\tilde{v}\rangle.P$ sends a tuple of values $\tilde{v}$ on channel x and proceeds as process P; the input process $x?(\tilde{y}).P$ performs the opposite operation, it receives on channel x a tuple of values and substitutes them for the placeholders $\tilde{y}$ in P. The process **if** v **then** P **else** Q is the standard one. The process $P \mid Q$ is the parallel composition of processes P, Q. The process $\mathbf{0}$ is the terminated process.

$P, Q ::=$	$x!\langle\tilde{v}\rangle.P$	output
	$x?(\tilde{y}).P$	input
	if v **then** P **else** Q	conditional
	$P \mid Q$	composition
	$\mathbf{0}$	inaction
	$(\nu x)P$	channel restriction
	case v **of** $\{l_i\_x_i \triangleright P_i\}_{i\in I}$	case
$v ::=$	x	variable
	`true	false`	boolean values
	l_v	variant value

Figure 4.1: Syntax of the standard π-calculus

The process $(\nu x)P$ creates a new variable x and binds it with scope P. The process **case** v **of** $\{l_i\_x_i \triangleright P_i\}_{i\in I}$ offers different behaviours depending on the (labelled) value v. Labels l_i for all i in some set I are all different, and their order is not important.

We say that a process is *prefixed* in a variable x, if it is either of the form $x!\langle v\rangle.P$ or of the form $x?(y).P$. For simplicity, we will avoid triggering the terminated process, so we will omit $\mathbf{0}$ from any process in the remainder of the dissertation. We use $FV(P)$ to denote the set of free variables in P, $BV(P)$ to denote the bound ones and $Vars(P) = FV(P) \cup BV(P)$ to denote the set of all variables in P. The bound variables are: in $(\nu x)P$ variable x is bound in P, in $x?(y).P$ variable y is bound in P and in **case** v **of** $\{l_i\_x_i \triangleright P_i\}_{i\in I}$ every variable x_i is bound in P_i. If not under the previous cases, then the variable is a free one. We will use *substitution* and *alpha-conversion* as defined in [101]. We use $P[x/y]$ as a notation for process P where every occurrence of the free variable y is substituted by variable x. As usual in the π-calculus, substitution is coupled with avoiding the unintended variable capture by the binders of the calculus. In order to achieve this, the alpha-conversion of variables is performed, which performs a renaming of bound variables in a process.

Definition 4.1.1 (Alpha-convertibility)**.** *The following give a procedure for substituting and renaming variables in a process.*

1. *If a variable x does not occur in a process P, then $P[x/y]$ is the process obtained by replacing every occurrence of y by x in P.*

2. *An* alpha conversion *of the bound variables in a process P is the replacement of a subterm*

- $x?(y).Q$ *by* $x?(w).Q[w/y]$ *or*
- $(\nu y)Q$ *by* $(\nu w)Q[w/y]$ *or*
- $\mathbf{case}\ v\ \mathbf{of}\ \{l_i\_y_i \triangleright Q_i\}_{i\in I}$ *by* $\mathbf{case}\ v\ \mathbf{of}\ \{l_i\_w_i \triangleright Q_i[w_i/y_i]\}_{i\in I}$

where in each case w does not occur in Q or any w_i does not occur in Q_i.

3. *Processes P and Q are* alpha-convertible $P =^{\alpha} Q$ *if Q can be obtained from P by a finite number of changes in the bound variables.*

However, in this work we adopt the Barendregt variable convention, namely that all variables in bindings in any mathematical context are pairwise distinct and distinct from the free variables.

4.2 Semantics

Before presenting the operational semantics, we introduce the notion of *structural congruence* $\equiv$ for the standard π-calculus as defined in [101]. It is the smallest congruence relation on processes that satisfies the following axioms.

$$\begin{array}{rcll} P \mid Q & \equiv & Q \mid P & \\ (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\ P \mid \mathbf{0} & \equiv & \mathbf{0} \mid P & \\ (\nu x)\mathbf{0} & \equiv & \mathbf{0} & \\ (\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P & \\ (\nu x)P \mid Q & \equiv & (\nu x)(P \mid Q) & (x \notin FV(Q)) \end{array}$$

Figure 4.2: Structural congruence for the standard π-calculus

The first three axioms state respectively that the parallel composition of processes is commutative, associative and uses process $\mathbf{0}$ as the neutral element. The last three axioms state respectively that one can safely add or remove any restriction to the terminated process, the order of restrictions is not important and the last one called *scope extrusion* states that one can extend the scope of the restriction to another process in parallel as long as the restricted variable is not present in the new process included in the restriction, side condition $x \notin FV(Q)$. By the convention of names adopted, this side condition is redundant, However, for more clarity, we report the condition as part of the last axiom.

The semantics of the standard π-calculus is given in Fig 4.3. It is a binary reduction relation $\rightarrow$ defined over processes. We use $\rightarrow^*$ to denote the reflexive and

transitive closure of $\rightarrow$. We call a *redex* a process of the form $(x!\langle\tilde{v}\rangle.P \mid x?(\tilde{y}).Q)$.

$$
\begin{array}{rc}
(\text{R}\pi\text{-Com}) & x!\langle\tilde{v}\rangle.P \mid x?(\tilde{y}).Q \rightarrow P \mid Q[\tilde{v}/\tilde{y}] \\
(\text{R}\pi\text{-Case}) & \textbf{case}\, l_j\_v\, \textbf{of}\, \{l_i\_x_i \triangleright P_i\}_{i\in I} \rightarrow P_j[v/x_j] \quad j \in I \\
(\text{R}\pi\text{-IfT}) & \textbf{if}\ \texttt{true}\ \textbf{then}\ P\ \textbf{else}\ Q \rightarrow P \\
(\text{R}\pi\text{-IfF}) & \textbf{if}\ \texttt{false}\ \textbf{then}\ P\ \textbf{else}\ Q \rightarrow Q \\
(\text{R}\pi\text{-Res}) & \dfrac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q} \\
(\text{R}\pi\text{-Par}) & \dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \\
(\text{R}\pi\text{-Struct}) & \dfrac{P \equiv P',\ P' \rightarrow Q',\ Q' \equiv Q}{P \rightarrow Q}
\end{array}
$$

Figure 4.3: Semantics of the standard π-calculus

Rule (Rπ-Com) is the communication rule: the process on the left sends a tuple of values $\tilde{v}$ on x, while the process on the right receives the values and substitutes them for the placeholders in $\tilde{y}$. Rule (Rπ-Case), is also called a *case normalization* since it does not require a counterpart to reduce. The **case** process reduces to P_j substituting x_j with the value v, if the label l_j is selected. This label should be among the offered labels, namely $j \in I$. Rules (Rπ-IfT) and (Rπ-IfF) state that the conditional process **if** v **then** P **else** Q reduces either to P or to Q depending on whether the value v is `true` or `false`, respectively. Rules (Rπ-Res) and (Rπ-Par) state that communication can happen under restriction and parallel composition, respectively. Rule (Rπ-Struct) is the standard one, stating that reduction can happen under the structural congruence, previously introduced.

4.3 π-Types

In this section we introduce the π-types, in particular we focus on the *linear channel types* and the *variant type* used in the encoding presented in Chapter 6.

Linearity in the standard typed π-calculus has the following meaning: a linear channel must be used *exactly* once, and then is no longer available for usage.

The syntax of (linear) π-types is defined in Fig. 4.4. Let α range over capabilities, τ over channel types and T range over types. A *capability* can be an input

α	::=	i	input capability
		o	output capability
		$\sharp$	connection capability
τ	::=	$\ell_\alpha[\widetilde{T}]$	channel type used linearly in α
		$\emptyset[\widetilde{T}]$	channel with no capability
		$\cdots$	other channel types
T	::=	τ	channel type
		$\langle l_i\_T_i\rangle_{i\in I}$	variant type
		`Bool`	boolean type
		$\cdots$	other constructs

Figure 4.4: Syntax of linear π-types

capability i, an output capability o, and a connection capability $\sharp$, meaning both input and output. For our goal, we adopt only linear channel types. Linear types are $\ell_{\mathsf{i}}\ [\widetilde{T}]$, $\ell_{\mathsf{o}}\ [\widetilde{T}]$ and $\ell_{\sharp}\ [\widetilde{T}]$. These types specify not only the *capability* or said differently, the *polarity* of the channel, namely how the channel is intended to be used, but also its *multiplicity*, namely the channel should be used *exactly once*. In particular, the type $\ell_{\mathsf{i}}\ [\widetilde{T}]$ is the type assigned to a channel that can be used exactly once for receiving a sequence of values of type $\widetilde{T}$; the type $\ell_{\mathsf{o}}\ [\widetilde{T}]$ is the type assigned to a channel that can be used exactly once for sending a sequence of values of type $\widetilde{T}$, and finally the type $\ell_{\sharp}\ [\widetilde{T}]$ is the type assigned to a channel that can be used exactly once for receiving and once for sending a sequence of values of type $\widetilde{T}$. In addition, we denote with $\emptyset[\widetilde{T}]$ the type of a channel with no capabilities, namely the channel cannot be used at all. – We adopt the latter in order to enable encoding of session types, as we will see in Chapter 6.– Types include channel types τ; the variant type $\langle l_i\_T_i\rangle_{i\in I}$ and `Bool` type. The variant type is a labelled form of disjoint union of types. The labels ranging in a set I are all distinct. The order of the components does not matter. The `Bool` type is the type assigned to boolean values, `true` and `false`. We include only the `Bool` type just for simplicity. One can add to the syntax of types any other standard constructs of the π-calculus. For example, other ground types like `Int`, `String` etc., or non-linear channel types that can be used an unbounded number of times (see [101]). We will use these types in examples.

In order to better understand linearity in the linearly typed π- calculus, we present the following simple examples. If x and y have types $\ell_{\mathsf{o}}\ [T]$ and $\ell_{\mathsf{i}}\ [S\,]$,

respectively then the following processes

$$x!\langle v\rangle.P \qquad y?(z).Q$$

respect the linearity of channels x and y, if $x \notin FV(P)$ and $y \notin FV(Q)$. Instead, the processes

$$x!\langle v\rangle.P \mid x!\langle w\rangle.Q \qquad x!\langle v\rangle.x!\langle w\rangle.R$$

do not respect the linearity of x since it is used twice, to output a value v and another value w.

4.4 π-Typing

A typing context is a partial function from variables to types. Namely,

$$\Gamma \quad ::= \quad \emptyset \quad | \quad \Gamma,\ x : T$$

Before presenting the typing rules, we present some operations on types and on] contexts, on which the type systems relies on.

The predicates lin and un on the standard π-types are defined as follows:

$$\begin{array}{ll} \mathsf{lin}(T) & \text{if } T = \ell_\alpha\ [\widetilde{T}] \text{ or } (T = \langle l_i\_T_i\rangle_{i\in I} \text{ and } \exists j \in I.\ \mathsf{lin}(T_j)) \\ \mathsf{un}(T) & \text{otherwise} \end{array}$$

So, a type is linear if it is a linear channel type or if it is a variant type containing a linear type in at least one of its branches; otherwise it is unrestricted. These predicates are extended to the typing contexts in the expected way:

$$\begin{array}{ll} \mathsf{lin}(\Gamma) & \text{if } \exists (x : T) \in \Gamma \text{ such that } \mathsf{lin}(T) \\ \mathsf{un}(\Gamma) & \text{otherwise} \end{array}$$

The typing rules make use of *combination of types* and *combination of typing contexts*, which are defined by the equations in Fig. 4.5. We denote the operator of combination with $\uplus$. This operator is associative and hence we do not use brackets. The combination of linear types states that a linear input channel type combined with a linear output channel type results in a linear connection channel type, whenever the tuple of carried types is the same. The combination of unrestricted types is defined only if the two types combined are the same, otherwise it is undefined. Notice that, in particular the unrestricted combination gives

$\emptyset[\widetilde{T}] \uplus \emptyset[\widetilde{T}] \stackrel{\texttt{def}}{=} \emptyset[\widetilde{T}]$. The combination of typing contexts is defined by following the same line as that of combination of types. The type of a variable x in $\Gamma_1 \uplus \Gamma_2$ is the combination of the type of x in Γ_1 and the type of $x$in Γ_2 if x is both in Γ_1 and Γ_2; otherwise, it is the type assumed either in Γ_1 or in Γ_2, where defined, otherwise `undef`. The combination of typing contexts $\Gamma_1 \uplus \Gamma_2$ is extended to a tuple of environments $\Gamma_1 \uplus \cdots \uplus \Gamma_n$ and we denote this for simplicity as $\widetilde{\Gamma}$.

Combination of π-types $T \uplus T = T$

$$\begin{array}{rcll} \ell_{\mathsf{i}}\,[\widetilde{T}] \uplus \ell_{\mathsf{o}}\,[\widetilde{T}] & \stackrel{\texttt{def}}{=} & \ell_\sharp\,[\widetilde{T}] & \\ T \uplus T & \stackrel{\texttt{def}}{=} & T & \text{if } \mathsf{un}(T) \\ T \uplus S & \stackrel{\texttt{def}}{=} & \texttt{undef} & \text{otherwise} \end{array}$$

Combination of typing contexts $\Gamma \uplus \Gamma = \Gamma$

$$(\Gamma_1 \uplus \Gamma_2)(x) \stackrel{\texttt{def}}{=} \begin{cases} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are defined} \\ \Gamma_1(x) & \text{if } \Gamma_1(x), \text{ but not } \Gamma_2(x), \text{ is defined} \\ \Gamma_2(x) & \text{if } \Gamma_2(x), \text{ but not } \Gamma_1(x), \text{ is defined} \\ \texttt{undef} & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are undefined} \end{cases}$$

Figure 4.5: Combination of π-types and typing contexts

We define the duality of π-types to be merely the duality on the capability of the channel. Formally, it is defined by the equations in Fig 4.6.

$$\begin{array}{rcl} \overline{\ell_{\mathsf{i}}\,[\widetilde{T}]} & = & \ell_{\mathsf{o}}\,[\widetilde{T}] \\ \overline{\ell_{\mathsf{o}}\,[\widetilde{T}]} & = & \ell_{\mathsf{i}}\,[\widetilde{T}] \\ \overline{\emptyset[\widetilde{T}]} & = & \emptyset[\widetilde{T}] \end{array}$$

Figure 4.6: Type duality for linear types

Typing judgements are of two forms: $\Gamma \vdash v : T$ stating that the value v is of type T in the typing context Γ, and $\Gamma \vdash P$ stating that process P is well-typed in the typing context Γ. The typing rules for the π-calculus with linear channel types are given in Fig 4.7. Rule (Tπ-Var) states that a variable is of type the one assumed in the typing context. Moreover, the typing context is weakened by unrestricted type assumptions. Rule (Tπ-Val) states that a boolean value, either `true` or `false`, is

of type `Bool`. Again, the typing context is weakened by unrestricted type assumptions. Rule (Tπ-Inact) states that the terminated process $\mathbf{0}$ is well-typed in every unrestricted typing context. Rule (Tπ-Par) states that the parallel composition of two processes is well-typed in the combination of typing contexts that are used to typecheck each of the processes. There are two typing rules for the restriction process, namely rule (Tπ-Res1) and rule (Tπ-Res2). Rule (Tπ-Res1) states that the restriction process $(\nu x)P$ is well-typed if process P is well-typed under the same typing context augmented with $x : \ell_\alpha\ [\widetilde{T}]$. Notice that, since the type assumption on variable x is needed to type P and it is of linear channel type, this implies that x is used in P. Rule (Tπ-Res2) states that the restriction $(\nu x)P$ is well-typed if P is well-typed and variable x is not used in P. This rule is needed in the standard π-calculus to prove the Subject Reduction Theorem (see [101]). However, interestingly this is also needed in our encoding that we present in Chapter 6. Rule (Tπ-If) is standard, except for the combination on typing contexts. Note that both branches of the conditional are typed in the same typing context, since only one of the branches will be chosen. Rules (Tπ-Inp) and (Tπ-Out) state that the input and output processes are well-typed if x is a linear channel used in input and output, respectively and the carried types are compatible to the types of $\tilde{y}$ and $\tilde{v}$. Note that $\widetilde{\Gamma_2}$ is the combination of all the typing contexts used to type $\tilde{v}$. Rule (Tπ-LVal) states that the variant value l_j_v is of type variant $\langle l_i\_T_i\rangle_{i\in I}$ if v is of type T_j and j is in I. Rule (Tπ-Case) states that process $\mathbf{case}\ v\ \mathbf{of}\ \{l_i\_x_i \triangleright P_i\}_{i\in I}$ is well-typed if the value v has compatible variant type and every process P_i is well-typed assuming x_i has type T_i. Notice that the **case** process, in the same way as for the conditional one, uses only one typing context, namely Γ_2 to type its branches. Again, this does not violate linearity, since only one of the branches is going to be executed.

4.5 Main Results

In this section we recall the main result for the linear π-calculus. We start with the definition of closed typing context.

Definition 4.5.1 (Closed Typing Context)**.** *A typing context* Γ *is* closed *if* $\forall x \in \mathsf{dom}(\Gamma)$*, then* $\Gamma(x) \neq \ell_\sharp\ [\widetilde{T}]$.

As usual, in order to prove the Type Soundness, one needs to prove first the Subject Reduction (or Type Preservation) and the Type Safety as stated in [97]. We start with the Subject Reduction Theorem.

$$\frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash x : T}\ (\mathrm{T}\pi\text{-}\textsc{Var}) \qquad \frac{\mathsf{un}(\Gamma) \quad v = \mathtt{true} \,/\, \mathtt{false}}{\Gamma \vdash v : \mathtt{Bool}}\ (\mathrm{T}\pi\text{-}\textsc{Val})$$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}\ (\mathrm{T}\pi\text{-}\textsc{Inact}) \qquad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash P \mid Q}\ (\mathrm{T}\pi\text{-}\textsc{Par})$$

$$\frac{\Gamma, x : \ell_\alpha\, [\widetilde{T}] \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}\ (\mathrm{T}\pi\text{-}\textsc{Res1}) \qquad \frac{\Gamma_1 \vdash v : \mathtt{Bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}\ (\mathrm{T}\pi\text{-}\textsc{If})$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}\ (\mathrm{T}\pi\text{-}\textsc{Res2}) \qquad \frac{\Gamma_1 \vdash x : \ell_{\mathsf{i}}\, [\widetilde{T}] \qquad \Gamma_2, \tilde{y} : \widetilde{T} \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x?(\tilde{y}).P}\ (\mathrm{T}\pi\text{-}\textsc{Inp})$$

$$\frac{\Gamma_1 \vdash x : \ell_{\mathsf{o}}\, [\widetilde{T}] \qquad \widetilde{\Gamma_2} \vdash \tilde{v} : \widetilde{T} \qquad \Gamma_3 \vdash P}{\Gamma_1 \uplus \widetilde{\Gamma_2} \uplus \Gamma_3 \vdash x!\langle \tilde{v} \rangle.P}\ (\mathrm{T}\pi\text{-}\textsc{Out})$$

$$\frac{\Gamma \vdash v : T_j \quad j \in I}{\Gamma \vdash l_j\_v : \langle l_i\_T_i \rangle_{i \in I}}\ (\mathrm{T}\pi\text{-}\textsc{LVal})$$

$$\frac{\Gamma_1 \vdash v : \langle l_i\_T_i \rangle_{i \in I} \qquad \Gamma_2, x_i : T_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{case}\ v\ \mathbf{of}\ \{l_i\_x_i \triangleright P_i\}_{i \in I}}\ (\mathrm{T}\pi\text{-}\textsc{Case})$$

Figure 4.7: Typing rules for the standard π-calculus

Theorem 4.5.2 (Subject Reduction for Linear π)**.** *Let* $\Gamma \vdash P$ *with* Γ *closed and* $P \to P'$*, then* $\Gamma \vdash P'$.

By analysing and combining the definition of closed typing context with the statement of the subject reduction property for linear π-types, we notice that since the typing context has no linear channel owning both capabilities, condition $\neq \ell_\sharp\, [\widetilde{T}]$, this means that, P reduces to P' either by a **case** normalisation or by a conditional reduction or in case a communication occurs, it is a communication on a restricted channel owning both capabilities of input and output.

Another important property is the following.

Lemma 4.5.3 (Type Preservation under $\equiv$ for Linear π)**.** *Let* $\Gamma \vdash P$ *and* $P \equiv P'$*, then* $\Gamma \vdash P'$.

In the following we give the definition of *well-formed* processes, which is

also present in the π- calculus with session types. The notion of well-formed processes is in opposition to that of *ill-formed* processes. The ill-formed processes fall in three different categories: i) conditional processes whose guard is neither `true` nor `false`, like **if** x **then** P **else** Q; ii) **case** processes whose guard is not a variant value, like **case** x **of** $\{l_i\_x_i \triangleright P_i\}_{i\in I}$; and iii) two threads, each owning the same variable but using it in the same capability, like $(\boldsymbol{\nu} x)(x?(z) \mid x?(z))$.

Definition 4.5.4 (Well-Formedness for Linear π)**.** *A process is* well-formed *if for any of its structural congruent processes of the form* $(\boldsymbol{\nu}\tilde{x})(P \mid Q)$ *the following hold.*

- *If P is of the form* **if** v **then** P_1 **else** P_2*, then v is either* `true` *or* `false`*.*
- *If P is of the form* **case** v **of** $\{l_i\_x_i \triangleright P_i\}_{i\in I}$*, then v is l_j_w for some variable w and for $j \in I$.*
- *If P is prefixed in x_i and Q is prefixed in x_i where $x_i \in \widetilde{x}$, then $P \mid Q$ is a redex.*

After the definition of well-formedness of a process, we are now ready to state the Type Safety property for the linear π-calculus.

Theorem 4.5.5 (Type Safety for Linear π)**.** *If $\vdash P$ then P is well-formed.*

The following theorem states that a well-typed closed process does not reduce to an ill-formed one.

Theorem 4.5.6 (Type Soundness for Linear π)**.** *If $\vdash P$ and $P \rightarrow^* Q$, then Q is well-formed.*

Notice that this is an alternative way of presenting the Soundness result in the standard typed π-calculus, informally stating that "well-typed programs do not go `wrong`" which is proved in [101].

CHAPTER 5

Background on Session Types

Session types are a formalism proposed as a foundation to model structured communication-based programming. They were originally introduced in [104] and later in [55] for a polyadic π-calculus.

Before formally introducing session types, we first present a typical distributed system example, namely the "Distributed Auction System" taken from [110].

Example 5.0.7. *Distributed Auction System*

There are three roles in this scenario: *sellers* that want to sell their items, *auctioneers* that are responsible for selling the items on behalf of the sellers and *bidders* that bid for the the items being auctioned. We describe now the protocols of the three roles. We will use meaningful names starting in capital letter to denote types for values, like `Item`, `Price` etc. We describe first the protocol for sellers. The only operation that a seller performs towards an auctioneer is selling, by first sending to the auctioneer the kind of the item that he wants to sell and the price that he wants the item to be sold. Then, the seller waits a for a reply from the auctioneer, which in case the item is sold, sends to the seller the price otherwise if the item is not sold, terminates the communication. However, in both cases the communication terminates. Formally we have:

$$\text{Seller: } \oplus\{\textit{selling} : \texttt{!Item.!Price.}\&\{\textit{sold} : \texttt{?Price.end}, \textit{not} : \texttt{end}\}\}$$

As previously, ? and ! denote, input and output actions, respectively; whether,

& and $\oplus$ denote external and internal choices, receptively, which are branch and select. Names in italics *selling, sold, not* indicate the labels of the choices. `Item` is the type of the items, which abstractly can be a string or an identifier denoted by a number etc. `Price` is the type of the price and generally can be an integer.

We now show the protocol for the auctioneers. An auctioneer communicates with both sellers and bidders, so its session type is as follows:

$$\text{Auctioneer: } \&\{\textit{selling} : ?\texttt{Item}.?\texttt{Price}.\oplus\{\textit{sold} : !\texttt{Price}.\texttt{end},\ \textit{not} : \texttt{end}\},$$
$$\textit{register} : ?\texttt{Id}.!\texttt{Item}.!\texttt{Price}.?\texttt{Bid}.\texttt{end}\}$$

The auctioneer offers a choice to the seller by the *selling* label: it receives from the seller the kind of item to be sold and the price and then, if the auctioneer manages to sell the item, he sends back to the seller the price to which the item was sold, if not, the communication ends. We can easily see the duality between the type of the seller and the *selling* branch of the auctioneer's session type.

$$\oplus\{\textit{selling} :\ !\texttt{Item}.!\texttt{Price}.\&\{\textit{sold} : ?\texttt{Price}.\texttt{end},\ \textit{not} : \texttt{end}\}\ldots\}$$
$$\&\{\textit{selling} : ?\texttt{Item}.?\texttt{Price}.\oplus\{\textit{sold} : !\texttt{Price}.\texttt{end},\ \textit{not} : \texttt{end}\}\}$$

The auctioneer offers a choice to the bidder by the *register* branch. `Id` is the type of the identity of the bidder, which abstractly can be an identity string or an identity number. `Bid` is the type of price that the bidder can offer for the item being auctioned. The *register* branch will be clearer once we describe the bidders protocol. Formally we have:

$$\text{Bidder: } \oplus\{\textit{register} : !\texttt{Id}.?\texttt{Item}.?\texttt{Price}.!\texttt{Bid}.\texttt{end}\}$$

This means that a bidder selects the *register* branch, which is the only branch available in its internal choice operator, and sends to the auctioneer his identity, which abstractly can be a string or a number etc. Then he receives from the auctioneer the item being auctioned and its price. Before terminating the communication, the bidder sends to the auctioneer his bid. Again, there is duality between the *register* branch of the auctioneer's session type and the type of the bidder.

$$\&\{\ldots,\ \textit{register} : ?\texttt{Id}.!\texttt{Item}.!\texttt{Price}.?\texttt{Bid}.\texttt{end}\}$$
$$\oplus\{\textit{register} : !\texttt{Id}.?\texttt{Item}.?\texttt{Price}.!\texttt{Bid}.\texttt{end}\}$$

So, summing it up we have the following situation:

$$\begin{aligned} \text{Auctioneer: } & \&\{\mathit{selling} : \ldots,\ \mathit{register} : \ldots\} \\ \text{Seller: } & \oplus\{\mathit{selling} : \ldots\} \\ \text{Bidder: } & \oplus\{\mathit{register} : \ldots\} \end{aligned}$$

Notice that, the above session types are not dual with each other, because the auctioneer's session type has one branch more than the seller's and the bidder's session type. However, by using subtyping, which we will introduce in Chapter 7, one can safely extend the types for seller and bidder to also include the missing branch, by thus establishing duality.

5.1 Syntax

The syntax of the π-calculus with sessions is given below:

$P, Q ::=$	$x!\langle v\rangle.P$	output
	$x?(y).P$	input
	$x \triangleleft l_j.P$	selection
	$x \triangleright \{l_i : P_i\}_{i\in I}$	branching
	if v **then** P **else** Q	conditional
	$P \mid Q$	composition
	$\mathbf{0}$	inaction
	$(\boldsymbol{\nu} xy)P$	session restriction
$v ::=$	x	variable
	`true` | `false`	boolean values

Figure 5.1: Syntax of the π-calculus with sessions

Let P, Q range over processes, x, y over variables, v over values, i.e., variables and ground values (integers, booleans, strings) and l over labels. For simplicity, we include in the present syntax only the boolean values, `true` and `false`. However, other ground values can be added to the above syntax and often in examples we will use them. The output process $x!\langle v\rangle.P$ sends a value v on channel x and proceeds as process P; the input process $x?(y).P$ receives a value on channel x, stores it in variable y and proceeds as P. The process $x \triangleleft l_j.P$ selects label l_j on channel x and proceeds as P. The branching process $x \triangleright \{l_i : P_i\}_{i\in I}$ offers a range of labelled alternatives on channel x, followed by their respective process continuations. The

branching and selection are called the choice processes, the external and the eternal one, respectively. The order of labelled processes is not important and the labels are all different. The process **if** v **then** P **else** Q is the standard conditional process. The process $P \mid Q$ is the parallel composition of processes P, Q. The process $\mathbf{0}$ is the terminated process. The process $(\nu xy)P$ is the scope restriction construct. This restriction is different from the standard one in the π-calculus. It is important to notice that (νxy) is not the same as $(\nu x)(\nu y)$. The latter restriction merely states that variables x and y are bound with scope P, but it does not state anything about a possible relation between x and y. Instead, (νxy) states that variables x and y are bound with scope P, and most importantly, are bound together, by representing two endpoints of the same (session) channel. When occurring under the same restriction, x and y are called *co-variables*. As we will see in the next section, communication occurs on co-variables. This special restriction, not only models the opposite endpoints of a channel but also the connection phase, which in other works [55, 104] is modelled by special primitives like `accept/request`. Some notational comments follow. We say that a process is *prefixed* in a variable x, if it is of the form $x!\langle v\rangle.P, x?(y).P, x \triangleleft l_j.P, or x \triangleright \{l_i : P_i\}_{i\in I}$. For simplicity, we will avoid triggering the terminated process, so we will omit $\mathbf{0}$ from any process in the examples to follow. The parenthesis in the terms represent bindings, in particular in $(\nu xy)P$ both variables x and y are bound with scope P; and in $x?(y).P$ variable y is bound with scope P. Hence, a variable can be *bound* or *free*, the latter holds when the variable does not occur under a restriction or as the object of an input process. We denote with $BV(P)$ the set of bound variables of process P and with $FV(P)$ we denote the set of free variables of process P. Hence, we use $Vars(P) = BV(P) \cup FV(P)$ to denote the set of variables in P. We will use *substitution* and *alpha-conversion* which are defined in the same way as for the standard π-calculus [101]. We use $P[x/y]$ as a notation for process P where every occurrence of the free variable y is substituted by variable x. As usual in the π-calculus, substitution is coupled with avoiding the unintended variable capture by the binders of the calculus. In order to achieve this, the alpha-conversion of variables is performed, which performs a renaming of bound variables in a process.

Definition 5.1.1 (Alpha-convertibility)**.** *The following give a procedure for substituting and renaming variables in a process.*

1. *If a variable x does not occur in a process P, then $P[x/y]$ is the process obtained by replacing every occurrence of y by x in P.*

2. *An* alpha conversion *of the bound variables in a process P is the replace-*

ment of a subterm $x?(y).Q$ *by* $x?(w).Q[w/y]$ *or the replacement of a subterm* $(\boldsymbol{\nu} xy)Q$ *by* $(\boldsymbol{\nu} wy)Q[w/x]$ *or* $(\boldsymbol{\nu} xw)Q[w/y]$ *where in both cases* w *does not occur in* Q*.*

3. *Processes* P *and* Q *are* alpha-convertible $P =^{\alpha} Q$ *if* Q *can be obtained from* P *by a finite number of changes in the bound variables.*

In this work, we adopt the same variable convention as in the original paper [109], namely that all variables in bindings in any mathematical context are pairwise distinct and distinct from the free variables.

5.2 Semantics

Before presenting the operational semantics, we introduce the notion of *structural congruence* $\equiv$ which is the smallest congruence relation on processes that satisfies the following axioms:

$$\begin{array}{rcll} P \mid Q & \equiv & Q \mid P & \\ (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\ P \mid \mathbf{0} & \equiv & \mathbf{0} \mid P & \\ (\boldsymbol{\nu} xy)\mathbf{0} & \equiv & \mathbf{0} & \\ (\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P & \equiv & (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P & \\ (\boldsymbol{\nu} xy)P \mid Q & \equiv & (\boldsymbol{\nu} xy)(P \mid Q) & (x, y \notin FV(Q)) \end{array}$$

Figure 5.2: Structural congruence for the π-calculus with sessions

The first three axioms state that the parallel composition of processes is commutative, associative and uses process $\mathbf{0}$ as the neutral element. The last three axioms state respectively that one can safely add or remove any restriction to the terminated process, the order of restrictions is not important and the last one called *scope extrusion* states that one can extend the scope of the restriction to another process in parallel. Notice that, as stated in [109], the side condition $x, y \notin FV(Q)$ is redundant, since in this calculus we adopt the variable convention that prohibits x, y to be free in Q since they occur bound in P. However, for more clarity, we report the condition as part of the last axiom.

The semantics of the π-calculus with sessions is given in terms of the reduction relation $\rightarrow$, and is a binary relation over processes, defined by the rules in Fig. 5.3. We denote with $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$.

$$\text{(R-Com)} \quad (\nu xy)(x!\langle v\rangle.P \mid y?(z).Q \mid R) \to (\nu xy)(P \mid Q[v/z] \mid R)$$

$$\text{(R-Sel)} \quad (\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I} \mid R) \to (\nu xy)(P \mid P_j \mid R) \quad j \in I$$

$$\text{(R-IfT)} \quad \textbf{if } \texttt{true} \textbf{ then } P \textbf{ else } Q \to P$$

$$\text{(R-IfF)} \quad \textbf{if } \texttt{false} \textbf{ then } P \textbf{ else } Q \to Q$$

$$\text{(R-Res)} \quad \frac{P \to Q}{(\nu xy)P \to (\nu xy)Q}$$

$$\text{(R-Par)} \quad \frac{P \to P'}{P \mid Q \to P' \mid Q}$$

$$\text{(R-Struct)} \quad \frac{P \equiv P',\ P' \to Q',\ Q' \equiv Q}{P \to Q}$$

Figure 5.3: Semantics of the π-calculus with sessions

We call *redexes* processes of the form $(\nu xy)(x!\langle v\rangle.P \mid y?(z).Q)$ or of the form $(\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I})$, for $j \in I$. Rule (R-Com) is the rule for communication: the process on the left sends a value v on x, while the process on the right receives the value on y and substitutes the placeholder z with it. A key difference with the standard π-calculus is that the subject of the output (x) and the subject of the input (y) are two co-variables, created and bound together by the (ν) construct. A consequence of this is that communication happens only in bound variables. After the communication the restriction still persists in order to enable further possible communications. This is another difference with respect to the standard π-calculus. Process R collects other usages of variables x and y. Rule (R-Sel) is similar: the communicating processes have prefixes that are co-variables according to the nearest restriction term. The selecting process continues as P and the branching process with the continuation P_j of the selected label. Again, notice that communication happens only on bound variables and the restriction persists after reduction in order to enable further communications. Process R collects other usages of variables x and y. Rules (R-IfT) and (R-IfF) state that the conditional process **if** v **then** P **else** Q reduces either to P or to Q depending on whether the value v is `true` or `false`, respectively. Rules (R-Res) and (R-Par) state that communication can happen under restriction and parallel composition, respectively. Rule (R-Struct) is the structural rule, previously introduced.

5.3 Session Types

Before presenting the session typing discipline for the π-calculus with sessions, we define the syntax of types, given in the following:

q ::=	$\mathsf{lin} \mid \mathsf{un}$	qualifiers
p ::=	$!T.U$	send
	$?T.U$	receive
	$\oplus\{l_i : T_i\}_{i \in I}$	select
	$\&\{l_i : T_i\}_{i \in I}$	branch
T ::=	$q\ p$	qualified pretype
	$\mathtt{end}$	termination
	$\mathtt{Bool}$	boolean type

Figure 5.4: Syntax of session types

Let q range over type qualifiers, p over pretypes, $q\ p$ over qualified pretypes, and T, U over types. A type can be `Bool`, the type of boolean values, `end`, the type of the terminated channel where no communication can take place further and $q\ p$, the qualified pretype. A pretype can be $!T.U$ or $?T.U$ which respectively, is the types of sending or receiving a value of type T with continuation of type U. Select $\oplus\{l_i : T_i\}_{i \in I}$ and branch $\&\{l_i : T_i\}_{i \in I}$ are sets of labelled types indicating, respectively, internal and external choice. The labels are all different and the order of the labelled types does not matter. Qualifiers are lin (for linear) or un (for unrestricted) and have the following meaning. Linear qualified pretypes describe channels whose pretype is executed *exactly* once, or said differently describe channels that are used exactly once in *one thread*, the latter being any process not including parallel composition. On the contrary, the unrestricted qualifier is used for channels that can be used an unlimited number of times in parallel. In the rest of this dissertation, we refer to types T whose qualifier is lin as *session types*. Instead, we refer to the unrestricted ones as shared channel types. In the rest of the work, we implicitly assume that the qualifier lin is used for every pretype unless it is explicitly stated otherwise. The following predicates, intuitively state when a type is linear or unrestricted.

$\mathsf{lin}(T)$	if $T = \mathsf{lin}\ p$
$\mathsf{un}(T)$	otherwise

A key notion in session types is *duality*. Type duality is standard, as in [55, 109],

and is defined in Fig 5.5. Qualifiers do not influence duality of types.

$$\begin{aligned}
\overline{\mathsf{end}} &\stackrel{\mathtt{def}}{=} \mathsf{end} \\
\overline{q!T.U} &\stackrel{\mathtt{def}}{=} q?T.\overline{U} \\
\overline{q?T.U} &\stackrel{\mathtt{def}}{=} q!T.\overline{U} \\
\overline{q \oplus \{l_i : T_i\}_{i \in I}} &\stackrel{\mathtt{def}}{=} q\&\{l_i : \overline{T}_i\}_{i \in I} \\
\overline{q\&\{l_i : T_i\}_{i \in I}} &\stackrel{\mathtt{def}}{=} q \oplus \{l_i : \overline{T}_i\}_{i \in I}
\end{aligned}$$

Figure 5.5: Type duality for session types

The dual of the terminated channel type is itself. The dual of an input type is an output type and vice versa, and the dual of a branch is a select and vice versa. Duality is undefined outside the equations presented above, for example, duality is not defined on `Bool`. If we include other ground types to the syntax above, like `Int` or `String`, duality would not be defined on them either. This is standard in session types theory and the reason for this is that if $\overline{\texttt{Bool}} = \texttt{Bool}$, then as stated in [109], the following process would be typable.

$$(\nu xy)\ \textbf{if}\ x\ \textbf{then 0 else 0}$$

Trivially, we do not want this to be the case. To conclude, duality satisfies the idempotence property, namely $\overline{\overline{T}} = T$.

5.4 Session Typing

The syntax of typing contexts is defined as follows:

$$\Gamma \quad ::= \quad \emptyset \quad | \quad \Gamma,\ x : T$$

As usual, we consider the typing context Γ to be a function from variables to types. Therefore, we write Γ, Γ' only when Γ and Γ' have disjoint domains. Typing rules make use of *context split* and *context update* defined in Fig. 5.6. These operations on typing contexts are used to deal with linearity of types. The context split operator $\circ$ adds a linear type $\mathsf{lin}\ p$ to either Γ_1 or Γ_2, when $\Gamma_1 \circ \Gamma_2$ is defined. When $\mathsf{lin}\ p$ is added to Γ_1 it is not present in Γ_2 and vice versa, when it is added to Γ_2 it is not present in Γ_1, since it is not in $\Gamma = \Gamma_1 \circ \Gamma_2$. In case $\mathsf{un}(T)$, then it is possible to add this type to both Γ_1 and Γ_2 and the context split operation is still

Context split $\Gamma = \Gamma \circ \Gamma$

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

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

Context update $\Gamma + x : T = \Gamma$

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

Figure 5.6: Context split and context update

defined on the typing contexts extended. The context update operator + is used to update the type of a variable with the continuation type in order to enable typing after an input (or branch) or an output (or select) operation has occurred. When the typing context Γ is updated with a variable having linear type –first rule– then the variable is not present in $\mathsf{dom}(\Gamma)$; otherwise, if the variable is of unrestricted type – second rule– then the typing context is updated only if the type of the variable is the same, namely $\mathsf{un}(T)$. We extend the lin and un predicates to typing contexts in the intuitive way, namely:

$$\mathsf{lin}(\Gamma) \text{ if } \exists (x : T) \in \Gamma \text{ such that } \mathsf{lin}(T)$$
$$\mathsf{un}(\Gamma) \text{ otherwise}$$

The type system for session processes satisfies two invariants. First, linear channels occur in exactly one thread, and second, co-variables have dual types. The first invariant is guaranteed by context split operation on typing contexts, and the second one is guaranteed by the typing rule for restriction. The type system avoids communication errors such as type mismatches and race conditions.

Typing judgements for values have the form $\Gamma \vdash v : T$, stating that a value v has type T in the typing context Γ, and typing judgements for processes have the form $\Gamma \vdash P$, stating that a process P is well-typed in the typing context Γ. The typing rules are given in Fig. 5.7.

$$\frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash x : T}\ (\text{T-Var}) \qquad \frac{\mathsf{un}(\Gamma) \quad v = \texttt{true} \,/\, \texttt{false}}{\Gamma \vdash v : \texttt{Bool}}\ (\text{T-Val})$$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}\ (\text{T-Inact}) \qquad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q}\ (\text{T-Par})$$

$$\frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)P}\ (\text{T-Res}) \qquad \frac{\Gamma_1 \vdash v : \texttt{Bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q}\ (\text{T-If})$$

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

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

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

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

Figure 5.7: Typing rules for the π-calculus with sessions

Rule (T-Var) states that a variable x is of type T, if this is the type assumed in the typing context. Rule (T-Val) states that a value v, being either `true` or `false`, is of type `Bool`. Rule (T-Inact) states that the terminated process $\mathbf{0}$ is always well-typed. Notice that in all the previous rules, the typing context Γ is an unrestricted one. The reason for $\mathsf{un}(\Gamma)$ is because every time we have a linearly qualified variable, that variable has to be used, which is not the case is these rules. Rule (T-Par) types the parallel composition of two processes, using the split operator for typing contexts $\circ$ which ensures that each linearly-typed channel x, is used linearly, i.e., in $P \mid Q$, x occurs either in P or in Q but never in both. However, this constraint is not required in case of unrestricted variables, which by context split definition can be on both Γ_1 and Γ_2. Rule (T-Res) states that $(\boldsymbol{\nu} xy)P$ is well-typed if P is well-typed and the co-variables have dual types, namely T and $\overline{T}$. Rule (T-If) states that the conditional statement is well-typed if its guard is typed by

a boolean type and the branches are well-typed under the same typing context. Γ_2 types both P and Q because only one of the branches is going to be executed. Rules (T-In) and (T-Out) type, respectively, the receiving and the sending of a value; these rules deal with both linear and unrestricted types. In (T-In) the typing context is split in two parts, Γ_1 and Γ_2, respectively. Γ_1 checks x is of type $q?T.U$, whether Γ_2 augmented with $y : T$ states the well-typedness of P. In addition, Γ_2 is updated by $x : U$ which is the type of the continuation of the communication. Notice that, by the definition of context update, if variable x is linearly qualified, then it is not in $\mathsf{dom}(\Gamma_2)$, otherwise, if it is unrestricted then the update is defined only if $U = q?T.U$. Rule (T-Out) splits the typing context in three parts, Γ_1, Γ_2 and Γ_3, respectively. Γ_1 checks x is of type $q!T.U$, Γ_2 checks the value to be sent v is of correct type T, and Γ_3 updated with the continuation type U checks the well-typedness of P. As in the previous rule, in case $q = \mathsf{un}$ the update operation is defined only if $U = q!T.U$. Rule (T-Brch) types an external choice on channel x, checking that each branch continuation P_i follows the respective type continuation in the type of x. Dually, rule (T-Sel) types an internal choice communicated on channel x, checking that the chosen label is among the ones offered by the receiver and that the continuation proceeds as expected by the type of x. In both rules, the typing context is split in $\Gamma_1 \circ \Gamma_2$. Γ_1 types the variable x by $q\&\{l_i : T_i\}_{i\in I}$ and $q \oplus \{l_i : T_i\}_{i\in I}$, respectively. In (T-Brch), every P_i process for $i \in I$ is well-typed in Γ_2 updated with x having type T_i. Since only one of the processes offered in the branching is going to be chosen, one can safely use only Γ_2 to typecheck them all. In (T-Sel), however, only the process P corresponding to the selected label l_j is typechecked. And again, the typing context Γ_2 is updated by the continuation type T_j that variable x has in P. The update of Γ_2 in case $q = \mathsf{un}$ is defined only if (T-Brch) $T_i = q\&\{l_i : T_i\}_{i\in I}$ and (T-Sel) $T_j = q \oplus \{l_i : T_i\}_{i\in I}$, respectively.

However, all the four equations reported above, for the input rules, (T-Inp) and (T-Brch) and for the output rules, (T-Out) and (T-Sel), in case variable x has an unrestricted type, are not solvable by only using the syntax of types presented so far. For example, consider the process

$$x!\langle \texttt{true} \rangle \mid x!\langle \texttt{false} \rangle$$

Since variable x is used in two threads in parallel, it should have an unrestricted type, in particular $x : \mathsf{unBool}.T$. Then by rule (T-Out) we have $x : \mathsf{unBool}.T + x : T$ which obviously is not satisfied by any type produced by the syntax of types presented in Section 5.3. This means that the only processes typable are the ones

that use only linear channels. However, it will be possible to typecheck the process previously written by introducing the recursive types, as we will see in Chapter 10.

5.5 Main Results

In this section we present the main properties satisfied by the type system presented in Fig. 5.7 in the previous section. The following lemmas and theorems are proven in [109]. We recall their statements and give a sketch of their proofs.

The following two lemmas are on the weakening and strengthening of typing contexts in session types.

Weakening allows us to introduce new unrestricted channels in a typing context. Notice that this holds only for unrestricted channels, for linear ones it would not be sound, since when a linear channel is in a typing context, this means that it is used in the process typed with that typing context. The weakening lemma is useful when we need to relax the typing assumptions for a process and include new typing assumptions of variables not free in the process. The statement of the lemma is as follows.

Lemma 5.5.1 (Unrestricted Weakening in Sessions)**.** *If* $\Gamma \vdash P$ *and* $\mathsf{un}(T)$ *then* $\Gamma, x : T \vdash P$.

Strengthening is somehow the opposite operation of weakening, since it allows us to remove unrestricted channels from the typing context that are not free in the process being typed. This operation is mostly used after a context split is performed. The statement of the lemma is as follows.

Lemma 5.5.2 (Strengthening in Sessions)**.** *Let* $\Gamma \vdash P$ *and* $x \notin FV(P)$ *then*

- $x : \mathsf{lin} p \notin \Gamma$
- $\Gamma = \Gamma', x : T$ *then,* $\Gamma' \vdash P$.

The substitution lemma that follows is important in proving the main results that we give at the end of the section. Notice that the lemma is not applicable in case $x = v$ and $\mathsf{un}(T)$ since there exists no Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$ where $x : T \in \Gamma_1$ but $x : U \notin \Gamma_2$ for all type U.

Lemma 5.5.3 (Substitution Lemma for Sessions)**.** *If* $\Gamma_1 \vdash v : T$ *and* $\Gamma_2, x : T \vdash P$ *and* $\Gamma = \Gamma_1 \circ \Gamma_2$ *then* $\Gamma \vdash P[v/x]$.

Another important property is the following one.

Lemma 5.5.4 (Type Preservation under ≡ for Sessions)**.** *Let* $\Gamma \vdash P$ *and* $P \equiv P'$*, then* $\Gamma \vdash P'$*.*

Before giving the Type Safety and the Subject Reduction properties, we first give the definitions of *well-formed* and *ill-formed* processes. The ill-formed processes fall in three different categories: i) conditional processes whose guard is neither `true` nor `false`, like **if** x **then** P **else** Q; ii) two threads using a variable in parallel with different actions like $(x!\langle \texttt{true} \rangle \mid x?(z))$; and iii) two threads, each owning a co-variable but using them by not respecting duality, like $(\nu xy)(x?(z) \mid y \triangleleft l_j.P)$. In order to avoid process as the previous ones, [109] defines the notion of well-formed processes, which we report in the following.

Definition 5.5.5 (Well-Formedness for Sessions)**.** *A process is* well-formed *if for any of its structural congruent processes of the form* $(\nu \widetilde{xy})(P \mid Q \mid R)$ *the following hold.*

- *If P is of the form* **if** v **then** P_1 **else** P_2*, then v is either* `true` *or* `false`*.*
- *If P and Q are prefixed at the same variable, then the variable performs the same action (input or output, branching or selection).*
- *If P is prefixed in* x_i *and Q is prefixed in* y_i *where* $x_i y_i \in \widetilde{xy}$*, then* $P \mid Q$ *is a redex.*

It is important to notice that well-typedness of a process does not imply that the process is well-formed. Consider **if** x **then** P **else** Q and $x : \texttt{Bool} \vdash$ **if** x **then** P **else** Q. This process is not well-formed since x is not a boolean value. However, this is no longer true when the process is closed, namely it is well-typed in an empty typing context. The following theorem holds and is proven in [109].

Theorem 5.5.6 (Type Safety for Sessions)**.** *If* $\vdash P$ *then P is well-formed.*

Another very important result is the Type Preservation or Subject Reduction property, given by the following theorem.

Theorem 5.5.7 (Subject Reduction for Sessions)**.** *If* $\Gamma \vdash P$ *and* $P \to Q$*, then* $\Gamma \vdash Q$*.*

Notice that by the operational semantics rules, since $P \to Q$ this means that either a conditional reduction occurs, rules (R-IfT) and (R-IfF), or a communication occurs, (R-Com) and (R-Sel); – the other cases are a generalisation of the former ones. – Since communication occurs only on co-variables, namely restricted variables, this implies that the channel in which the communication occurs is not in the typing context Γ.

We are ready now to present the main result of the session type system The following theorem states that a well-typed closed process does not reduce to an ill-formed one.

Theorem 5.5.8 (Main Result)**.** *If $\vdash P$ and $P \to^* Q$, then Q is well-formed.*

CHAPTER 6

Session Types Revisited

In this chapter we introduce the encoding of session types into linear channel types and variant types in the standard typed π-calculus and of session processes into π-calculus processes.

As previously mentioned, session types guarantee that only the communicating parties know the corresponding endpoints of the session channel, thus providing privacy. Moreover, the opposite endpoints should have dual types, thus providing communication safety. In addition, they guarantee that the structure of the session is the one expected, thus guaranteeing session fidelity. The interpretation of session types should take into account these fundamental issues. In order to guarantee privacy and safety of communication we adopt linear channels that are used *exactly once*. Privacy is ensured since the linear channel is used *at most once* and so it is known only to the interacting parties. Communication safety is ensured since the linear channel is used *at least once* and so the input/output actions are necessarily performed. Instead, session fidelity is guaranteed by simulating the structure of the session type by sending along with the value also the channel implementing the continuation of the communication.

$$
\begin{array}{rcll}
[\![\mathtt{end}]\!] & \stackrel{\text{def}}{=} & \emptyset\,[] & \text{(E-End)} \\
[\![!T.U]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{o}}\,[[\![T]\!], [\![\overline{U}]\!]] & \text{(E-Out)} \\
[\![?T.U]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{i}}\,[[\![T]\!], [\![U]\!]] & \text{(E-Inp)} \\
[\![\oplus\{l_i : T_i\}_{i\in I}]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{o}}\,[\langle l_i \_ [\![\overline{T_i}]\!]\rangle_{i\in I}] & \text{(E-Select)} \\
[\![\&\{l_i : T_i\}_{i\in I}]\!] & \stackrel{\text{def}}{=} & \ell_{\mathsf{i}}\,[\langle l_i \_ [\![T_i]\!]\rangle_{i\in I}] & \text{(E-Branch)}
\end{array}
$$

Figure 6.1: Encoding of session types

6.1 Types Encoding

Recall that the syntax of types presented in Section 5.3 uses the notion of qualifiers: lin and un. The purpose of the qualifiers is to give semantics to pretypes and distinguish between the linear and unrestricted ones. In particular, linear pretypes denote the standard session types, as are known in the literature, whereas the unrestricted ones can be roughly associated to the standard π-channels used multiple times in different threads, with the additional feature of being structured and describing a communication. In this chapter, by following [31], we present the encoding of session types presented in Section 5.3 namely, we only encode the *linear* pretypes into linear π-types augmented with variant type. – We will define the encoding of the unrestricted pretypes in Chapter 10, when dealing with recursion and recursive types. –

Formally, we encode the types produced by the following grammar:

$$T ::= \mathtt{Bool} \mid \mathtt{end} \mid \mathsf{lin}p$$

where the encoding of a boolean type, and in general, the encoding of any other ground type that is added to the syntax of types, like `Int`, `String`, `Unit` ..., is the identity function, since the same type constructs can be added to the syntax of types in the standard π-calculus, namely:

$$
\begin{array}{rcll}
[\![\mathtt{Bool}]\!] & \stackrel{\text{def}}{=} & \mathtt{Bool} & \text{(E-Bool)} \\
[\![\mathtt{Int}]\!] & \stackrel{\text{def}}{=} & \mathtt{Int} & \text{(E-Int)} \\
[\![\mathtt{String}]\!] & \stackrel{\text{def}}{=} & \mathtt{String} & \text{(E-String)} \\
[\![\mathtt{Unit}]\!] & \stackrel{\text{def}}{=} & \mathtt{Unit} & \text{(E-Unit)}
\end{array}
$$

The encoding of session types into the standard π-types is presented in Fig. 6.1.

Equation (E-End) states that the encoding of the terminated communication channel is $\emptyset$ [], namely the channel with no capability which cannot be used for communication. Equation (E-Out) states that the encoding of $!T.U$ is a linear type used in output to carry a pair of values of type the encoding of T and of type the encoding of the dual of U. The reason for duality of U is that the sender sends to its peer the channel for the continuation of the communication, and hence the sender sends a channel being typed according to how the peer is going to use it. Equation (E-Inp) states that the session type $?T.U$ is encoded as the linear input channel type carrying a pair of values of type the encoding of T and of the encoding of continuation type U. Equations (E-Select) and (E-Branch) define the encoding of select and branch, respectively. Select and branch types are generalisations of output and input types, respectively. Consequently, they are interpreted as linear output and linear input channels carrying variant types with the same labels $l_1 \dots l_n$ and types the encodings of $\overline{T_1} \dots \overline{T_n}$ and $T_1 \dots T_n$, respectively. Again, the reason for duality is the same as for the output type.

Let us now illustrate the encoding of types with a simple example. Let $x : T$ and $y : \overline{T}$ where

$$T = ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

and

$$\overline{T} = !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

A process well-typed in $x : T$ uses channel x to receive in sequence two integer numbers and then to output a boolean value. Instead, a process well-typed in $y : \overline{T}$ uses channel y to perform exactly the opposite actions: it outputs in sequence two integer numbers and waits for a boolean value in return.

The encoding of these types is as follows:

$$[\![T]\!] = \ell_{\mathsf{i}}\,[\texttt{Int}, \ell_{\mathsf{i}}\,[\texttt{Int}, \ell_{\mathsf{o}}\,[\texttt{Bool}, \emptyset[]]]]$$

and

$$[\![\overline{T}]\!] = \ell_{\mathsf{o}}\,[\texttt{Int}, \ell_{\mathsf{i}}\,[\texttt{Int}, \ell_{\mathsf{o}}\,[\texttt{Bool}, \emptyset[]]]]$$

This simple example shows that the duality of session types boils down to duality of capabilities of linear channel types, namely the encodings above differ only in the outermost level, that corresponds to having ℓ_i or ℓ_o channel types. The π- channels having these types carry exactly the same messages. This happens because duality is incorporated in the output typing, where the receiver's point of view of the output type is considered, which is therefore dual with respect to that

$$
\begin{array}{rcll}
[\![x]\!]_f & \stackrel{\text{def}}{=} & f_x & \text{(E-Variable)} \\
[\![\texttt{true}]\!]_f & \stackrel{\text{def}}{=} & \texttt{true} & \text{(E-True)} \\
[\![\texttt{false}]\!]_f & \stackrel{\text{def}}{=} & \texttt{false} & \text{(E-False)} \\
[\![\mathbf{0}]\!]_f & \stackrel{\text{def}}{=} & \mathbf{0} & \text{(E-Inaction)} \\
[\![x!\langle v\rangle.P]\!]_f & \stackrel{\text{def}}{=} & (\nu c) f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Output)} \\
[\![x?(y).P]\!]_f & \stackrel{\text{def}}{=} & f_x?(y, c).[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Input)} \\
[\![x \triangleleft l_j.P]\!]_f & \stackrel{\text{def}}{=} & (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Selection)} \\
[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f & \stackrel{\text{def}}{=} & f_x?(y).\ \mathbf{case}\ y\ \mathbf{of}\ \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I} & \text{(E-Branching)} \\
[\![\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!]_f & \stackrel{\text{def}}{=} & \mathbf{if}\ f_v\ \mathbf{then}\ [\![P]\!]_f\ \mathbf{else}\ [\![Q]\!]_f & \text{(E-Conditional)} \\
[\![P \mid Q]\!]_f & \stackrel{\text{def}}{=} & [\![P]\!]_f \mid [\![Q]\!]_f & \text{(E-Composition)} \\
[\![(\nu xy)P]\!]_f & \stackrel{\text{def}}{=} & (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}} & \text{(E-Restriction)}
\end{array}
$$

Figure 6.2: Encoding of session terms

of the sender. So, the encoding simplifies the structure of the dual session types and consequently the typechecking of a language term.

6.2 Terms Encoding

In this section we present the encoding of terms of the π-calculus with sessions into the standard π-calculus terms. The encoding of terms is different from the encoding of types as it is parametrised in a partial function f from variables to variables. The reason for using a function f is the following: since we are using linear channel types to encode session types, for the linearity to be guaranteed, once a channel is used it cannot be used again for transmission. However, to enable structured communication, and thus simulate session types, at every output action a new channel is created and is sent to the peer in order to use it for the continuation of the session. Finally, the function f is updated with the new variable created. Formally, the update of a function f is defined as follows:

$$
f, \{x \mapsto c\} \stackrel{\text{def}}{=} \begin{cases} f \cup \{x \mapsto c\} & \text{if } x \notin \mathsf{dom}(f) \\ (f \setminus \{x \mapsto f(x)\}) \cup \{x \mapsto c\} & \text{otherwise} \end{cases}
$$

The encoding of terms of the π- calculus with sessions is defined by the equations in Fig. 6.2. Equation (E-Variable) states that a variable x is encoded as the mapping of the function f. For simplicity, we denote with f_x the output of $f(x)$. Equations (E-True) and (E-False) state that the encoding of boolean values is the identity function. In particular, this holds for every ground value, like integers, strings etc. that one wants to add to both the π- calculus with and without sessions. Equation (E-Inaction) states that the terminated process is interpreted as the terminated process in the standard π- calculus. The encoding of the output process, equation (E-Output), is as follows: a new channel c is created and is sent along with the value v on channel x renamed by function f; the encoding of the continuation process P is parametrised in f updated by the association of x to c. Dually, the input process, equation (E-Input), receives on channel f_x a value that substitutes variable y and a fresh channel c that substitutes f_x in the continuation process. The selection and branching encodings, given by (E-Selection) and (E-Branching), are somehow generalisations of the output and input ones. The selection process $x \triangleleft l_j.P$ is encoded as the process that first creates a new channel c and then sends on f_x a variant value l_j_c, since l_j is the label selected and c is the channel created for the continuation of the session and proceeds as process P encoded in f updated. The branching process is the most complicated one: it receives on f_x a value, typically being a variant value. The value that substitutes y is the guard of the **case** process and according to its label one of the corresponding processes $[\![P_i]\!]_{f,\{x \mapsto c\}}$ $i \in I$ is chosen. The encoding of the conditional process, given by (E-Conditional) is given by the corresponding conditional in the standard π- calculus where the guard v and both branches P and Q are encoded in f. The encoding of the parallel composition of processes, given by (E-Composition) states that the encoding is an homomorphism, namely, is the composition of the encodings of the subprocesses. The encoding of the restriction is given by (E-Restriction). A new channel c is created, typically having linear connection type, and substitutes both co-variables x, y in the encoding of P.

Let us now illustrate the encoding of processes by a simple example. Consider the "equality test" problem. Given two processes, a server and a client, where the client sends to the server two integers, one after the other, and receives from the server a boolean value, being `true` if the integers are equal or `false` otherwise.

The processes are defined as follows:

$$\mathsf{server} \stackrel{\mathtt{def}}{=} x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$

$$\mathsf{client} \stackrel{\mathtt{def}}{=} y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

These processes communicate on a session channel by owning two opposite endpoints x and y, respectively. The system is given by

$$(\boldsymbol{\nu} xy)\ (\mathsf{server} \mid \mathsf{client})$$

The client process sends over channel y two integers, being 3 and 5, respectively, and waits for a boolean value in return which asserts the equality of the integers. On the other hand, the server process receives the two integers, which substitutes for the placeholders $nr1$ and $nr2$ and sends back to the client the boolean value corresponding to the result of testing $(nr1 == nr2)$, which in this case is `false`.

The encoding of the above system, by following (E-Restriction), is

$$[\![(\boldsymbol{\nu} xy)\ (\mathsf{server} \mid \mathsf{client})]\!]_f = (\boldsymbol{\nu} z)\ [\![(\mathsf{server} \mid \mathsf{client})]\!]_{f,\{x,y\mapsto z\}}$$

where the encodings of server and client processes are as follows:

$$[\![\mathsf{server}]\!]_{f,\{x,y\mapsto z\}} \stackrel{\mathtt{def}}{=} z?(nr1, c).c?(nr2, c').(\boldsymbol{\nu} c'')c!\langle nr1 == nr2, c''\rangle.\mathbf{0}$$

$$[\![\mathsf{client}]\!]_{f,\{x,y\mapsto z\}} \stackrel{\mathtt{def}}{=} (\boldsymbol{\nu} c)z!\langle 3, c\rangle.(\boldsymbol{\nu} c')c!\langle 5, c'\rangle.c'?(eq, c'').\mathbf{0}$$

In the encoding, at the very first step, function f is empty and x and y are mapped to a new name z; after that, at every output action a new channel c, c', c'' is created and sent to the communicating peer along with the value.

The above system is well-typed in a type environment that associates to the endpoints x, y the following session types:

$$x : ?\mathtt{Int}.?\mathtt{Int}.!\mathtt{Bool}.\mathtt{end}$$

$$y : !\mathtt{Int}.!\mathtt{Int}.?\mathtt{Int}.\mathtt{end}$$

which are already encoded in the previous section.

6.3 Properties of the Encoding

In this section we present some important theoretical results regarding the encoding. The properties we prove about our encoding follow the requirements in [51] about an encoding being a good means of language comparison. Recall that we are confining ourselves to the session types setting, where the only unrestricted types that we encode are the ground types, in particular `Bool` and the type of the terminated channel, `end`. Formally:

$$\mathsf{un}(T) \text{ if and only if } T = \texttt{Bool} \text{ or } T = \texttt{end}$$

This implies the following:

$$\mathsf{un}(\Gamma) \text{ if and only if } \forall (x : T) \in \Gamma \text{ implies } T = \texttt{Bool} \text{ or } T = \texttt{end}$$

In order to prove these results, the encoding is extended to typing contexts in the expected way. It is presented in Fig. 6.3.

$$\begin{aligned} [\![\emptyset]\!]_f &\stackrel{\text{def}}{=} \emptyset && \text{(E-Empty)} \\ [\![\Gamma, x : T]\!]_f &\stackrel{\text{def}}{=} [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] && \text{(E-Gamma)} \end{aligned}$$

Figure 6.3: Encoding of typing contexts

Notice that, the ',' operator on session typing contexts is interpreted as the '$\uplus$' operator on linear typing contexts. The reason is the following: the (dual) co-variables are interpreted as the same (linear) channel, which in order to be used for communication, must have connection capability. Hence, by using the $\uplus$ the dual capabilities of linear channels can be "summed-up" into the connection capability.

6.3.1 Auxiliary Results

In this section we give a few auxiliary lemmas that are used to prove properties about our encoding.

The following two lemmas give a relation between the context update operator '+' and the standard ',' operator in session typing contexts.

Lemma 6.3.1. *If* $\Gamma + x : T$ *is defined and* $x \notin \mathsf{dom}(\Gamma)$*, then also* $\Gamma, x : T$.

Proof. The result follows immediately by the definition of context update given in Fig. 5.6. □

Lemma 6.3.2. *If* $\Gamma, x : T$ *is defined, then also* $\Gamma + x : T$ *is defined.*

Proof. The result follows immediately by the definition of context update given in Fig. 5.6. □

In the same spirit, the following two lemmas give the relation between the combination operator $\uplus$ and the standard , operator in linear π-typing contexts.

Lemma 6.3.3. *If* $\Gamma \uplus x : T$ *is defined and* $x \notin \mathsf{dom}(\Gamma)$*, then also* $\Gamma, x : T$ *is defined.*

Proof. The result follows immediately by the definition of combination of typing contexts. □

Lemma 6.3.4. *If* $\Gamma, x : T$ *is defined, then also* $\Gamma \uplus x : T$ *is defined.*

Proof. By definition on ',' operator, we have that $x : T \notin \Gamma$. The result follows immediately by the definition of combination of typing contexts. □

The following two lemmas give a relation between the context split operator '$\circ$' in session typing contexts and the combination operator '$\uplus$' in linear π-typing contexts by using the encoding of typing contexts presented in Fig. 6.3.

Lemma 6.3.5 (Split to Combination)**.** *Let* $\Gamma_1, \ldots, \Gamma_n$ *be session typing contexts, then* $[\![\Gamma_1 \circ \ldots \circ \Gamma_n]\!]_f = [\![\Gamma_1]\!]_f \uplus \ldots \uplus [\![\Gamma_n]\!]_f$.

Proof. First suppose that every Γ_i is a linear session typing context, then by definition of context split, given in Fig. 5.6, this means that any linear type assumption occurs only in one typing context in $\Gamma_1, \ldots, \Gamma_n$, namely the $\circ$ operator boils down to the ',' operator. Then, the result follows immediately by equation (E-Gamma) 6.3. Suppose now there exist at least one Γ_j such that $\mathsf{un}(\Gamma_j)$. By the definition of context split, given in Fig. 5.6, this means that any unrestricted type assumption is split to every Γ_i present in the context split. By the encoding of types and hence of typing contexts, and by the combination of types in the π-calculus where $T \uplus T = T$ if $\mathsf{un}(T)$, we also have that the combination of typing contexts is well-defined and hence $[\![\Gamma_1]\!]_f \uplus \ldots \uplus [\![\Gamma_n]\!]_f$. □

Lemma 6.3.6 (Combination to Split)**.** *Let* $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \ldots \uplus \Gamma_n^\pi$*. For all* $i \in 1 \ldots n$*,* $\Gamma_i^\pi = [\![\Gamma_i]\!]_f$ *such that* $\Gamma = \Gamma_1 \circ \ldots \circ \Gamma_n$.

Proof. The result follows immediately by the encoding of typing contexts given in Fig. 6.3 and Definition 5.6 on context split $\circ$. □

The following lemma gives an important result that relates the encoding of dual session types to dual linear π-calculus channel types.

Lemma 6.3.7 (Encoding of dual types)**.** *If* $[\![T]\!] = \tau$ *then,* $[\![\overline{T}]\!] = \overline{\tau}$.

Proof. The proof is done by induction on the structure of session type T. We use the duality of session types defined in Fig. 5.5 and the duality of standard π-types defined in Fig. 4.6.

- $T = \mathsf{end}$
  By (E-End) we have $[\![\mathsf{end}]\!] = \emptyset[]$ and $\overline{T} = \mathsf{end}$. We conclude by the duality of $\emptyset[]$.

- $T = !T.U$
  By (E-Out) we have $[\![!T.U]\!] = \ell_{\mathsf{o}}\,[[\![T]\!], [\![\overline{U}]\!]]$. By duality of session types we have $\overline{!T.U} = ?T.\overline{U}$. By (T-In) we have $[\![?T.\overline{U}]\!] = \ell_{\mathsf{i}}\,[[\![T]\!], [\![\overline{U}]\!]]$. We conclude by the duality of π-types.

- $T = ?T.U$
  This case follows the same line as the previous one. By (E-In) we have $[\![?T.U]\!] = \ell_{\mathsf{i}}\,[[\![T]\!], [\![U]\!]]$. By duality of session types we have $\overline{?T.U} = !T.\overline{U}$. By (E-Out) we have $[\![!T.\overline{U}]\!] = \ell_{\mathsf{o}}\,[[\![T]\!], [\![\overline{\overline{U}}]\!]]$, which by the idempotence property of duality means $\ell_{\mathsf{o}}\,[[\![T]\!], [\![U]\!]]$. We conclude by the duality of π-types.

- $T = \oplus\{l_i : T_i\}_{i\in I}$
  By (E-Select) we have $[\![\oplus\{l_i : T_i\}_{i\in I}]\!] = \ell_{\mathsf{o}}\,[\langle l_i\_[\![\overline{T_i}]\!]\rangle_{i\in I}]$ By duality on session types we have $\overline{\oplus\{l_i : T_i\}_{i\in I}} = \&\{l_i : \overline{T}_i\}_{i\in I}$. By (E-Branch) we have $[\![\&\{l_i : \overline{T}_i\}_{i\in I}]\!] = \ell_{\mathsf{i}}\,[\langle l_i\_[\![\overline{T_i}]\!]\rangle_{i\in I}]$ We conclude by the duality of π-types.

- $T = \&\{l_i : T_i\}_{i\in I}$
  By (E-Branch) we have $[\![\&\{l_i : T_i\}_{i\in I}]\!] = \ell_{\mathsf{i}}\,[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$ By duality on session types we have $\overline{\&\{l_i : T_i\}_{i\in I}} = \oplus\{l_i : \overline{T}_i\}_{i\in I}$. By (E-Select) we have $[\![\oplus\{l_i : \overline{T}_i\}_{i\in I}]\!] = \ell_{\mathsf{o}}\,[\langle l_i\_[\![\overline{\overline{T_i}}]\!]\rangle_{i\in I}]$, which by the idempotence property of duality means $\ell_{\mathsf{o}}\,[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$. We conclude by the duality of π-types.

□

6.3.2 Typing Values by Encoding

The following two lemmas state the soundness and completeness of the encoding, in typing derivations for values. Namely, if a session value v has a session type T in a session typing context Γ, then the encoding of v has a type the encoding of T in the typing context being the encoding of Γ, and vice versa.

Lemma 6.3.8 (Soundness:Value Typing)**.** *If* $\Gamma \vdash v : T$*, then* $[\![\Gamma]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$.

Proof. The proof is done by induction on the derivation $\Gamma \vdash v : T$, by analysing the last rule applied.

- Case (T-Var):
$$\frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash x : T}$$
To prove $[\![\Gamma, x : T]\!]_f \vdash [\![x]\!]_f : [\![T]\!]$. By (E-Gamma) and (E-Variable) it means $[\![\Gamma]\!]_f \uplus f_x : [\![T]\!] \vdash f_x : [\![T]\!]$. By Lemma 6.3.3 and rule (Tπ- Var) we obtain the result.

- Case (T-Val):
$$\frac{\mathsf{un}(\Gamma) \quad v = \mathtt{true} \,/\, \mathtt{false}}{\Gamma \vdash v : \mathtt{Bool}}$$
To prove $[\![\Gamma]\!]_f \vdash [\![v]\!]_f : [\![\mathtt{Bool}]\!]$. By applying (E-True) or (E-False) depending on whether v is `true` or `false`, (E-Bool) and rule (Tπ- Val) we obtain the result.

□

Lemma 6.3.9 (Completeness:Value Typing)**.** *If* $[\![\Gamma]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$*, then* $\Gamma \vdash v : T$.

Proof. The proof is done by induction on the structure of the value v:

- Case $v = x$:
By (E-Variable) we have $[\![x]\!]_f = f_x$ and assume $[\![\Gamma]\!]_f \vdash f_x : [\![T]\!]$. By (T$\pi$-Var) this means that $(f_x : [\![T]\!]) \in [\![\Gamma]\!]_f$ and hence $[\![\Gamma]\!]_f = \Gamma_1^\pi, f_x : [\![T]\!]$ which by Lemma 6.3.4 and by (E-Gamma) means that $\Gamma = \Gamma_1, x : T$, where $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$. By (T$\pi$-Var) we have $\mathsf{un}([\![\Gamma']\!]_f)$. By the encoding of types also $\mathsf{un}(\Gamma')$ holds. By applying rule (T-Var) we obtain the result.

- Case $v =$ `true`:
By (E-True) and (E-Bool) we have $[\![\mathtt{true}]\!]_f = \mathtt{true}$ and assume $[\![\Gamma]\!]_f \vdash \mathtt{true} : \mathtt{Bool}$ and $\mathsf{un}([\![\Gamma]\!]_f)$. Then also $\mathsf{un}(\Gamma)$ holds. By applying rule (T-Val) we obtain the result.

- Case v = `false`:
  The proof is done in the same way as the previous case, by considering (E-False) instead of (E-True).

□

6.3.3 Typing Processes by Encoding

The following two theorems state the soundness and completeness of the encoding, presented in Section 6.2, in typing derivations for processes. Namely, if a session process P is well-typed in a session typing environment Γ, then the encoding of P is also well-typed in the encoding of Γ, and vice versa.

Recall that typing rules for the π- calculus with sessions, presented in Section 5.4 use qualified pretypes. Since we are encoding only session types, namely linear pretypes, in the remainder we will omit q from the typing rules. Moreover, these typing rules use the update operator +, which updates a typing environment with type assumption for the continuation of the communication. Basically, this operator is needed for the unrestricted types. Since the only unrestricted types we encode are `Bool` and `end`, but the only unrestricted type that can occur in typing the continuation of the communication is just `end`, then the '+' operator boils down to ',' operator by following the definition of context update given in Fig 5.6.

Theorem 6.3.10 (Soundness: Process Typing)**.** *If* $\Gamma \vdash P$ *then* $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$.

Proof. The proof is done by induction on the the derivation $\Gamma \vdash P$, by analysing the last typing rule applied.

- Case (T-Inact):
$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}\ \text{(T-Inact)}$$
  By applying rule (Tπ-Inact) and equation (E-Inaction) and letting f be any function on $\mathsf{dom}(\Gamma)$, we obtain the result.

- Case (T-Par):
$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q}\ \text{(T-Par)}$$
  By induction hypothesis $[\![\Gamma_1]\!]_f \vdash [\![P]\!]_f$ and $[\![\Gamma_2]\!]_f \vdash [\![Q]\!]_f$. To prove that $[\![\Gamma_1 \circ \Gamma_2]\!]_f \vdash [\![P \mid Q]\!]_f$. The result follows by applying rule (Tπ- Par), Lemma 6.3.5 and equation (E-Composition).

- Case (T-Res):

$$\frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu xy)P} \text{ (T-Res)}$$

To prove that $[\![\Gamma]\!]_f \vdash [\![(\nu xy)P]\!]_f$, which by equation (E-Restriction) means $[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$. We distinguish the following two cases:

  - Suppose $T \neq \mathsf{end}$, and hence $\overline{T} \neq \mathsf{end}$. By induction hypothesis $[\![\Gamma, x : T, y : \overline{T}]\!]_{f'} \vdash [\![P]\!]_{f'}$, for some function f' such that $\mathsf{dom}(f') = \mathsf{dom}(\Gamma) \cup \{x, y\}$ and let $f'(x) = f'(y) = c$ and let $f = f' - \{x \mapsto c, y \mapsto c\}$. By applying (E-Gamma), the typing judgement becomes $[\![\Gamma]\!]_f \uplus c : [\![T]\!] \uplus c : [\![\overline{T}]\!] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$, namely $[\![\Gamma]\!]_f \uplus c : [\![T]\!] \uplus [\![\overline{T}]\!] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By Lemma 6.3.7, $[\![T]\!] = \tau$ and $[\![\overline{T}]\!] = \overline{\tau}$. Since $T \neq \mathsf{end}$, $\overline{T} \neq \mathsf{end}$, we have that $[\![T]\!] = \ell_\alpha\ [W]$ and $[\![\overline{T}]\!] = \ell_{\overline{\alpha}}\ [W]$ and hence $c : \ell_\sharp\ [W]$, where W denote the (tuple of) carried type which is irrelevant. Since variable c owns both capabilities it means that $c \notin \mathsf{dom}([\![\Gamma]\!]_f)$, otherwise $\uplus$ would not have been defined. Hence, by Lemma 6.3.3, $[\![\Gamma]\!]_f, c : [\![T]\!] \uplus [\![\overline{T}]\!] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By applying rule (T$\pi$-Res1) we obtain $[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$ which concludes this case.
  - Suppose $T = \overline{T} = \mathsf{end}$. By induction hypothesis $[\![\Gamma, x : \mathsf{end}, y : \mathsf{end}]\!]_{f'} \vdash [\![P]\!]_{f'}$, for some function f' such that $\mathsf{dom}(f') = \mathsf{dom}(\Gamma) \cup \{x, y\}$ and let $f'(x) = f'(y) = c$ and let $f = f' - \{x \mapsto c, y \mapsto c\}$. By applying equation (E-Gamma), the typing judgement becomes $[\![\Gamma]\!]_f \uplus c : [\![\mathsf{end}]\!] \uplus c : [\![\mathsf{end}]\!] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$, namely $[\![\Gamma]\!]_f \uplus c : \emptyset[] \uplus c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$ which by the combination of unrestricted π-types means $[\![\Gamma]\!]_f \uplus c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. Notice that $c \notin \mathsf{dom}([\![\Gamma]\!]_f)$, otherwise $\uplus$ would not have been defined. Hence, by Lemma 6.3.3 we obtain $[\![\Gamma]\!]_f, c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. Notice that $c \notin [\![P]\!]_{f,\{x,y\mapsto c\}}$, since $x, y \notin FV(P)$ being x, y terminated channels. Then, by rule (Tπ-Res2) we obtain $[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$ which concludes this case.

- Case (T-If):

$$\frac{\Gamma_1 \vdash v : \mathtt{Bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q} \text{ (T-If)}$$

By induction hypothesis $[\![\Gamma_2]\!]_{f'} \vdash [\![P]\!]_{f'}$, and $[\![\Gamma_2]\!]_{f'} \vdash [\![Q]\!]_{f'}$ for some function f' such that $\mathsf{dom}(f') = \mathsf{dom}(\Gamma_2)$ and $[\![\Gamma_1]\!]_{f''} \vdash [\![v]\!]_{f''} : \mathtt{Bool}$, for some function f'' such that $\mathsf{dom}(f'') = \mathsf{dom}(\Gamma_1)$. To prove that $[\![\Gamma_1 \circ \Gamma_2]\!]_f \vdash \textbf{if } [\![v]\!]_f \textbf{ then } [\![P]\!]_f \textbf{ else } [\![Q]\!]_f$, for some function f such that

$\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Let $f = f' \cup f''$, then the result follows by equation (E-Conditional), Lemma 6.3.5 and rule (Tπ-If).

- Case (T-In):

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

To prove that $[\![\Gamma_1 \circ \Gamma_2]\!]_f \vdash [\![x?(y).P]\!]_f$ for some function f such that $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. By induction hypothesis $[\![\Gamma_1]\!]_{f'} \vdash f'_x : [\![?T.U]\!]$ which by (E-Inp) means $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_{\mathsf{i}}[[\![T]\!], [\![U]\!]]$. and $[\![\Gamma_2, x : U, y : T]\!]_{f''} \vdash [\![P]\!]_{f''}$, which by (E-Gamma) means $[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![U]\!] \uplus f''_y : [\![T]\!] \vdash [\![P]\!]_{f''}$. Let $f''(x) = c, f''(y) = y$ and let $f = f' \cup f'' - \{x \mapsto c, y \mapsto y\}$. Then the induction hypothesis become $[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathsf{i}}[[\![T]\!], [\![U]\!]]$ and $[\![\Gamma_2]\!]_f, y : [\![T]\!], c : [\![U]\!] \vdash [\![P]\!]_{f,\{x \mapsto c\}}$. By equation (E-Input), rule (Tπ-Inp) and Lemma 6.3.5 we obtain $[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash f_x?(y, c).[\![P]\!]_{f,\{x \mapsto c\}}$.

- Case (T-Out):

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

To prove that $[\![\Gamma_1 \circ \Gamma_2 \circ \Gamma_3]\!]_f \vdash [\![x!\langle v \rangle.P]\!]_f$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) \cup \mathsf{dom}(\Gamma_3)$. By (E-Output) and Lemma 6.3.5 this means that $[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f \vdash (\nu c) f_x!\langle v, c \rangle.[\![P]\!]_{f,\{x \mapsto c\}}$. By induction hypothesis we have that $[\![\Gamma_1]\!]_{f'} \vdash [\![x : !T.U]\!]_{f'}$ which by (E-Out) means that $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_{\mathsf{o}}[[\![T]\!], [\![\overline{U}]\!]]$, and $[\![\Gamma_2]\!]_{f''} \vdash f''_v : [\![T]\!]$, and $[\![\Gamma_3]\!]_{f'''}, f'''_x : [\![U]\!] \vdash [\![P]\!]_{f'''}$. Let $f''_v = v, f'''_x = c$ and let $f = f' \cup f'' \cup f''' - \{x \mapsto c\}$. Then the induction hypothesis become $[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathsf{o}}[[\![T]\!], [\![\overline{U}]\!]]$ and $[\![\Gamma_2]\!]_f \vdash v : [\![T]\!]$ and $[\![\Gamma_3]\!]_f, c : [\![U]\!] \vdash [\![P]\!]_{f,\{x \mapsto c\}}$. By applying (T$\pi$-Out) and (T$\pi$-Var) to derive $c : [\![\overline{U}]\!]$ and by using Lemma 6.3.7 and Lemma 6.3.4 to obtain $c : \ell_\sharp\,[W]$, where $[\![U]\!] = \ell_\alpha\,[W]$, we have the following:

$$\frac{[\![\Gamma_2]\!]_f \vdash v : [\![T]\!] \qquad \begin{array}{c}[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathsf{o}}[[\![T]\!], [\![\overline{U}]\!]] \\ c : [\![\overline{U}]\!] \vdash c : [\![\overline{U}]\!]\end{array} \qquad [\![\Gamma_3]\!]_f, c : [\![U]\!] \vdash [\![P]\!]_{f,\{x \mapsto c\}}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f, c : \ell_\sharp\,[W] \vdash f_x!\langle v, c \rangle.[\![P]\!]_{f,\{x \mapsto c\}}}$$

Then by applying (Tπ-Res1) we have the following:

$$\frac{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f, c : \ell_\sharp\,[W] \vdash f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f \vdash (\boldsymbol{\nu} c) f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

which concludes this case.

- Case (T-Brch):

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

To prove that $[\![\Gamma_1 \circ \Gamma_2]\!]_f \vdash [\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f$ where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. By (E-Branching) and Lemma 6.3.5 it means that $[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$. By induction hypothesis we have $[\![\Gamma_1]\!]_{f'} \vdash [\![x : \&\{l_i : T_i\}_{i\in I}]\!]_{f'}$ which by equation (E-Branch) means that $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_{\mathrm{i}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$, and $[\![\Gamma_2]\!]_{f''}, f''_x : [\![T_i]\!] \vdash [\![P_i]\!]_{f''}\ \forall i \in I$. Let $f''_x = c$ and $f = f' \cup f'' - \{x \mapsto c\}$. Then, the induction hypothesis become $[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathrm{i}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$, and $[\![\Gamma_2]\!]_f, c : [\![T_i]\!] \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}\ \forall i \in I$. By applying rules (T$\pi$-Case) and (T$\pi$-Var) for deriving $y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I}$, we have:

$$\frac{\begin{array}{c} y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \vdash y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \\ [\![\Gamma_2]\!]_f, c : [\![T_i]\!] \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}\ \forall i \in I \end{array}}{[\![\Gamma_2]\!]_f, y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

Then, by applying (Tπ-Inp) we have:

$$\frac{\begin{array}{c} [\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathrm{i}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \\ [\![\Gamma_2]\!]_f, y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I} \end{array}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

which concludes this case.

- Case (T-Sel):

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

To prove that $[\![\Gamma_1 \circ \Gamma_2]\!]_f \vdash [\![x \triangleleft l_j.P]\!]_f$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup$

$\mathsf{dom}(\Gamma_2)$. By (E-Selection) and Lemma 6.3.5 it means that $[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$. By induction hypothesis we have that $[\![\Gamma_1]\!]_{f'} \vdash [\![x : \oplus\{l_i : T_i\}_{i\in I}]\!]_{f'}$ which by equation (E-Select) means that $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$, and $[\![\Gamma_2]\!]_{f''}, f''_x : [\![T_j]\!] \vdash [\![P]\!]_{f''}$ $j \in I$. Let $f''_x = c$ and $f = f' \cup f'' - \{x \mapsto c\}$. Then, the induction hypothesis become $[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$, and $[\![\Gamma_2]\!]_f, c : [\![T_j]\!] \vdash [\![P]\!]_{f,\{x\mapsto c\}}$ $j \in I$. By applying axiom (Tπ-Var) to derive $c : [\![\overline{T_j}]\!]$ and rule (Tπ-LVal) we have:

$$\dfrac{\dfrac{}{c : [\![\overline{T_j}]\!] \vdash c : [\![\overline{T_j}]\!]}\ (\text{T}\pi\text{-Var})}{c : [\![\overline{T_j}]\!] \vdash l_j\_c : \langle l_i\_[\![\overline{T_j}]\!]\rangle_{i\in I}}\ (\text{T}\pi\text{-LVal})$$

Then, by rule (Tπ-Out):

$$\dfrac{c : [\![\overline{T_j}]\!] \vdash l_j\_c : \langle l_i\_[\![\overline{T_j}]\!]\rangle_{i\in I} \qquad \begin{array}{c}[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \\ [\![\Gamma_2]\!]_f, c : [\![T_j]\!] \vdash [\![P]\!]_{f,\{x\mapsto c\}}\ j \in I\end{array}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus c : \ell_\sharp[W] \vdash f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}\ (\text{T}\pi\text{-Out})$$

We have used Lemma 6.3.7 and Lemma 6.3.4 to obtain $c : \ell_\sharp[W]$, where $[\![T_j]\!] = \ell_\alpha[W]$ and $[\![\overline{T_j}]\!] = \ell_{\overline{\alpha}}[W]$. We conclude by applying (Tπ-Res1) and Lemma 6.3.3:

$$\dfrac{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f, c : \ell_\sharp[W] \vdash f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

□

Theorem 6.3.11 (Completeness: Process Typing)**.** *If* $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ *then* $\Gamma \vdash P$.

Proof. The proof is done by induction on the structure of P.

- Case $\mathbf{0}$:
  By equation (E-Inaction) we have $[\![\mathbf{0}]\!]_f \stackrel{\mathtt{def}}{=} \mathbf{0}$ and assume $[\![\Gamma]\!]_f \vdash \mathbf{0}$, where $\mathsf{un}([\![\Gamma]\!]_f)$ holds; but then also $\mathsf{un}(\Gamma)$ holds. By applying (T-Inact) we conclude this case.

- Case $P \mid Q$:
  By equation (E-Composition) we have $[\![P \mid Q]\!]_f \stackrel{\mathtt{def}}{=} [\![P]\!]_f \mid [\![Q]\!]_f$ and assume

$[\![\Gamma]\!]_f \vdash [\![P]\!]_f \mid [\![Q]\!]_f$ which by rule (Tπ-Par) means:

$$\frac{\Gamma_1^\pi \vdash [\![P]\!]_f \quad \Gamma_2^\pi \vdash [\![Q]\!]_f}{\Gamma_1^\pi \uplus \Gamma_2^\pi \vdash [\![P]\!]_f \mid [\![Q]\!]_f}$$

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By induction hypothesis we have $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash Q$. Then, by applying (T-Par) we obtain $\Gamma_1 \circ \Gamma_2 \vdash P \mid Q$.

- Case **if** v **then** P **else** Q:
  By equation (E-Conditional) we have $[\![\textbf{if } v \textbf{ then } P \textbf{ else } Q]\!]_f \overset{\texttt{def}}{=} \textbf{if } f_v \textbf{ then } [\![P]\!]_f \textbf{ else } [\![Q]\!]_f$ and assume $[\![\Gamma]\!]_f \vdash \textbf{if } f_v \textbf{ then } [\![P]\!]_f \textbf{ else } [\![Q]\!]_f$, which by rule (T$\pi$-If) means:

  $$\frac{\Gamma_1^\pi \vdash f_v : \texttt{Bool} \quad \Gamma_2^\pi \vdash [\![P]\!]_f \quad \Gamma_2^\pi \vdash [\![Q]\!]_f}{\Gamma_1^\pi \uplus \Gamma_2^\pi \vdash \textbf{if } f_v \textbf{ then } [\![P]\!]_f \textbf{ else } [\![Q]\!]_f}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 6.3.9 we have $\Gamma_1 \vdash v : \texttt{Bool}$. By induction hypothesis we have $\Gamma_2 \vdash P$ and $\Gamma_2 \vdash Q$. Then, by applying (T-If) we obtain $\Gamma_1 \circ \Gamma_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q$.

- Case $(\nu xy)P$:
  By equation (E-Restriction) we have $[\![(\nu xy)P]\!]_f \overset{\texttt{def}}{=} (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$ and by assumption $[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$. Then, either (T$\pi$-Res1) or (T$\pi$-Res2) is the last rule applied. We distinguish these cases:

  – Suppose (Tπ-Res1) is applied

  $$\frac{[\![\Gamma]\!]_f, c : \ell_\alpha\ [W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}}{[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}}\ (\text{T}\pi\text{-Res1})$$

  The premise of the rule asserts that $c \in [\![P]\!]_{f,\{x,y\mapsto c\}}$ and it substitutes both x and y in P, as shown by the encoding of the process. This means that c is used twice and since $(\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$ is well-typed, in order to preserve linearity, it must be the case that $\alpha = \sharp$, which implies $[\![\Gamma]\!]_f, c : \ell_\beta\ [W] \uplus c : \ell_{\overline{\beta}}\ [W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. Let $[\![T]\!] = \ell_\beta\ [W]$, then by Lemma 6.3.7 $[\![\overline{T}]\!] = \ell_{\overline{\beta}}\ [W]$. So, by induction hypothesis we have $\Gamma, x : T, y : \overline{T} \vdash P$. By applying rule (T-Res) we obtain $\Gamma \vdash (\nu xy)P$.

– Suppose (Tπ-Res2) is applied

$$\frac{[\![\Gamma]\!]_f \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}}{[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}}\ (\text{T}\pi\text{-Res2})$$

By induction hypothesis we obtain $\Gamma \vdash P$. Notice that since c is not needed to type $[\![P]\!]_{f,\{x,y\mapsto c\}}$ it means that $c \notin FV([\![P]\!]_{f,\{x,y\mapsto c\}})$. By the encoding of P, we notice that variable c substitutes both variables x and y, hence this implies that $x, y \notin FV(P)$. Let $x : T$ and $y : \overline{T}$ for some type T, be two type assumption that we are going to add to Γ. Since the duality function is defined on T, it means that $T \neq \texttt{Bool}$. On the other hand, $T \neq \mathsf{lin}p$, otherwise the session channel is being thrown away, without being used in P. This means that $T = \mathsf{end}$ and $\overline{T} = \mathsf{end}$ and $\mathsf{un}(\mathsf{end})$. By Lemma 5.5.1 we can obtain $\Gamma, x : T, y : \overline{T} \vdash P$. By applying rule (T-Res) we obtain $\Gamma \vdash (\nu xy)P$.

- Case $x?(y).P$:
  By equation (E-Input) we have $[\![x?(y).P]\!]_f \overset{\texttt{def}}{=} f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f \vdash f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$ which by rule (Tπ-Inp) means:

$$\frac{\Gamma_1^\pi \vdash f_x : \ell_{\mathsf{i}}[T^\pi, U^\pi] \qquad \Gamma_2^\pi, y : T^\pi, c : U^\pi \vdash [\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Gamma]\!]_f \vdash f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$, and $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. By Lemma 6.3.9 we have $\Gamma_1 \vdash x : ?T.U$. By induction hypothesis $\Gamma_2, y : T, x : U \vdash P$ where $T^\pi = [\![T]\!]$, $U^\pi = [\![U]\!]$ and by the encoding of P we notice that c substitutes x. By applying (T-Inp) we obtain $\Gamma_1 \circ \Gamma_2 \vdash x?(y).P$.

- Case $x!\langle v\rangle.P$:
  By equation (E-Output) we have $[\![x!\langle v\rangle.P]\!]_f \overset{\texttt{def}}{=} (\nu c)f_x!\langle v,c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f \vdash (\nu c)f_x!\langle v,c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ which by rules (Tπ- Res1), since c is present in $[\![P]\!]_{f,\{x\mapsto c\}}$, and (T$\pi$- Out) means (there exists) the following derivation:

$$\dfrac{\dfrac{\begin{array}{cc}\Gamma_1^\pi \vdash f_x : \ell_{\mathsf{o}}[T^\pi, U^\pi] & \Gamma_2^\pi \vdash v : T^\pi \\ \Gamma_3^\pi, c : \ell_{\overline{\alpha}}[W] \vdash [\![P]\!]_{f,\{x\mapsto c\}} & c : \ell_\alpha[W] \vdash c : \ell_\alpha[W]\end{array}}{[\![\Gamma]\!]_f, c : \ell_\sharp[T^\pi, U^\pi] \vdash f_x!\langle v,c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}\ (\text{T}\pi\text{-Out})}{[\![\Gamma]\!]_f \vdash (\nu c)f_x!\langle v,c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}\ (\text{T}\pi\text{-Res1})$$

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi \uplus \Gamma_3^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ and $\Gamma_3^\pi = [\![\Gamma_3]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$, and $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) \cup \mathsf{dom}(\Gamma_3)$. Notice that, the type of c is $\ell_\sharp[W]$, meaning that it has both capabilities, because one capability of c is sent along with value v, precisely α and the other one is used in $[\![P]\!]_{f,\{x\mapsto c\}}$, precisely $\overline{\alpha}$. By Lemma 6.3.9 we have $\Gamma_1 \vdash x : !T.U$ where $\ell_{\mathsf{o}}[T^\pi, U^\pi] = [\![!T.U]\!]$, which by following (E-Out) means that $T^\pi = [\![T]\!]$ and $U^\pi = [\![\overline{U}]\!] = \ell_\alpha[W]$. Also, by Lemma 6.3.9 we have $\Gamma_2 \vdash v : T$, and let $f_v = v$. By induction hypothesis we have $\Gamma_3, x : U \vdash P$, where $[\![U]\!] = \ell_{\overline{\alpha}}[W]$, which is obtained by applying Lemma 6.3.7. By applying rule (T-Out) we obtain $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v\rangle.P$

- Case $x \triangleright \{l_i : P_i\}_{i\in I}$:
  By equation (E-Branching) $[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f = f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$ and assume $[\![\Gamma]\!]_f \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$ which by rules (Tπ-Inp) and (Tπ-Case) and Lemma 6.3.3

$$\dfrac{\Gamma_1^\pi \vdash f_x : \ell_{\mathsf{i}}[\langle l_i\_T_i^\pi\rangle_{i\in I}] \quad \dfrac{\text{(T}\pi\text{-Case)}\quad y : \langle l_i\_T_i^\pi\rangle_{i\in I} \vdash y : \langle l_i\_T_i^\pi\rangle_{i\in I} \quad \Gamma_2^\pi, c : T_i^\pi \vdash [\![P_i]\!]_{f,\{x\mapsto c\}} \quad \forall i \in I}{\Gamma_2^\pi, y : \langle l_i\_T_i^\pi\rangle_{i\in I} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}\ \text{(T}\pi\text{-Inp)}}{[\![\Gamma]\!]_f \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$, and $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. By Lemma 6.3.9 we have $\Gamma_1 \vdash x : \&\{l_i : T_i\}_{i\in I}$ where $[\![\&\{l_i : T_i\}_{i\in I}]\!]_f = \ell_{\mathsf{i}}[\langle l_i\_T_i^\pi\rangle_{i\in I}]$ by applying (E-Branch) which implies $\forall i\ [\![T_i]\!] = T_i^\pi$. By induction hypothesis $\Gamma_2, x : T_i \vdash P_i\ \forall i \in I$. By applying rule (T-Brch) we obtain $\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i\in I}$.

- Case $x \triangleleft l_j.P$:
  By equation (E-Selection) we have $[\![x \triangleleft l_j.P]\!]_f = (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ which by rules (Tπ-Res1), since c is present in $[\![P]\!]_{f,\{x\mapsto c\}}$, and (T$\pi$-Out) and (T$\pi$-Lval) means (there exists) the

following derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_1^\pi \vdash f_x : \ell_{\mathsf{o}}\,[\langle l_i\_T_i^\pi\rangle_{i\in I}]}\quad \dfrac{}{\Gamma_2^\pi, c : \overline{T_j^\pi} \vdash [\![P]\!]_{f,\{x\mapsto c\}}}\quad \dfrac{\dfrac{}{c : T_j^\pi \vdash c : T_j^\pi}\quad j \in I}{c : T_j^\pi \vdash l_j\_c : \langle l_i\_T_i^\pi\rangle_{i\in I}}}{[\![\Gamma]\!]_f, c : \ell_\sharp\,[W] \vdash f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}}{[\![\Gamma]\!]_f \vdash (\boldsymbol{\nu} c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

(Tπ-Res1)
(Tπ-Out)
(Tπ-Lval)

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$, and $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Notice that, the type of c is $\ell_\sharp\,[W]$, meaning that it has both capabilities, because one capability of c is sent along with value l_j_c and the other one is used in the continuation $[\![P]\!]_{f,\{x\mapsto c\}}$, this implies $\ell_\sharp\,[W] = T_j^\pi \uplus \overline{T_j^\pi}$. By Lemma 6.3.9 we have $\Gamma_1 \vdash x : \oplus\{l_i : T_i\}_{i\in I}$ where by (E-Select) $\ell_{\mathsf{o}}\,[\langle l_i\_T_i^\pi\rangle_{i\in I}] = [\![\oplus\{l_i : T_i\}_{i\in I}]\!]$ and $T_i^\pi = [\![\overline{T_i}]\!]\ \forall i \in I$. By induction hypothesis we have $\Gamma_2, x : T_j \vdash P$. By applying rule (T-Sel) we obtain $\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P$.

□

6.3.4 Operational Correspondence

In this section we prove the operational correspondence. Before stating the theorem, we introduce the notion of *evaluation context*, which is defined as follows.

Definition 6.3.12 (Evaluation Context)**.** *An* evaluation context *is a process with a hole* $[\cdot]$ *and is produced by the following grammar:*

$$\mathcal{E}[\cdot] \stackrel{def}{=} [\cdot] \mid (\boldsymbol{\nu} xy)\,\mathcal{E}[\cdot]$$

In order to prove the following theorem, we need to extend the definition of process encoding to accommodate substitution.

Definition 6.3.13. *Let Q be a session process and $z \in FV(Q)$ and let $Q[v/z]$ denote process Q where variable z is substituted by value v. Then,*

$$[\![Q[v/z]]\!]_f \stackrel{def}{=} [\![Q]\!]_f[v/z]$$

where $f_v = v$ and $f_z = z$.

In order to simplify the proof of the following theorem, without loss of generality, we assume that a substitution occurs only when ground values are transmitted; in case of variables instead, we assume that the sender sends the exact variable expected by the receiver. This convention validates the above definition. Let $\hookrightarrow$ denote a structural congruence *possibly* extended with a *case normalisation*, namely a reduction denoted by (Rπ-Case).

Theorem 6.3.14 (Operational Correspondence)**.** *Let P be a process in the π-calculus with sessions. The following hold.*

1. *If $P \to P'$ then $[\![P]\!]_f \to\!\hookrightarrow [\![P']\!]_f$,*

2. *If $[\![P]\!]_f \to\!\equiv Q$ then, $\exists\, P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \to^*\!\equiv [\![P']\!]_{f'}$, where f' is the updated f after the communication and $f_x = f_y$ for all $(\boldsymbol{\nu} xy) \in \mathcal{E}[\cdot]$.*

Proof. We split the proof as follows

1. The proof is done by induction on the length of the derivation $P \to P'$.

   - Case (R-Com):

     $$P = (\boldsymbol{\nu} xy)(x!\langle v\rangle.Q_1 \mid y?(z).Q_2 \mid R) \to (\boldsymbol{\nu} xy)(Q_1 \mid Q_2[v/z] \mid R) = P'$$

By the encoding of processes we have

$$
\begin{aligned}
&[\![P]\!]_f \\
&\quad = \quad [\![(\nu xy)(x!\langle v\rangle.Q_1 \mid y?(z).Q_2 \mid R)]\!]_f \\
&\quad \stackrel{\text{def}}{=} \quad (\nu c)\,([\![x!\langle v\rangle.Q_1 \mid y?(z).Q_2 \mid R]\!]_{f,\{x,y\mapsto c\}}) \\
&\quad \stackrel{\text{def}}{=} \quad (\nu c)\,([\![x!\langle v\rangle.Q_1]\!]_{f,\{x\mapsto c\}} \mid [\![y?(z).Q_2]\!]_{f,\{y\mapsto c\}} \mid [\![R]\!]_f) \\
&\quad \stackrel{\text{def}}{=} \quad (\nu c)\,[(\nu c')(c!\langle v, c'\rangle.[\![Q_1]\!]_{f,\{x\mapsto c,c\mapsto c'\}}) \mid \\
&\qquad\qquad\qquad c?(z, c').[\![Q_2]\!]_{f,\{y\mapsto c,c\mapsto c'\}} \mid [\![R]\!]_f] \\
&\quad \equiv \quad (\nu c)\,[(\nu c')(c!\langle v, c'\rangle.[\![Q_1]\!]_{f,\{x\mapsto c,c\mapsto c'\}} \mid \\
&\qquad\qquad\qquad c?(z, c').[\![Q_2]\!]_{f,\{y\mapsto c,c\mapsto c'\}} \mid [\![R]\!]_f)] \\
&\quad \to \quad (\nu c)\,[(\nu c')([\![Q_1]\!]_{f,\{x\mapsto c,c\mapsto c'\}} \mid \\
&\qquad\qquad\qquad [\![Q_2]\!]_{f,\{y\mapsto c,c\mapsto c'\}}[v/z] \mid [\![R]\!]_f)] \\
&\quad \equiv \quad (\nu c')([\![Q_1]\!]_{f,\{x\mapsto c'\}} \mid \\
&\qquad\qquad\qquad [\![Q_2]\!]_{f,\{y\mapsto c'\}}[v/z] \mid [\![R]\!]_f) \\
&\quad \equiv \quad (\nu c')([\![Q_1]\!]_{f'} \mid [\![Q_2]\!]_{f'}[v/z] \mid [\![R]\!]_{f'})
\end{aligned}
$$

Where $f' = f, \{x \mapsto c', y \mapsto c'\}$, and let $f_v = v$ and $f_z = z$. On the other hand we have:

$$
\begin{aligned}
&[\![P']\!]_f \\
&\quad = \quad [\![(\nu xy)(Q_1 \mid Q_2[v/z] \mid R)]\!]_f \\
&\quad \stackrel{\text{def}}{=} \quad (\nu c')([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[v/z] \mid [\![R]\!]_{f,\{x,y\mapsto c'\}}) \\
&\quad \stackrel{\text{def}}{=} \quad (\nu c')([\![Q_1]\!]_{f'} \mid [\![Q_2]\!]_{f'}[v/z] \mid [\![R]\!]_{f'})
\end{aligned}
$$

Where we have used Definition 6.3.13. Which means

$$[\![P]\!]_f \to\equiv [\![P']\!]_f$$

- Case (R-Sel):

$$P = (\nu xy)(x \triangleleft l_j.Q \mid y \triangleright \{l_i : P_i\}_{i\in I} \mid R) \to (\nu xy)(Q \mid P_j \mid R) = P' \quad \text{if } j \in I$$

By the encoding of processes we have

$$
\begin{array}{lll}
[\![P]\!]_f & & \\
& = & [\![(\nu xy)(x \triangleleft l_j.Q \mid y \triangleright \{l_i : P_i\}_{i\in I} \mid R)]\!]_f \\
& \stackrel{\text{def}}{=} & (\nu c)\,([\![x \triangleleft l_j.Q \mid y \triangleright \{l_i : P_i\}_{i\in I} \mid R]\!]_{f,\{x,y\mapsto c\}}) \\
& \stackrel{\text{def}}{=} & (\nu c)\,([\![x \triangleleft l_j.Q]\!]_{f,\{x\mapsto c\}} \mid [\![y \triangleright \{l_i : P_i\}_{i\in I}]\!]_{f,\{y\mapsto c\}} \mid [\![R]\!]_f) \\
& \stackrel{\text{def}}{=} & (\nu c)\,[(\nu c')(c!\langle l_j\_c'\rangle.[\![Q]\!]_{f,\{x\mapsto c,c\mapsto c'\}}) \mid \\
& & \qquad c?(z).\mathbf{case}\, z\, \mathbf{of}\, \{l_i\_c' \triangleright [\![P_i]\!]_{f,\{y\mapsto c,c\mapsto c'\}}\}_{i\in I} \mid [\![R]\!]_f] \\
& \to & (\nu c)\,[(\nu c')([\![Q]\!]_{f,\{x\mapsto c,c\mapsto c'\}} \mid \\
& & \qquad \mathbf{case}\, l_j\_c'\, \mathbf{of}\, \{l_i\_c' \triangleright [\![P_i]\!]_{f,\{y\mapsto c,c\mapsto c'\}}\}_{i\in I} \mid [\![R]\!]_f)] \\
& \to & (\nu c)\,[(\nu c')([\![Q]\!]_{f,\{x\mapsto c,c\mapsto c'\}} \mid \\
& & \qquad [\![P_j]\!]_{f,\{y\mapsto c,c\mapsto c'\}} \mid [\![R]\!]_f)] \\
& \equiv & (\nu c)\,[(\nu c')([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid \\
& & \qquad [\![P_j]\!]_{f,\{x,y\mapsto c'\}} \mid [\![R]\!]_{f,\{x,y\mapsto c'\}})] \\
& = & (\nu c')([\![Q]\!]_{f'} \mid [\![P_j]\!]_{f'} \mid [\![R]\!]_{f'})
\end{array}
$$

Where $f' = f, \{x \mapsto c', y \mapsto c'\}$. On the other hand we have:

$$
\begin{array}{lll}
[\![P']\!]_f & & \\
& = & [\![(\nu xy)(Q \mid P_j \mid R)]\!]_f \\
& \stackrel{\text{def}}{=} & (\nu c')([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}} \mid [\![R]\!]_{f,\{x,y\mapsto c'\}}) \\
& \stackrel{\text{def}}{=} & (\nu c')([\![Q]\!]_{f'} \mid [\![P_j]\!]_{f'} \mid [\![R]\!]_{f'})
\end{array}
$$

Where we have used Definition 6.3.13. Which means

$$[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$$

- Case (R-IfT):

$$\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \to P_1$$

By the encoding of processes we have

$$\begin{array}{lll} [\![P]\!]_f & & \\ & = & [\![\textbf{if } \texttt{true} \textbf{ then } P_1 \textbf{ else } P_2]\!]_f \\ & \stackrel{\texttt{def}}{=} & \textbf{if } \texttt{true} \textbf{ then } [\![P_1]\!]_f \textbf{ else } [\![P_2]\!]_f \\ & \rightarrow & [\![P_1]\!]_f \end{array}$$

- Case (R-IfF):

$$\textbf{if } \texttt{false} \textbf{ then } P_1 \textbf{ else } P_2 \rightarrow P_2$$

By the encoding of processes we have

$$\begin{array}{lll} [\![P]\!]_f & & \\ & = & [\![\textbf{if } \texttt{false} \textbf{ then } P_1 \textbf{ else } P_2]\!]_f \\ & \stackrel{\texttt{def}}{=} & \textbf{if } \texttt{false} \textbf{ then } [\![P_1]\!]_f \textbf{ else } [\![P_2]\!]_f \\ & \rightarrow & [\![P_2]\!]_f \end{array}$$

- Case (R-Res):

$$\frac{P \rightarrow Q}{(\nu xy)P \rightarrow (\nu xy)Q}$$

By the encoding of processes we have

$$[\![(\nu xy)P]\!]_f \stackrel{\texttt{def}}{=} (\nu c)[\![P]\!]_{f,\{x,y \mapsto c\}}$$

Since by the premise of the rule $P \rightarrow Q$ then, by induction hypothesis we have that $[\![P]\!]_{f,\{x,y\mapsto c\}} \rightarrow\!\!\!\!\!\rightarrow [\![Q]\!]_{f,\{x,y\mapsto c\}}$. We conclude that $(\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}} \rightarrow\!\!\!\!\!\rightarrow (\nu c)[\![Q]\!]_{f,\{x,y\mapsto c\}}$ by applying (Rπ-Res) and (Rπ-Struct) and the transitivity of the reduction relation.

- Case (R-Par):

$$\frac{P \rightarrow Q}{P \mid Q \rightarrow P' \mid Q}$$

Since by the premise of the rule $P \rightarrow Q$ then, by induction hypothesis we have that $[\![P]\!]_f \rightarrow\!\!\!\!\!\rightarrow [\![Q]\!]_f$. We conclude that $[\![P]\!]_f \mid [\![Q]\!]_f \rightarrow\!\!\!\!\!\rightarrow [\![P']\!]_f \mid [\![Q]\!]_f$ by applying (Rπ-Par) and (Rπ-Struct) and the transitivity of the reduction relation.

- Case (R-Struct):

$$\frac{P \equiv P',\ P' \rightarrow Q',\ Q' \equiv Q}{P \rightarrow Q}$$

  Trivial case, by applying the encoding and (Rπ-Struct) and the transitivity of the reduction relation.

2. The proof is done by induction on the structure of the session process P. The only cases to be considered are the following:

   - Case $P = P_1 \mid P_2$:
     By (E-Composition) we have $[\![P_1 \mid P_2]\!]_f \stackrel{\text{def}}{=} [\![P_1]\!]_f \mid [\![P_2]\!]_f$ and by hypothesis $[\![P_1]\!]_f \mid [\![P_2]\!]_f \rightarrow\equiv Q$. To prove that there exist $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \rightarrow \mathcal{E}[P']$ and $Q \rightarrow^*\equiv [\![P']\!]_{f'}$. There are the following 3 cases to be considered:

     (a) Only $[\![P_1]\!]_f$ reduces: $[\![P_1]\!]_f \rightarrow R$ which means $Q \equiv R \mid [\![P_2]\!]_f$. By induction hypothesis, since P_1 is a subprocess of P and $[\![P_1]\!]_f \rightarrow R$ then there exist $P'_1, \mathcal{E}'[\cdot]$ such that $\mathcal{E}'[P_1] \rightarrow \mathcal{E}'[P'_1]$ and $R \rightarrow^*\equiv [\![P'_1]\!]_{f''}$. Let $\mathcal{E}[\cdot] = \mathcal{E}'[\cdot]$. Then, by (R-Par) and by structural congruence, (in particular the equation $(\nu xy)(P \mid Q) \equiv (\nu xy)P \mid Q$) $\mathcal{E}[P] = \mathcal{E}[P_1 \mid P_2] \rightarrow \mathcal{E}[P'_1 \mid P_2] = \mathcal{E}[P']$ and so $P' = P'_1 \mid P_2$. Then, $R \mid [\![P_2]\!]_f \rightarrow^*\equiv [\![P'_1]\!]_f \mid [\![P_2]\!]_f = [\![P']\!]_f$, by applying (Rπ-Par) and letting $f = f' = f''$ and this concludes the case.

     (b) Only $[\![P_2]\!]_f$ reduces: this case is the same as the previous one, by changing P_1 with P_2.

     (c) $[\![P_1]\!]_f$ and $[\![P_2]\!]_f$ communicate and both reduce. Since a communication occurs between two processes in parallel, the only possible case is rule (Rπ-Com) – as we already considered rule (Rπ-Par) in the previous cases.– This means that the processes exhibit an input and the other an output action, which is then consumed. Let $[\![P_1]\!]_f$ perform and output action and $[\![P_2]\!]_f$ perform and input action – the other way around is totally similar.– But then, since these are encodings of session processes there are only two possible cases:

        - Let $P_1 \mid P_2 = x!\langle v\rangle.P'_1 \mid y?(z).P'_2$:
          By the encodings, equations (E-Composition), (E-

Output) and (E-Input), we have $[\![P_1]\!]_f \mid [\![P_2]\!]_f \stackrel{\text{def}}{=} (\nu c) f_x!\langle v, c\rangle.[\![P'_1]\!]_{f,\{x\mapsto c\}} \mid f_y?(z, c).[\![P'_2]\!]_{f,\{y\mapsto c\}} \rightarrow [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid [\![P'_2]\!]_{f,\{y\mapsto c\}}[v/z] \equiv Q$. We now show that there exist $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P_1 \mid P_2] \rightarrow \mathcal{E}[P']$ and $Q \rightarrow^* \equiv [\![P']\!]_{f'}$. Choose $\mathcal{E}[\cdot] = (\nu xy)[\cdot]$, then by (R-Com) we have $(\nu xy)(x!\langle v\rangle.P'_1 \mid y?(z).P'_2) \rightarrow (\nu xy)(P'_1 \mid P'_2[v/z])$, and hence $P' = P'_1 \mid P'_2[v/z]$. Trivially, $[\![P']\!]_{f'} = [\![P'_1 \mid P'_2[v/z]]\!]_{f'} \stackrel{\text{def}}{=} [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid [\![P'_2]\!]_{f,\{y\mapsto c\}}[v/z] \equiv Q$ by applying Definition 6.3.13 and letting $f' = f \cup \{x \mapsto c, y \mapsto c\}$.

– Let $P_1 \mid P_2 = x \triangleleft l_j.P'_1 \mid y \triangleright \{l_i : P''_i\}_{i\in I}$:
By the encodings, equations (E-Composition), (E-Selection) and (E-Branching), we have

$$
\begin{aligned}
&[\![P_1]\!]_f \mid [\![P_2]\!]_f \\
&\qquad \stackrel{\text{def}}{=} \quad (\nu c) f_x!\langle l_j\_c\rangle.[\![P'_1]\!]_{f,\{x\mapsto c\}} \mid \\
&\qquad\qquad\quad f_y?(z).\ \mathbf{case}\ z\ \mathbf{of}\ \{l_i\_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I} \\
&\qquad \rightarrow \quad [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid \mathbf{case}\ l_j\_c\ \mathbf{of}\ \{l_i\_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I} \\
&\qquad \rightarrow \quad [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid [\![P''_j]\!]_{f,\{y\mapsto c\}} \\
&\qquad \equiv \quad Q
\end{aligned}
$$

We now show that there exist $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P_1 \mid P_2] \rightarrow \mathcal{E}[P']$ and $Q \rightarrow^* \equiv [\![P']\!]_{f'}$. Choose $\mathcal{E}[\cdot] = (\nu xy)[\cdot]$, then by (R-Sel) we have

$$
\begin{aligned}
\mathcal{E}[P_1 \mid P_2] \quad &= \quad (\nu xy)(x \triangleleft l_j.P'_1 \mid y \triangleright \{l_i : P''_i\}_{i\in I}) \\
&\rightarrow \quad (\nu xy)(P'_1 \mid P''_j) \\
&= \quad \mathcal{E}[P']
\end{aligned}
$$

Trivially, $[\![P']\!]_{f'} = [\![P'_1 \mid P''_j]\!]_{f'} \stackrel{\text{def}}{=} [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid [\![P''_j]\!]_{f,\{y\mapsto c\}} \equiv Q$ where $f' = f \cup \{x \mapsto c, y \mapsto c\}$.

These two cases conclude point 2c.

- Case $P = (\nu xy)P_1$:
  By (E-Restriction) we have $[\![(\nu xy)P_1]\!]_f \stackrel{\texttt{def}}{=} (\nu c)[\![P_1]\!]_{f,\{x,y\mapsto c\}}$ and by hypothesis $(\nu c)[\![P_1]\!]_{f,\{x,y\mapsto c\}} \rightarrow\equiv Q$. This means that the reduction comes from $[\![P_1]\!]_{f,\{x,y\mapsto c\}}$, since in the standard π-calculus the restriction does not enable any further communication in addition to the ones already performed by process $[\![P_1]\!]_{f,\{x,y\mapsto c\}}$. Hence, $Q \equiv (\nu c)R$. By induction hypothesis if $[\![P_1]\!]_{f,\{x,y\mapsto c\}} \rightarrow\equiv R$ then there exists P_1' such that $\mathcal{E}[P_1] \rightarrow \mathcal{E}[P_1']$ and $R \rightarrow^*\equiv [\![P_1']\!]_{f,\{x,y\mapsto c\}}$. To prove that there exists P' such that $\mathcal{E}'[(\nu xy)P_1] \rightarrow \mathcal{E}'[P']$ and $Q \rightarrow^*\equiv [\![P']\!]_{f,\{x,y\mapsto c\}}$. We distinguish 3 cases according to the structure of $\mathcal{E}[\cdot]$.

  (a) $\mathcal{E}[\cdot] = [\cdot]$
  This means that $P_1 \rightarrow P_1'$. Then by (R-Res) we also have $(\nu xy)P_1 \rightarrow (\nu xy)P_1'$. Let $P' = (\nu xy)P_1'$, and $[\![P']\!]_f = (\nu c)[\![P_1']\!]_{f,\{x,y\mapsto c\}}$. By induction hypothesis we have that $R \rightarrow^*\equiv [\![P_1']\!]_{f,\{x,y\mapsto c\}}$. Hence, by applying (R$\pi$-Res) and (R$\pi$-Struct) we have $(\nu c)R \rightarrow^*\equiv (\nu c)[\![P_1']\!]_{f,\{x,y\mapsto c\}}$ which concludes this case.

  (b) $\mathcal{E}[\cdot] = (\nu xy)[\cdot]$
  Let $\mathcal{E}'[\cdot] = [\cdot]$, then the result follows by applying (Rπ-Res) and (Rπ-Struct).

  (c) $\mathcal{E}[\cdot] = (\nu x'y')[\cdot]$
  Let $\mathcal{E}'[\cdot] = \mathcal{E}[\cdot]$, then the result follows by applying (R-Res) and (R-Struct).

- Case $P =$ **if** v **then** P_1 **else** P_2:
  By (E-Conditional) we have $[\![$**if** v **then** P_1 **else** $P_2]\!]_f \stackrel{\texttt{def}}{=}$ **if** f_v **then** $[\![P_1]\!]_f$ **else** $[\![P_2]\!]_f$ and by hypothesis **if** f_v **then** $[\![P_1]\!]_f$ **else** $[\![P_2]\!]_f \rightarrow\equiv Q$. There are two cases to consider, either $v = \texttt{true}$ or $v = \texttt{false}$. Let $v = \texttt{true}$, then $Q = [\![P_1]\!]_f$. To prove that there exist $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \rightarrow \mathcal{E}[P']$ and $Q \rightarrow^*\equiv [\![P']\!]_f$. Let $\mathcal{E}[\cdot] = [\cdot]$ and $P' = P_1$. Trivially the result follows, being $Q \rightarrow^0\equiv [\![P_1]\!]_f$.

□

6.4 Corollaries from Encoding

In this section we show how we can use the encoding presented earlier in this chapter and properties from the standard π-calculus [101] to derive the analogous properties in the π-calculus with sessions [109].

Before proving the Subject Reduction and Type Safety, we give the following lemmas, which are auxiliary lemmas for our results.

Lemma 6.4.1 (Well-Formedness by Encoding)**.** *P is well-formed if and only if $[\![P]\!]_f$ is well-formed.*

Proof. The result follows immediately by applying the definition of well-formedness in session calculus, the definition of well-formedness in linear π-calculus and the encoding of processes given in Fig. 6.2. □

Lemma 6.4.2 (Structural Congruence Preservation under Encoding)**.** *$P \equiv P'$ if and only if $[\![P]\!]_f \equiv [\![P']\!]_f$.*

Proof. The proof is done by case induction on the structural congruence equations. □

We start with an easy result, that of Type Preservation for $\equiv$.

Lemma 6.4.3 (Type Preservation for $\equiv$ by Encoding)**.** *Let $\Gamma \vdash P$ and $P \equiv P'$, then $\Gamma \vdash P'$.*

Proof. By assumption $\Gamma \vdash P$ and $P \equiv P'$. By Theorem 6.3.10 on the soundness of the encoding in process typing we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ and by Lemma 6.4.2 and by Lemma 4.5.3 we have $[\![\Gamma]\!]_f \vdash [\![P']\!]_f$. We conclude by Theorem 6.3.11 on the completeness of the encoding in process typing. □

Now we are ready to prove the Subject Reduction property in the π-calculus with sessions by using our encoding and by the corresponding Subject Reduction in the linear π-calculus.

Theorem 6.4.4 (Subject Reduction for Sessions by Encoding)**.** *If $\Gamma \vdash P$ and $P \to P'$, then $\Gamma \vdash P'$.*

Proof. By assumption $\Gamma \vdash P$ and $P \to P'$. By Theorem 6.3.10 on the soundness of the encoding in process typing we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ and by point 1. of the Operational Correspondence Theorem 6.3.14 we have that $[\![P]\!]_f \to^* \equiv [\![P']\!]_f$ where $\to^* \equiv$ is a short form of "there exists Q such that $[\![P]\!]_f \to Q$ and $Q \hookrightarrow [\![P']\!]_f$" where

$\hookrightarrow$ denotes a structural congruence *possibly* extended with a *case normalisation*, namely a reduction performed by (Rπ-Case). By Subject Reduction in the linear π-calculus, Theorem 4.5.2 we have $[\![\Gamma]\!]_f \vdash Q$ and by Type Preservation for $\equiv$, Lemma 4.5.3 and (possibly) Subject Reduction again we have $[\![\Gamma]\!]_f \vdash [\![P']\!]_f$. The fact that $[\![\Gamma]\!]_f$ is closed, comes from the assumption of the theorem and the operational semantics rules in session calculus, where communication occurs only in restricted variables. By the Theorem 6.3.11 on the completeness of the encoding in process typing, we conclude that $\Gamma \vdash P'$. □

Theorem 6.4.5 (Type Safety for Sessions by Encoding)**.** *If $\vdash P$ then P is well-formed.*

Proof. By assumption $\emptyset \vdash P$. By Theorem 6.3.10 on the soundness of the encoding in process typing we have $\emptyset \vdash [\![P]\!]_f$. By Type Safety in the linear π-calculus, Theorem 4.5.5 we have that $[\![P]\!]_f$ is well-formed. We conclude by Lemma 6.4.1. □

At this point, we can derive the main result on the type system for the π-calculus with sessions, the Type Soundness which asserts the absence of runtime errors of well-typed programs. We prove this theorem by the above Subject Reduction Theorem 6.4.4 and Type Safety Theorem 6.4.5; which we proved from the corresponding ones in the π-calculus, by exploiting the encoding.

Theorem 6.4.6 (Type Soundness for Sessions)**.** *If $\vdash P$ and $P \rightarrow^* Q$, then Q is well-formed.*

Proof. The result follows immediately from Theorem 6.4.4 and Theorem 6.4.5. □

Part III

Advanced Features on Safety by Encoding

Introduction to Part III

In the dyadic session types, different typing features have been added. Subtyping relation for (recursive) session types is added in [43]. Bounded polymorphism is added in [44] as a further extension to subtyping. The authors in [90] add higher-order primitives in order to allow not only mobility of channels but also mobility of processes.

In most of these works, when new typing features are added, they are added on both syntactic categories of standard π-types and session types. Also the syntax of processes will contain both standard process constructs and session primitives. This redundancy in the syntax leads to redundancy also in the theory, and makes the proofs of properties of the language heavy. For instance, if a new type construct is added, the corresponding properties must be checked both on ordinary types and on session types.

In Part III we try to understand at which extent this redundancy is necessary, in the light of the following similarities between session constructs and standard π-calculus constructs. After analysing the effectiveness of the encoding on basic session types, in the following chapters we show its robustness by examining non-trivial extensions, namely subtyping, polymorphism, higher-order and recursion. Furthermore, we present an optimisation of linear channels enabling the reuse of the same channel, instead of a new one, for the continuation of the communication.

Roadmap to Part III Chapters 7, 8, 9 and 10 present the extensions done to the π-calculus with sessions and as a result also to the encoding. They present subtyping, polymorphism, higher-order, and recursion respectively and analyse

how the encoding performs with respect to these extensions. Chapter 11 presents an optimisation of linear channels usage. By enhancing the type system for linear types, we show that it is possible to avoid the redundancy of creating a fresh channel before every output operation.

CHAPTER 7

Subtyping

Subtyping in π-calculus is a relation on the syntactic types and in particular, between channel types. It is based on a notion of substitutability, meaning that language constructs meant to act on channels of the *supertype* can also act on channels of the *subtype*. In particular, if T is a subtype of T', then any channel of type T can be safely used in a context where a channel of type T' is expected.

7.1 Subtyping Rules

Subtyping has been studied extensively in the standard π-calculus [98, 101]. It has also been studied later in the π-calculus with sessions [43]. In this section we show that subtyping in the π-calculus is enough to derive subtyping in session types. Notice that we are confining ourselves to the set of finite types, namely, no recursion in present yet. Subtyping rules for the π-calculus with sessions are given in Fig. 7.1 whether the ones for the standard π-calculus are given in Fig. 7.2. We use the symbol $<:$ for subtyping in session types, and $\leq$ for subtyping in the standard π-calculus. We start with subtyping rules for session types. Rules (S-Bool) and (S-End) state the reflexivity of subtyping on a boolean type and on a terminated channel type, respectively. Rules (S-Inp) and (S-Out) define subtyping on input and output session types. The input rule states that subtyping is covariant on the first type, whether the output rule states that subtyping is contra-

$$\frac{}{\texttt{Bool} <: \texttt{Bool}}\ (\text{S-Bool}) \qquad \frac{}{\texttt{end} <: \texttt{end}}\ (\text{S-End})$$

$$\frac{T <: T' \qquad U <: U'}{?T.U <: ?T'.U'}\ (\text{S-Inp}) \qquad \frac{T' <: T \qquad U <: U'}{!T.U <: !T'.U'}\ (\text{S-Out})$$

$$\frac{I \subseteq J \qquad T_i <: T'_j \quad \forall i \in I}{\&\{l_i : T_i\}_{i\in I} <: \&\{l_j : T'_j\}_{j\in J}}\ (\text{S-Brch}) \qquad \frac{I \supseteq J \qquad T_i <: T'_j \quad \forall j \in J}{\oplus\{l_i : T_i\}_{i\in I} <: \oplus\{l_j : T'_j\}_{j\in J}}\ (\text{S-Sel})$$

Figure 7.1: Subtyping rules for the π-calculus with sessions

$$\frac{}{T \leq T}\ (\text{S}\pi\text{-Refl}) \qquad \frac{T \leq T' \qquad T' \leq T''}{T \leq T''}\ (\text{S}\pi\text{-Trans})$$

$$\frac{\widetilde{T} \leq \widetilde{T'}}{\ell_{\mathsf{i}}\,[\widetilde{T}] \leq \ell_{\mathsf{i}}\,[\widetilde{T'}]}\ (\text{S}\pi\text{-ii}) \qquad \frac{\widetilde{T'} \leq \widetilde{T}}{\ell_{\mathsf{o}}\,[\widetilde{T}] \leq \ell_{\mathsf{o}}\,[\widetilde{T'}]}\ (\text{S}\pi\text{-oo})$$

$$\frac{I \subseteq J \qquad T_i \leq T'_j \quad \forall i \in I}{\langle l_i\_T_i\rangle_{i\in I} \leq \langle l_j\_T'_j\rangle_{j\in J}}\ (\text{S}\pi\text{-Variant})$$

Figure 7.2: Subtyping rules for the standard π-calculus

variant on the first type. Subtyping is co-variant on the continuation type, for both the input and the output rules. Rules (S-Brch) and (S-Sel) are similar to the previous ones, being branch and select generalisation of input and output, respectively. These rules state that subtyping is co-variant in depth in the types of values being transmitted. Rule (S-Brch) states that subtyping is co-variant in breadth, whether for (S-Brch) it is contra-variant in breadth. We now explain subtyping on the standard π- types. Rules (Sπ- Refl) and (Sπ- Trans) state that the subtyping relation is a pre-order. Rules (Sπ-ii) and (Sπ-oo) define subtyping for input and output channel types, respectively. The input operation is co-variant in the carried types, whether the output operation is contra-variant. Rule (Sπ-Variant) defines subtyping for variant types which is is co-variant both in depth and in breadth.

7.2 Properties

We use the encoding of the π-calculus with sessions to derive basic properties of session types. Therefore, it is important to prove the validity of subtyping, which is necessary in order to extend Subject Reduction and Type Safety.

Lemma 7.2.1 (Subtyping on dual types)**.** *If* $[\![T]\!] \leq [\![T']\!]$*, then* $[\![\overline{T'}]\!] \leq [\![\overline{T}]\!]$.

Proof. The lemma follows immediately by the definition of encoding, the duality function in standard π-types and the subtyping rules presented in Fig. 7.2. □

Theorem 7.2.2 (Soundness of Subtyping)**.** *If* $T <: T'$ *then* $[\![T]\!] \leq [\![T']\!]$.

Proof. The proof is done by induction on the last rule applied in the derivation of $T <: T'$.

- Case (S-Bool):
  It means $\texttt{Bool} <: \texttt{Bool}$. By equation (E-Bool) and rule (Sπ-Refl) we obtain $\texttt{Bool} \leq \texttt{Bool}$ and this concludes the case.

- Case (S-End):
  It means $\texttt{end} <: \texttt{end}$. By equation (E-End) and rule (Sπ- Refl) we obtain $\emptyset[] \leq \emptyset[]$ and this concludes the case.

- Case (S-Inp):

$$\frac{T <: T' \qquad U <: U'}{?T.U <: ?T'.U'}$$

  By induction hypothesis we have that $[\![T]\!] \leq [\![T']\!]$ and $[\![U]\!] \leq [\![U']\!]$. To prove $[\![?T.U]\!] \leq [\![?T'.U']\!]$. By applying (E-Inp) we obtain $[\![?T.U]\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{i}}\ [[\![T]\!], [\![U]\!]]$ and $[\![?T'.U']\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{i}}\ [[\![T']\!], [\![U']\!]]$. By applying rule (S$\pi$- ii) on the induction hypothesis, we obtain the result.

- Case (S-Out):

$$\frac{T' <: T \qquad U <: U'}{!T.U <: !T'.U'}$$

  By induction hypothesis we have that $[\![T']\!] \leq [\![T]\!]$ and $[\![U]\!] \leq [\![U']\!]$. To prove $[\![!T.U]\!] \leq [\![!T'.U']\!]$. By applying (E-Out) we obtain $[\![!T.U]\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{o}}\ [[\![T]\!], [\![\overline{U}]\!]]$ and $[\![!T'.U']\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{o}}\ [[\![T']\!], [\![\overline{U'}]\!]]$. By Lemma 7.2.1 we get $[\![\overline{U'}]\!] \leq [\![\overline{U}]\!]$. By applying rule (S$\pi$- oo) on the induction hypothesis, we obtain the result.

- Case (S-Brch):

$$\frac{I \subseteq J \qquad T_i <: T'_j \quad \forall i \in I}{\&\{l_i : T_i\}_{i\in I} <: \&\{l_j : T'_j\}_{j\in J}}$$

By induction hypothesis we have that $[\![T_i]\!] \leq [\![T'_j]\!] \quad \forall i \in I$. To prove $[\![\&\{l_i : T_i\}_{i\in I}]\!] \leq [\![\&\{l_j : T'_j\}_{j\in J}]\!]$. By applying (E-Branch) we obtain $[\![\&\{l_i : T_i\}_{i\in I}]\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{i}}\ [\langle l_i\text{-}[\![T_i]\!]\rangle_{i\in I}]$ and $[\![\&\{l_j : T'_j\}_{j\in J}]\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{i}}\ [\langle l_j\text{-}[\![T'_j]\!]\rangle_{j\in J}]$. By applying rules (S$\pi$-Variant) and (S$\pi$-ii) on the induction hypothesis, we obtain the result.

- Case (S-Sel):

$$\frac{I \supseteq J \qquad T_i <: T'_j \quad \forall j \in J}{\oplus\{l_i : T_i\}_{i\in I} <: \oplus\{l_j : T'_j\}_{j\in J}}$$

By induction hypothesis we have that $[\![T_i]\!] \leq [\![T'_j]\!] \quad \forall j \in J$. To prove $[\![\oplus\{l_i : T_i\}_{i\in I}]\!] \leq [\![\oplus\{l_j : T'_j\}_{j\in J}]\!]$. By applying (E-Select) we obtain $[\![\oplus\{l_i : T_i\}_{i\in I}]\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{o}}\ [\langle l_i\text{-}[\![\overline{T_i}]\!]\rangle_{i\in I}]$ and $[\![\oplus\{l_j : T'_j\}_{j\in J}]\!] \stackrel{\texttt{def}}{=} \ell_{\mathsf{o}}\ [\langle l_j\text{-}[\![\overline{T'_j}]\!]\rangle_{j\in J}]$. By Lemma 7.2.1 we get $[\![\overline{T'_j}]\!] \leq [\![\overline{T}_j]\!] \quad \forall j \in J$. By applying rules (S$\pi$-Variant) and (S$\pi$-oo) on the induction hypothesis, we obtain the result.

□

Theorem 7.2.3 (Completeness of Subtyping)**.** *If* $[\![T]\!] \leq [\![T']\!]$*, then* $T <: T'$.

Proof. The proof is done by induction on the structure of session types T, T'. The only cases to consider are the following.

- Case $T = T' = \texttt{Bool}$:
  By (E-Bool) we have $[\![T]\!] = [\![T']\!] = \texttt{Bool}$. By rule (S$\pi$-Refl) we have that $\texttt{Bool} \leq \texttt{Bool}$. By applying rule (S-Bool) we obtain the result.

- Case $T = T' = \texttt{end}$:
  By (E-End) we have $[\![T]\!] = [\![T']\!] = \emptyset[]$. By rule (S$\pi$-Refl) we have that $\texttt{end} \leq \texttt{end}$. By applying rule (S-End) we obtain the result.

- Case $T = ?T_1.U_1$ and $T' = ?T_2.U_2$:
  Assume that $[\![?T_1.U_1]\!] \leq [\![?T_2.U_2]\!]$, which means $\ell_{\mathsf{i}}[[\![T_1]\!], [\![U_1]\!]] \leq \ell_{\mathsf{i}}[[\![T_2]\!], [\![U_2]\!]]$. In order to establish this relation, the last rule applied must be (Sπ-ii), which by its premise asserts that $[\![T_1]\!] \leq [\![T_2]\!]$ and $[\![U_1]\!] \leq [\![U_2]\!]$.

By induction hypothesis we have that $T_1 <: T_2$ and $U_1 <: U_2$. By applying rule (S-Inp) on the induction hypothesis we obtain $?T_1.U_1 <: ?T_2.U_2$.

- Case $T = !T_1.U_1$ and $T' = !T_2.U_2$:
  Assume that $[\![!T_1.U_1]\!] \leq [\![!T_2.U_2]\!]$, which means $\ell_o[[\![T_1]\!], [\![\overline{U_1}]\!]] \leq \ell_o[[\![T_2]\!], [\![\overline{U_2}]\!]]$. In order to establish this relation, the last rule applied must be (Sπ- oo), which by its premise asserts that $[\![T_2]\!] \leq [\![T_1]\!]$ and $[\![\overline{U_2}]\!] \leq [\![\overline{U_1}]\!]$. By Lemma 7.2.1 and the idempotence of duality, we get $[\![U_1]\!] \leq [\![U_2]\!]$. By induction hypothesis we have that $T_2 <: T_1$ and $U_1 <: U_2$. By applying rule (S-Out) on the induction hypothesis we obtain $!T_1.U_1 <: !T_2.U_2$.

- Case $T = \&\{l_i : T_i\}_{i\in I}$ and $T' = \&\{l_j : T'_j\}_{j\in J}$:
  Assume that $[\![\&\{l_i : T_i\}_{i\in I}]\!] \leq [\![\&\{l_j : T'_j\}_{j\in J}]\!]$, which means $\ell_i\,[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \leq \ell_i\,[\langle l_j\_[\![T'_j]\!]\rangle_{j\in J}]$. In order to establish this relation, the last rule applied must be (Sπ- ii), which by its premise asserts that $\langle l_i\_[\![T_i]\!]\rangle_{i\in I} \leq \langle l_j\_[\![T'_j]\!]\rangle_{j\in J}$. By rule (S$\pi$- Variant) this means that $[\![T_i]\!] \leq [\![T'_j]\!] \quad \forall i \in I$ and $I \subseteq J$. By induction hypothesis we have that $T_i <: T'_j \quad \forall i \in I$ and $I \subseteq J$. By applying rule (S-Brch) on the induction hypothesis we obtain $\&\{l_i : T_i\}_{i\in I} <: \&\{l_j : T'_j\}_{j\in J}$.

- Case $T = \oplus\{l_i : T_i\}_{i\in I}$ and $T' = \oplus\{l_j : T'_j\}_{j\in J}$:
  Assume that $[\![\oplus\{l_i : T_i\}_{i\in I}]\!] \leq [\![\oplus\{l_j : T'_j\}_{j\in J}]\!]$, which means $\ell_o\,[\langle l_i\_[\![\overline{T_i}]\!]\rangle_{i\in I}] \leq \ell_o\,[\langle l_j\_[\![\overline{T'_j}]\!]\rangle_{j\in J}]$. In order to establish this relation, the last rule applied must be (Sπ- oo), which by its premise asserts that $\langle l_j\_[\![\overline{T'_j}]\!]\rangle_{j\in J} \leq \langle l_i\_[\![\overline{T_i}]\!]\rangle_{i\in I}$. By rule (S$\pi$- Variant) this means that $[\![\overline{T'_j}]\!] \leq [\![\overline{T_i}]\!] \quad \forall j \in J$ and $J \subseteq I$. By Lemma 7.2.1 and the idempotence of duality, we obtain $[\![T_i]\!] \leq [\![T_j]\!] \quad \forall j \in J$ and $J \subseteq I$. By induction hypothesis we have that $T_i <: T'_j \quad \forall j \in J$ and $J \subseteq I$. By applying rule (S-Sel) on the induction hypothesis we obtain $\oplus\{l_i : T_i\}_{i\in I} <: \oplus\{l_j : T'_j\}_{j\in J}$.

□

In order to benefit from the subtyping relation, we introduce the *subsumption rule* to the typing system, both on the π-calculus with and without sessions.

$$\frac{\Gamma \vdash x : T \qquad T \textit{ subtype } T'}{\Gamma \vdash x : T'}$$

where *subtype* is instantiated with <: or ≤ depending on the calculus where it applied. Then, we can prove the following results of Soundness and Completeness.

Lemma 7.2.4 (Value Typing)**.** $\Gamma \vdash v : T$ *if and only if* $[\![\Gamma]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$.

Proof. The proof is split as follows.

(⇒) The cases are exactly as in Lemma 6.3.8 plus the case of subsumption which trivial, since this rule is added on both calculi.

(⇐) The cases are exactly as in Lemma 6.3.9 plus the case of subsumption which trivial, since this rule is added on both calculi.

□

Theorem 7.2.5 (Process Typing)**.** *If* $\Gamma \vdash P$ *if and only if* $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$.

Proof. The proof is split as follows. The proof is split as follows.

(⇒) The cases are exactly as in Theorem 6.3.10 where instead of Lemma 6.3.8, we apply Lemma 7.2.4.

(⇐) The cases are exactly as in Theorem 6.3.11 where instead of Lemma 6.3.9 we apply Lemma 7.2.4.

□

CHAPTER 8

Polymorphism

Polymorphism is a common and useful type abstraction in programming languages as it allows generic operations by using an expression with several types. In Chapter 7 we exploited subtyping on both session types and standard π-types, which is another form of type abstraction.

However, it is only possible for a process to take advantage of subtyping in sending messages to another process, not in receiving from it.

In [44] the author takes a step further and extends subtyping by adding *bounded polymorphism* to overcome this problem. To the best of our knowledge, this is the first work on polymorphism in session types and the first work on bounded polymorphism in the π-calculus. In Section 8.2 we will show how one can get bounded polymorphism in session types, by adding bounded polymorphism in the π-calculus types and by extending our encoding.

Another form of polymorphism is the *parametric polymorphism* that is already present and well studied in π-calculus [101], and in general is the form of polymorphism best known in programming languages. In Section 8.1 we show that, by extending the encoding and by adding parametric polymorphism to the syntax of types (and terms) in session calculus, we get all the properties in the polymorphic sessions for free by deriving them from the theory of the polymorphic π-calculus.

This shows that this duplication is not necessary: all the theory of polymorphism in session types can be derived by the corresponding theory in the π-calculus. This holds for the standard parametric polymorphism as well as for

bounded polymorphism. We first start with parametric polymorphism, since it is the simplest form and then proceed with the bounded one.

8.1 Parametric Polymorphism

In this section we consider the parametric polymorphism. We present the syntax of types and terms, then we show the typing rules and the reduction rules added to relate the new process constructs with the type constructs. Most importantly, we extend the encoding and by proving again soundness and completeness of typing derivations for values and processes, we show our encoding is robust.

8.1.1 Syntax

The syntax of the polymorphic π-calculus with and without sessions is given in the following. Notice that, since the new constructs for polymorphic types and terms are the same for both the π-calculi with and without sessions, for simplicity, we present them under the same grammar. We will distinguish them in the context and often we will refer to the standard π-calculus constructs as the encoded constructs of the π-calculus with sessions.

$T ::=$	X	type variable
	$\langle X; T \rangle$	polymorphic type
$P ::=$	**open** v **as** $(X; x)$ **in** P	unpacking process
$v ::=$	$\langle T; v \rangle$	polymorphic value
$\Delta ::=$	$\Delta, X \mid \emptyset$	type variable environment

Figure 8.1: Syntax of parametric polymorphic constructs

We extend both syntaxes of the π-calculus with sessions and the standard π-calculus with type variable X and *polymorphic type* $\langle X; T \rangle$.

Modifications in the syntax of types introduce modifications in the syntax of terms, as expected. So, we add *polymorphic value* $\langle T; v \rangle$ and *unpacking process* **open** v **as** $(X; x)$ **in** P, on both calculi.

To conclude, we add another typing environment Δ containing polymorphic type variables. We will present the new typing judgements in the following.

8.1.2 Semantics

The reduction rule for the unpacking process is given below.

$$(\mathrm{R}[\pi]\text{-}\textsc{Unpack}) \quad \textbf{open } \langle T; v\rangle \textbf{ as } (X; x) \textbf{ in } P \rightarrow P[T/X][v/x]$$

This reduction rule holds on both calculi, with and without sessions. We distinguish it by the presence of π in the rule name. This reduction is similar to the **case** reduction, as it does not require any communication. We can refer to it, as *unpack normalisation*, – in analogy to *case normalisation*.– It states that an unpacking process **open** $\langle T; v\rangle$ **as** $(X; x)$ **in** P, where the guard is a polymorphic value $\langle T; v\rangle$ reduces to process P where two substitutions occur: type T substitutes type variable X and value v substitutes the placeholder variable x.

8.1.3 Typing Rules

We give now the typing rules for the π-calculus with and without sessions. Typing judgements are of the new form $\Gamma; \Delta \vdash v : T$ or $\Gamma; \Delta \vdash P$, where Γ is the type environment introduced in Section 5.4 for the π- calculus with sessions and in Section 4.4 for the standard π-calculus, whether Δ collects the polymorphic type variables, as shown in the previous section.

Typing rules are presented in Fig. 8.2. Again, we present in the same figure, both the typing rules for the session π- calculus and the typing rules for the standard one. In order to distinguish them, we use $[\pi]$ in square brackets, which means optional: where π is present, then the rule refers to the standard π-calculus, otherwise refers to the session π-calculus.

$$\frac{\Gamma; \Delta \vdash v : T[T'/X]}{\Gamma; \Delta \vdash \langle T'; v\rangle : \langle X; T\rangle}\ (\mathrm{T}[\pi]\text{-}\textsc{PolyVal})$$

$$\frac{\Gamma; \Delta \vdash v : \langle X; T\rangle \qquad \Gamma, x : T; \Delta, X \vdash P}{\Gamma; \Delta \vdash \textbf{open } v \textbf{ as } (X; x) \textbf{ in } P}\ (\mathrm{T}[\pi]\text{-}\textsc{Unpack})$$

Figure 8.2: Typing rules for polymorphic constructs

Rule (T[π]-PolyVal) asserts that a polymorphic value $\langle T'; v\rangle$ is of a polymorphic type $\langle X; T\rangle$, whenever the value v is of type T, where T' substitutes the type variable X. Rule (T[π]-Unpack) states the well-typedness of the unpacking process.

Process **open** v **as** $(X; x)$ **in** P is well-typed if the guard v is of a polymorphic type $\langle X; T\rangle$ on type variable X and the process P is well-typed under x of type T.

8.1.4 Encoding

The encoding of polymorphic types and terms constructs is an homomorphism and is presented by the following equations.

$$
\begin{aligned}
[\![X]\!] &\stackrel{\text{def}}{=} X && \text{(E-PolyVar)}\\
[\![\langle X; T\rangle]\!] &\stackrel{\text{def}}{=} \langle X; [\![T]\!]\rangle && \text{(E-PolyType)}\\
[\![\langle T; v\rangle]\!]_f &\stackrel{\text{def}}{=} \langle [\![T]\!]; f_v\rangle && \text{(E-PolyVal)}\\
[\![\textbf{open } v \textbf{ as } (X; x) \textbf{ in } P]\!]_f &\stackrel{\text{def}}{=} \textbf{open } f_v \textbf{ as } (X; f_x) \textbf{ in } [\![P]\!]_f && \text{(E-Unpack)}
\end{aligned}
$$

Figure 8.3: Encoding of parametric polymorphic constructs

Equation (E-PolyVar) states that the encoding of a type variable X is the identity function. Equation (E-PolyType) states that the encoding of a polymorphic type $\langle X; T\rangle$ added to the session types is a polymorphic type $\langle X; [\![T]\!]\rangle$ added to the standard π-types, on the same type variable X and carrying $[\![T]\!]$. The encoding of a polymorphic value and a polymorphic process is parametrised in a function f that renames variables in the session term in variables in the standard π-term. The definition of f is given in Section 6.2. Equation (E-PolyVal) states that the encoding of a polymorphic value $\langle T; v\rangle$ added to the the session π-calculus is a polymorphic value $\langle [\![T]\!]; f_v\rangle$ added to the standard π-calculus having type the encoding of T and the value v is renamed according f, resulting in f_v. Equation (E-Unpack) states that the encoding of the unpacking process **open** v **as** $(X; x)$ **in** P added to the session π-calculus is the unpacking process **open** f_v **as** $(X; f_x)$ **in** $[\![P]\!]_f$ added to the standard π-calculus where the guard is the renamed value f_v, the polymorphic placeholder x is renamed as f_x and process P is encoded in f.

To conclude, the encoding of type environments is given in the following.

$$[\![\Gamma; \Delta]\!]_f \stackrel{\text{def}}{=} [\![\Gamma]\!]_f; \Delta$$

We encode the type environment Γ as presented in Fig. 6.3, whether on Δ the encoding is the identity function, since the encoding of type variables is the identity function, and Δ is a collection of type variables.

8.1.5 Properties of the Encoding

In this section we prove the correctness of the encoding of polymorphic type constructs and polymorphic term constructs, by proving the soundness and completeness of typing derivations for polymorphic processes and values as well as the operational correspondence. We start with the following definition, which is similar to Definition 6.3.13.

Definition 8.1.1. *Let T be a session type and let $T[T'/X]$ denote type T where type variable X is substituted by type T'. Then,*

$$[\![T[T'/X]]\!] = [\![T]\!][[\![T']\!]/X]$$

To complete Lemma 6.3.8 of soundness and Lemma 6.3.9 of completeness of typing values via the encoding, it suffices to add the case for polymorphic values. However, adding this case requires modification in the typing judgements: previous typing judgements of the form $\Gamma \vdash v : T$ can be now written as $\Gamma; \Delta \vdash v : T$, where Δ can be empty in absence of polymorphism. We give the lemma for the correctness of typing derivations for polymorphic values in the following.

Lemma 8.1.2 (Correctness of Typing Polymorphic Value)**.** $\Gamma; \Delta \vdash \langle T'; v\rangle : \langle X; T\rangle$ *if and only if* $[\![\Gamma; \Delta]\!]_f \vdash [\![\langle T'; v\rangle]\!]_f : [\![\langle X; T\rangle]\!]$.

Proof. We split the proof as follows:

- (*only if*):
  The proof is done by induction on the last rule applied for deriving $\Gamma; \Delta \vdash \langle T'; v\rangle : \langle X; T\rangle$
  $$\frac{\Gamma; \Delta \vdash v : T[T'/X]}{\Gamma; \Delta \vdash \langle T'; v\rangle : \langle X; T\rangle}$$
  By induction hypothesis and Definition 8.1.1 we have that $[\![\Gamma; \Delta]\!]_{f'} \vdash f'_v : [\![T]\!][[\![T']\!]/X]$. Choose $f = f'$, then by rule (Tπ-PolyVal), and the equations (E-PolyType) and (E-PolyVal), we obtain the result.

- (*if*):
  The proof is done by induction on the structure of the polymorphic value. By (E-PolyVal) we have $[\![\langle T'; v\rangle]\!]_f \stackrel{\text{def}}{=} \langle [\![T]\!]; f_v\rangle$ and assume $[\![\Gamma; \Delta]\!]_f \vdash \langle [\![T]\!]; f_v\rangle : \langle X; [\![T]\!]\rangle$, which means that the last typing rule applied is (Tπ-PolyVal).
  $$\frac{[\![\Gamma; \Delta]\!]_f \vdash f_v : [\![T]\!][[\![T']\!]/X]}{[\![\Gamma; \Delta]\!]_f \vdash \langle [\![T]\!]; f_v\rangle : \langle X; [\![T]\!]\rangle}$$

By induction hypothesis and by Definition 8.1.1 we obtain $\Gamma \vdash v : T[T'/X]$. We conclude by applying (T-PolyVal).

□

To complete Theorem 6.3.10 of soundness and Theorem 6.3.11 of completeness of typing processes via the endocing, it suffices to add the case for the unpack process. As with values, adding this case to the proofs of the previous theorems requires modification in the typing judgements: previous typing judgements of the form $\Gamma \vdash P$ can be now written as $\Gamma; \Delta \vdash P$, where Δ can be empty in absence of polymorphism.

The following theorem states the correctness of typing derivations for polymorphic processes. Namely, an unpacking session process is well-typed if and only if the encoded unpacking process is well-typed.

Theorem 8.1.3 (Correctness of Typing Unpacking)**.** $\Gamma; \Delta \vdash \mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P$ *if and only if* $[\![\Gamma; \Delta]\!]_f \vdash [\![\mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P]\!]_f$.

Proof. We split the proof as follows:

- (*only if*):
  The proof is done by induction on the last typing rule used for deriving $\Gamma; \Delta \vdash \mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P$. By rule (T-Unpack) in the session calculus, it means that
  $$\frac{\Gamma; \Delta \vdash v : \langle X; T\rangle \qquad \Gamma, x : T; \Delta, X \vdash P}{\Gamma; \Delta \vdash \mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P}$$
  By induction hypothesis we have that $[\![\Gamma; \Delta]\!]_{f'} \vdash f'_v : \langle X; [\![T]\!]\rangle$ and $[\![\Gamma, x : T; \Delta, X]\!]_{f'} \vdash [\![P]\!]_{f'}$ and let f be such that $f' = f \cup \{x \mapsto f'_x\}$ meaning that f and f' coincide in $\mathsf{dom}(\Gamma)$. Then, by applying (E-Unpack) and rule (Tπ-Unpack) in the π-calculus, we obtain the result.

- (*if*):
  The proof is done by induction on the structure of the unpacking process. By (E-Unpack) we have $[\![\mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P]\!]_f \overset{\mathsf{def}}{=} \mathbf{open}\ f_v\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f$, and assume $[\![\Gamma; \Delta]\!]_f \vdash \mathbf{open}\ f_v\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f$. This means the last rule used is (Tπ-Unpack):
  $$\frac{[\![\Gamma; \Delta]\!]_f \vdash f_v : \langle X; [\![T]\!]\rangle \qquad [\![\Gamma]\!]_f, f'_x : [\![T]\!]; \Delta, X \vdash [\![P]\!]_{f'}}{[\![\Gamma; \Delta]\!]_f \vdash \mathbf{open}\ f_v\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f}$$

where $f' = f \cup \{x \mapsto f'_x\}$ meaning that f and f' coincide in $\mathsf{dom}(\Gamma)$. By the induction hypothesis we obtain $\Gamma; \Delta \vdash v : \langle X; T \rangle$ and $\Gamma, x : T; \Delta, X \vdash P$. Then, by applying (T-Unpack), we obtain the result.

□

To complete the operational correspondence, Theorem 6.3.14, we add the case for the new transition (R[π]-Unpack) for polymorphic processes. We consider this case separately in the following theorem.

Theorem 8.1.4 (OC for Polymorphic Process)**.** *Let P be a process in the π-calculus with sessions. Then,*

1. *If* $\mathbf{open}\ \langle T; v \rangle\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P \rightarrow P[T/X][v/x]$ *then,* $[\![\mathbf{open}\ \langle T; v \rangle\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P]\!]_f \rightarrow\equiv [\![P[T/X][v/x]]\!]_f$*;*

2. *If* $[\![\mathbf{open}\ \langle T; v \rangle\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P]\!]_f \rightarrow\equiv Q$ *then,* $\exists\ P'$ *such that* $\mathbf{open}\ \langle T; v \rangle\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P \rightarrow P'$ *and* $Q \equiv [\![P']\!]_f$.

Proof. The two cases are as follows

1. The results follows immediately by the equation (E-Unpack), Definition 8.1.1 and Definition 6.3.13 and by reduction (Rπ-Unpack).

2. The results follows immediately by the equation (E-Unpack), rule (R-Unpack) and the definition of structural congruence. We apply Definition 8.1.1 and Definition 6.3.13.

□

8.2 Bounded Polymorphism

In this section we consider the bounded polymorphism for session types, introduces in [44]. This kind of polymorphism has not been studied yet in the standard typed π-calculus; we add it and show how we can derive bounded polymorphism in session types passing through the π-types and by using the extended encoding in order to accommodate the new type and process constructs.

8.2.1 Syntax

In this section we present both syntaxes of π-calculus with sessions and the standard one. We present them in two separate grammars, since this time there are different types and processes constructs added on the π-calculi.

Syntax of bounded polymorphic constructs in session π- calculus We first modify the syntax of (pre)types with new constructs to accommodate bounded polymorphism, as in [44]. We report only the new constructs or the modifications made to the syntaxes of session types and session processes introduced in Section 5.3 and Section 5.1, respectively.

$T_s ::=$	B	basic type
$B ::=$	D	data type
	X	type variable
$p ::=$	$\oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I}$	bounded polymorphic select
	$\&\{l_i(X_i <: B_i) : T_i\}_{i\in I}$	bounded polymorphic branch
$P_s ::=$	$x \triangleleft l_j(B).P$	bounded polymorphic selection
	$x \triangleright \{l_i(X_i <: B_i) : P_i\}_{i\in I}$	bounded polymorphic branching

Figure 8.4: Syntax of bounded polymorphic session constructs

Types produced by T_s, in addition to $\mathsf{lin}p \mid \mathtt{end}$, include data types D, which in particular can be `Bool` or other ground types, like `Int`, `String`..., or data structure on the ground types, like list of boolean values or list of integers, and type variable X. Recall that in Section 5.3 we adopted only the boolean type and stated that every other ground type can be added as well as data structures. In this section, we adopt the same syntax as in the original paper [44], so we include data structures explicitly. The pretypes produced by p report modifications only in the select and branch types, where labels are annotated with conditions of the form $(X_i <: B_i)$, resulting in $\oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I}$ and $\&\{l_i(X_i <: B_i) : T_i\}_{i\in I}$, respectively. This basically means that the variables X_i which occur in T_i can be instantiated by types that respect the condition, where $<:$ indicates the subtyping relation on session types presented in Fig. 7.1. Processes produced by P_s, as a consequence of the select and branch types, report modifications only in the selection and branching processes, namely $x \triangleleft l_j(B).P$ and $x \triangleright \{l_i(X_i <: B_i) : P_i\}_{i\in I}$,

respectively. In the bounded polymorphic branching every label l_i is annotated with the condition $(X_i <: B_i)$ which has the same meaning as for the types. In the bounded polymorphic selection the selected label is accompanied also with a selected basic type. The reduction rules, introduced in the next section, give a better understanding of how label annotations are used.

Type duality for the bounded polymorphic pretypes is defined in Fig. 8.5 and is as expected, by following the standard definition of type duality in session types.

$$\overline{\oplus\{l_i(X_i <: B_i) : T\}_{i\in I}} \stackrel{\texttt{def}}{=} \&\{l_i(X_i <: B_i) : \overline{T_i}\}_{i\in I}$$
$$\overline{\&\{l_i(X_i <: B_i) : T\}_{i\in I}} \stackrel{\texttt{def}}{=} \oplus\{l_i(X_i <: B_i) : \overline{T_i}\}_{i\in I}$$

Figure 8.5: Type duality for bounded polymorphic session types

Syntax of bounded polymorphic constructs in standard π-calculus We add bounded polymorphism in the standard typed π-calculus, by following the same idea as in session types: we add constraints to the labels in variant types. We report only the new constructs or the modifications made to the syntaxes of standard π-types and π-processes introduced in Section 4.3 and Section 4.1, respectively. The syntaxes of standard π-types and π-processes become as follows.

$T_\pi ::=$	B	basic type
	$\langle l_i(X_i \leq B_i)\_T_i\rangle_{i\in I}$	bounded polymorphic variant
$B ::=$	D	data type
	X	type variable
$P_\pi ::=$	$\mathbf{case}\ v\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i\in I}$	bounded polymorphic case
$v_\pi ::=$	$l(B)\_v$	bounded polymorphic variant value

Figure 8.6: Syntax of bounded polymorphic π-constructs

Types produced by T_π include basic types, which can be data types and type variables, and a modified version of variant type, called bounded polymorphic variant. The difference with respect to the standard variant is the presence of constraints of the form $(X_i \leq B_i)$ which are added to the labels of the variant. The meaning of this constraint is the same as for session types namely, the variables X_i which

occur in T_i can be instantiated be types that respect the condition, where $\leq$ indicates the subtyping relation on π-types presented in Fig. 7.2. As long as terms are concerned, the modification of variant type triggers modifications in the **case** process and in the variant value, which now are bounded polymorphic forms of the standard ones. The bounded polymorphic case, as the variant type, has attached to the labels l_i the constraints $(X_i \leq B_i)$, whether the bounded polymorphic value, has attached to its label l a chosen basic type B. Again, the reduction rules, will give us a better understanding of how label annotations are used.

8.2.2 Semantics

In this section we introduce the reduction rules for the bounded polymorphic processes. Rule (R-BPolySel) substitutes (R-Sel) in the session processes and rule (Rπ-BPolyCase) substitutes (Rπ-Case) in the π-processes.

$$\text{(R-BPolySel)} \qquad (\nu xy)(x \triangleleft l_j(B).P \mid y \triangleright \{l_i(X_i <: B_i) : P_i\}_{i\in I} \mid R) \rightarrow (\nu xy)(P \mid P_j[B/X_j] \mid R) \quad j \in I$$

$$\text{(R}\pi\text{-BPolyCase)} \quad \textbf{case}\, l_j(B)\_v \,\textbf{of}\, \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i\in I} \rightarrow P_j[B/X_j][v/x_j] \quad j \in I$$

Rule (R-BPolySel) states that a communication occurs between a selection process $l_j(B).P$ and a branching process $y \triangleright \{l_i(X_i <: B_i) : P_i\}_{i\in I}$, whenever x and y are co-variables. In addition, together with the selected label l_j there is also a selection of basic type B. This communication reduces to P composed with the j-th process offered by branching where the corresponding type variable X_j is substituted by the selected basic type B. Rule (Rπ-BPolyCase) states that a case normalisation occurs when the guard of **case** is a variant value $l_j(B)\_v$. This reduces to the j-th process offered by the bounded polymorphic case where except the standard substitution of the placeholder x by v, also the type variable X_j is substituted by the selected basic type B. In both cases, the reduction rules succeed only if $j \in I$.

8.2.3 Typing Rules

In this section we present the typing rules for both the π-calculus with sessions and the standard π-calculus.

Typing rules for bounded polymorphic session π-calculus We start with sessions. Typing judgements are of the form $\Gamma; \Delta \vdash v : T_s$ stating that a session value

v is of bounded polymorphic session type T_s in a typing environment Γ and a set of type variables Δ, and $\Gamma; \Delta \vdash P_s$ stating that a bounded polymorphic session process is well-typed in a typing environment Γ and a set of type variables Δ.

The new typing rules for the bounded polymorphic branching and selection are given in the following:

$$\frac{\begin{array}{c}\Gamma_1; \Delta \vdash x : \oplus\{l_i(X_i <: B_i) : T_i\}_{i \in I} \\ \Gamma_2, x : T_j[B/X_j]; \Delta \vdash P \quad j \in I \\ B <: B_i \quad \forall i \in I\end{array}}{\Gamma_1 \circ \Gamma_2; \Delta \vdash x \triangleleft l_j(B).P} \text{ (T-BPolySel)}$$

$$\frac{\begin{array}{c}\Gamma_1; \Delta \vdash x : \&\{l_i(X_i <: B_i) : T_i\}_{i \in I} \\ \Gamma_2, x : T_i; \Delta, X_i <: B_i \vdash P_i \quad \forall i \in I\end{array}}{\Gamma_1 \circ \Gamma_2; \Delta \vdash x \triangleright \{l_i(X_i <: B_i) : P_i\}_{i \in I}} \text{ (T-BPolyBrch)}$$

Figure 8.7: Typing rules for bounded polymorphic session constructs

Recall that since we are dealing with session types, namely only linear pretypes, we omit from the typing rules the qualifier q and the context update operator + is replaced by , operator. However, in order to deal with linearity we use the context split operator $\circ$. Some comments on the typing rules follow. Rule (T-BPolySel) states that the selection process, where label l_j together with the basic type B is selected, is well-typed whenever channel x is of bounded polymorphic select type and $B <: B_i$ for all $i \in I$. In addition, process P is well-typed under x having the appropriate type where type variable X_j is substituted by the selected type B. Rule (T-BPolyBrch) states that the branching process is well-typed whenever channel x is of bounded polymorphic branch type and every process P_i in the branching is well-typed under the condition $X_i <: B_i$.

Typing rules for bounded polymorphic standard π- calculus We consider now the standard π- calculus. Typing judgements in the bounded polymorphic π- calculus are of the form $\Gamma; \Delta \vdash v : T_\pi$, stating that a value v is of type T_π in a typing environment Γ and a set of type variables Δ, and $\Gamma; \Delta \vdash P_\pi$, stating that the bounded polymorphic process P_π is well-typed in a typing environment Γ and a set of type variables Δ. The new typing rules are presented in Fig. 8.8. Rule (Tπ- BPolyLVal) states that the bounced polymorphic variant value $l_j(B)\_v$ is of bounded polymorphic variant type $\langle l_i(X_i \leq B_i)\_T_i\rangle_{i \in I}$ whenever $B \leq B_i$ for all i and value v is of type T_j where the corresponding type variable X_j is substituted by the

selected basic type B. Rule (Tπ-BPolyCase) states that the bounded polymorphic case is well-typed whenever the guard v is of the appropriate variant type and every process P_i is well-typed under the augmented type environment with the type assumption $x_i : T_i$ and the constraint $X_i \leq B_i$.

$$\frac{\Gamma; \Delta \vdash v : T_j[B/X_j] \quad j \in I \qquad B \leq B_i \quad \forall i \in I}{\Gamma; \Delta \vdash l_j(B)\_v : \langle l_i(X_i \leq B_i)\_T_i \rangle_{i \in I}} \text{ (T}\pi\text{-BPolyLVal)}$$

$$\frac{\Gamma_1; \Delta \vdash v : \langle l_i(X_i \leq B_i)\_T_i \rangle_{i \in I} \qquad \Gamma_2, x_i : T_i; \Delta, X_i \leq B_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \uplus \Gamma_2; \Delta \vdash \mathbf{case}\, v\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i \in I}} \text{ (T}\pi\text{-BPolyCase)}$$

Figure 8.8: Typing rules for bounded polymorphic π-constructs

8.2.4 Encoding

In this section we present the encoding of bounded polymorphic constructs: types and terms. The encoding of bounded polymorphic types is defined in Fig. 8.9.

$$\begin{aligned}
[\![B]\!] &\stackrel{\text{def}}{=} B && \text{(E-BPolyB)} \\
[\![\oplus\{l_i(X_i <: B_i) : T_i\}_{i \in I}]\!] &\stackrel{\text{def}}{=} \ell_{\text{o}}\, [\langle l_i(X_i \leq B_i)_[\![\overline{T_i}]\!] \rangle_{i \in I}] && \text{(E-BPolySel)} \\
[\![\&\{l_i(X_i <: B_i) : T_i\}_{i \in I}]\!] &\stackrel{\text{def}}{=} \ell_{\text{i}}\, [\langle l_i(X_i \leq B_i)_[\![T_i]\!] \rangle_{i \in I}] && \text{(E-BPolyBrch)}
\end{aligned}$$

Figure 8.9: Encoding of bounded polymorphic types

$$\begin{aligned}
&\text{(E-BPolySelection)} \\
&[\![x \triangleleft l_j(B).P]\!]_f \stackrel{\text{def}}{=} (\nu c) f_x!\langle l_j(B)_c \rangle.[\![P]\!]_{f,\{x \mapsto c\}} \\
&\text{(E-BPolyBranching)} \\
&[\![x \triangleright \{l_i(X_i <: B_i) : P_i\}_{i \in I}]\!]_f \stackrel{\text{def}}{=} f_x?(y).\, \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)_c \triangleright [\![P_i]\!]_{f,\{x \mapsto c\}}\}_{i \in I}
\end{aligned}$$

Figure 8.10: Encoding of bounded polymorphic terms

Equation (E-BPolyB) states that the encoding of a basic type is the identity function, namely, the encoding of a data type and of a type variable is the same data type and type variable in the standard π-calculus. Equation (E-BPolySel) states that the encoding of a bounded polymorphic select type is a linear channel type, used to output a value of type bounded polymorphic variant where subtyping constraint $X_i <: B_i$ in the select type is interpreted as the subtyping constraint $X_i \leq B_i$ in the variant type. As in (E-Select), the types in the branches of the variant type are $[\![\overline{T_i}]\!]$. Equation (E-BPolyBrch) states the dual of the previous one. The bounded polymorphic branch is interpreted as a linear input channel type. The subtyping constraints are the same and as in (E-Branch) the types in the branches of the variant type are $[\![T_i]\!]$.

Let us consider now the terms. The encoding of bounded polymorphic terms is defined in Fig. 8.10. These equations are similar to (E-Selection) and (E-Branching). The difference is the annotation of labels with types. Equation (E-BPolySelection) states that the bounded polymorphic selection is interpreted as an output with subject the renamed variable x and object a bounded polymorphic variant value, where the label and the basic type selected are the same and the value carried by the variant value is a freshly created channel c, used for the rest of the communication. The continuation process P is encoded in f updated with x renamed as c. Equation (E-BPolyBranching) states that the bounded polymorphic branching is interpreted as an input with subject the renamed x followed by a **case** process having as guard the object of the input. The branches of **case** are encoded as in (E-Branching).

The encoding of the typing environments is given by

$$[\![\Gamma; \Delta]\!]_f \stackrel{\text{def}}{=} [\![\Gamma]\!]_f; \Delta$$

and is the same as in the case of parametric polymorphism.

8.2.5 Properties of the Encoding

In this section we prove the correctness of the encoding in the extended calculi with bounded polymorphic constructs. This means that by using the encoding and the bounded polymorphism in the π-calculus, we can derive bounded polymorphism in session types.

To complete Lemma 6.3.8 of soundness and Lemma 6.3.9 of completeness of typing values via the encoding, it suffices to add the case for bounded poly-

morphic variables. However, adding this case requires modification in the typing judgements: previous typing judgements of the form $\Gamma \vdash v : T$ can be now written as $\Gamma; \Delta \vdash v : T$, where Δ can be empty in absence of polymorphism. The following lemma that states the correctness of typing variables with bounded polymorphic types by using the encoding.

Lemma 8.2.1 (Correctness: Typing Bounded Polymorphic Values)**.** *The following hold:*

- $\Gamma; \Delta \vdash x : \oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I}$ *if and only if* $[\![\Gamma; \Delta]\!]_f \vdash f_x : \ell_{\mathsf{o}}\,[\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i\in I}]$
- $\Gamma; \Delta \vdash x : \&\{l_i(X_i <: B_i) : T_i\}_{i\in I}$ *if and only if* $[\![\Gamma; \Delta]\!]_f \vdash f_x : \ell_{\mathsf{i}}\,[\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}]$

Proof. We consider both select and branch type.

- Follows immediately by equation (E-BPolySel) and rules (T-Var) and (Tπ-Var).
- Follows immediately by equation (E-BPolyBrch) and rules (T-Var) and (Tπ- Var).

□

To complete Theorem 6.3.10 of soundness and Theorem 6.3.11 of completeness of typing processes via the endocing, it suffices to add the case for bounded branching and selection. As with values, adding this case to the proofs of the previous theorems requires modification in the typing judgements: previous typing judgements of the form $\Gamma \vdash P$ can be now written as $\Gamma; \Delta \vdash P$, where Δ can be empty in absence of polymorphism. We give the two theorems of soundness and completeness in the following.

Theorem 8.2.2 (Soundness: Typing Bounded Polymorphic Processes)**.** *If* $\Gamma; \Delta \vdash Q$*, then* $[\![\Gamma; \Delta]\!]_f \vdash [\![Q]\!]_f$*, where either* $Q = x\triangleleft l_j(B).P$*, or* $Q = x\triangleright\{l_i(X_i \leq B_i) : P_i\}_{i\in I}$

Proof. The proof is done by induction on the derivation $\Gamma; \Delta \vdash Q$, by analysing the last typing rule applied.

- Case (T-BPolySel):

$$\frac{\Gamma_1; \Delta \vdash x : \oplus\{l_i(X_i \leq B_i) : T_i\}_{i \in I} \quad \Gamma_2, x : T_j[B/X_j]; \Delta \vdash P \quad j \in I \qquad B <: B_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2; \Delta \vdash x \triangleleft l_j(B).P} \text{ (T-BPolySel)}$$

To prove that $[\![\Gamma_1 \circ \Gamma_2; \Delta]\!]_f \vdash [\![x \triangleleft l_j(B).P]\!]_f$. By (E-BPolySelection) and Lemma 6.3.5 and the encoding of type environments, it means that $[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \Delta \vdash (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}$. By Lemma 8.2.1 $[\![\Gamma_1]\!]_{f'}; \Delta \vdash f'_x : \ell_{\mathsf{o}}\ [\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}]$. By induction hypothesis and by Lemma 5.6 and by Lemma 8.1.1 we have that $[\![\Gamma_2]\!]_{f''}, f''_x : [\![T_j]\!][B/X_j]; \Delta \vdash [\![P]\!]_{f''}\ j \in I$. By induction hypothesis and by Lemma 7.2.2 $B \leq B_i\ \forall i \in I$. Let $f''_x = c$ and $f = f' \cup f'' - \{x \mapsto c\}$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Then, by rewriting the previous judgements we have $[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_{\mathsf{o}}\ [\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}]$, and $[\![\Gamma_2]\!]_f, c : [\![T_j]\!][B/X_j]; \Delta \vdash [\![P]\!]_{f,\{x \mapsto c\}}\ j \in I$. By applying (T$\pi$-Var) in order to derive $c : [\![\overline{T_j}]\!]$, rule (T$\pi$-BPolyLVal) and by soundness of subtyping we have the following:

$$\frac{\dfrac{}{c : [\![\overline{T_j}]\!][B/X_j]; \Delta \vdash c : [\![\overline{T_j}]\!][B/X_j]}\ (\text{T}\pi\text{-Var}) \quad j \in I \qquad B \leq B_i \quad \forall i \in I}{c : [\![\overline{T_j}]\!][B/X_j]; \Delta \vdash l_j(B)\_c : \langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}} \text{ (T}\pi\text{-BPolyLVal)}$$

Then, by applying rule (Tπ-Out) on the former:

$$\frac{[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_{\mathsf{o}}\ [\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}] \quad c : [\![\overline{T_j}]\!][B/X_j]; \Delta \vdash l_j(B)\_c : \langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I} \quad [\![\Gamma_2]\!]_f, c : [\![T_j]\!][B/X_j]; \Delta \vdash [\![P]\!]_{f,\{x \mapsto c\}}\ j \in I}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus c : \ell_\sharp[W][B/X_j]; \Delta \vdash f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}}$$

Notice that, we use Lemma 6.3.7 and Lemma 6.3.4 in order to obtain $c : \ell_\sharp[W][B/X_j]$, where $[\![T_j]\!] = \ell_\alpha[W]$ and $[\![\overline{T_j}]\!] = \ell_{\overline{\alpha}}[W]$. We conclude by applying (Tπ-Res1):

$$\frac{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus c : \ell_\sharp[W][B/X_j]; \Delta \vdash f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \Delta \vdash (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}}$$

- Case (T-BPolyBrch):

$$\frac{\Gamma_1;\Delta \vdash x : \&\{l_i(X_i \leq B_i) : T_i\}_{i\in I} \quad \Gamma_2, x : T_i;\Delta, X_i <: B_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2;\Delta \vdash x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}}\ \text{(T-BPolyBrch)}$$

To prove that $[\![\Gamma_1 \circ \Gamma_2;\Delta]\!]_f \vdash [\![x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}]\!]_f$. By equation (E-BPolyBranching) and Lemma 6.3.5 it means that $[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f;\Delta \vdash f_x?(y).\ \mathbf{case}\ y\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$. By Lemma 8.2.1 $[\![\Gamma_1;\Delta]\!]_{f'} \vdash f'_x : \ell_{\mathrm{i}}\ [\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}]$. By induction hypothesis and by Lemma 6.3.1 and by Lemma 7.2.2 we have that $[\![\Gamma_2]\!]_{f''}, f''_x : [\![T_i]\!];\Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f''}\ \forall i \in I$. Let $f''_x = c$ and $f = f' \cup f'' - \{x \mapsto c\}$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Then, by rewriting the previous typing judgements we have $[\![\Gamma_1]\!]_f;\Delta \vdash f_x : \ell_{\mathrm{i}}\ [\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}]$, and $[\![\Gamma_2]\!]_f, c : [\![T_i]\!];\Delta, X_i \leq B_i \vdash [\![P_i]\!]_f\ \forall i \in I$. Then, by applying (Tπ-Var) in order to derive $y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}$ and (Tπ-BPolyCase) we have:

$$\frac{y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I};\Delta \vdash y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I} \quad [\![\Gamma_2]\!]_f, c : [\![T_i]\!];\Delta, X_i \leq B_i \vdash [\![P_i]\!]_f \quad \forall i \in I}{[\![\Gamma_2]\!]_f, y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I};\Delta \vdash \mathbf{case}\ y\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

Then, by applying (Tπ-Inp) we have:

$$\frac{[\![\Gamma_1]\!]_f;\Delta \vdash f_x : \ell_{\mathrm{i}}\ [\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}] \quad [\![\Gamma_2]\!]_f, y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I};\Delta \vdash \mathbf{case}\ y\ \mathbf{of}\ \{l_i\_c(X_i \leq B_i) \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f;\Delta \vdash f_x?(y).\ \mathbf{case}\ y\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

□

Theorem 8.2.3 (Completeness: Typing Bounded Polymorphic Processes)**.** *If* $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$, *then* $\Gamma;\Delta \vdash Q$, *where either* $Q = x \triangleleft l_j(B).P$, *or* $Q = x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}$.

Proof. The proof is done by induction on the structure of Q.

- Case $Q = x \triangleleft l_j(B).P$:
  By equation (E-BPolySelection) we have $[\![x \triangleleft l_j(B).P]\!]_f \stackrel{\mathtt{def}}{=} (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f;\Delta \vdash (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$.

By rules (Tπ-Res1), since c is present in $[\![P]\!]_{f,\{x\mapsto c\}}$, and (T$\pi$-Out) we have:

$$\dfrac{\begin{array}{c}(\text{T}\pi\text{-Res1})\\(\text{T}\pi\text{-Out})\\ \Gamma_1^\pi;\Delta \vdash f_x : \ell_{\mathsf{o}}\,[\langle l_i(X_i \leq B_i)\_[\![T_i^\pi]\!]\rangle_{i\in I}] \qquad \Gamma_2^\pi, c : \overline{T_j^\pi}[B/X_j];\Delta \vdash [\![P]\!]_{f,\{x\mapsto c\}} \\ c : T_j^\pi[B/X_j];\Delta \vdash l_j(B)\_c : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}\end{array}}{\dfrac{[\![\Gamma]\!]_f, c : \ell_\sharp\,[W][B/X_j];\Delta \vdash f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Gamma]\!]_f;\Delta \vdash (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}}$$

where by (Tπ-BPolyLval) we have:

$$\dfrac{\dfrac{}{c : T_j^\pi[B/X_j];\Delta \vdash c : T_j^\pi[B/X_j]}\,(\text{T}\pi\text{-Var}) \qquad j \in I \\ B \leq B_i \quad \forall i \in I}{c : T_j^\pi[B/X_j];\Delta \vdash l_j(B)\_c : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}}\,(\text{T}\pi\text{-BPolyLVal})$$

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. Notice that c is a channel and one capability of c is sent along $l_j(B)\_c$ whether the other is used in the continuation $[\![P]\!]_{f,\{x\mapsto c\}}$, hence $\ell_\sharp\,[W] = T_j^\pi \uplus \overline{T_j^\pi}$. By Lemma 8.2.1 we have $\Gamma_1;\Delta \vdash x : \oplus\{l_i(X_i \leq B_i) : T_i\}_{i\in I}$ where by (E-BPolySel) $\ell_{\mathsf{o}}\,[\langle l_i(X_i \leq B_i)\_[\![T_i^\pi]\!]\rangle_{i\in I}] = [\![\oplus\{l_i(X_i \leq B_i) : T_i\}_{i\in I}]\!]$ and $T_i^\pi = [\![\overline{T_i}]\!]$ $\forall i \in I$. By induction hypothesis $\Gamma_2, x : T_j[B/X_j];\Delta \vdash P$. By Theorem 7.2.3 we obtain $B <: B_i$ $\forall i \in I$. By applying rule (T-BPolySel) we obtain $\Gamma_1 \circ \Gamma_2;\Delta \vdash x \triangleleft l_j(B).P$, as follows.

$$\dfrac{\begin{array}{c}\Gamma_1;\Delta \vdash x : \oplus\{l_i(X_i \leq B_i) : T_i\}_{i\in I} \\ \Gamma_2, x : T_j[B/X_j];\Delta \vdash P \quad j \in I \\ B <: B_i \quad \forall i \in I\end{array}}{\Gamma_1 \circ \Gamma_2;\Delta \vdash x \triangleleft l_j(B).P}\,(\text{T-BPolySel})$$

- Case $Q = x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}$:
  By equation (E-BPolyBranching) we have $[\![x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}]\!]_f \stackrel{\text{def}}{=} f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$ and assume $[\![\Gamma]\!]_f;\Delta \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$ which by rules (Tπ-Inp) and

(Tπ-BPolyCase) and Lemma 6.3.3 means that

$$\dfrac{\Gamma_1^\pi;\Delta \vdash f_x : \ell_{\mathrm{i}}\,[\langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}] \quad \dfrac{\begin{array}{c}\text{(T}\pi\text{-BPolyCase)}\\ y : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I};\Delta \vdash \\ y : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I} \\ \Gamma_2^\pi, c : T_i^\pi;\Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}\ \forall i \in I\end{array}}{\begin{array}{c}\Gamma_2^\pi, y : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I};\Delta \vdash \\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}\end{array}}}{[\![\Gamma]\!]_f;\Delta \vdash f_x?(y).\, \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}\ \text{(T}\pi\text{-Inp)}$$

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 8.2.1 we have $\Gamma_1;\Delta \vdash x : \&\{l_i(X_i \leq B_i) : T_i\}_{i\in I}$ where $[\![\&\{l_i(X_i \leq B_i) : T_i\}_{i\in I}]\!]_f = \ell_{\mathrm{i}}\,[\langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}]$ by applying (E-BPolyBrch) and hence $\forall i \in I\ [\![T_i]\!] = T_i^\pi$. By induction hypothesis we have $\Gamma_2, x : T_i;\Delta, X_i <: B_i \vdash P_i\ \forall i \in I$ where $f(x) = c$. By rule (T-BPolyBrch) we obtain $\Gamma_1 \circ \Gamma_2;\Delta \vdash x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}$, as follows.

$$\dfrac{\begin{array}{l}\Gamma_1;\Delta \vdash x : \&\{l_i(X_i \leq B_i) : T_i\}_{i\in I} \\ \Gamma_2, x : T_i;\Delta, X_i <: B_i \vdash P_i \quad \forall i \in I\end{array}}{\Gamma_1 \circ \Gamma_2;\Delta \vdash x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}}\ \text{(T-BPolyBrch)}$$

□

In the following, we prove the operational correspondence for bounded polymorphic constructs, which is a case added to the proof of the main operational correspondence result, namely Theorem 6.3.14.

Theorem 8.2.4 (OC for Bounded Polymorphism)**.** *Let P be a bounded polymorphic process in the π-calculus with sessions. Then,*

1. *If $P \to P'$ then, $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$;*
2. *If $[\![P]\!]_f \to\!\!\equiv Q$ then, $\exists\, P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \to^*\!\!\equiv [\![P']\!]_{f'}$, where f' is the updated f after the communication and $f_x = f_y$ for all $(\nu xy) \in \mathcal{E}[\cdot]$.*

Proof. We consider both cases in the following.

1. The proof is done by induction on the length of the derivation $P \to P'$ and there is just one case to consider, rule (R-BPolySel):

   $$(\nu xy)(x \triangleleft l_j(B).Q \mid y \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I} \mid R) \to (\nu xy)(Q \mid P_j[B/X_j] \mid R)\ j \in I$$

   By the encoding of bounded polymorphic processes we have

   $$\begin{aligned}
   &[\![P]\!]_f \\
   &\quad = \quad [\![(\nu xy)(x \triangleleft l_j(B).Q \mid y \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I} \mid R)]\!]_f \\
   &\quad \stackrel{\text{def}}{=} \quad (\nu c)\,([\![x \triangleleft l_j(B).Q]\!]_{f,\{x\mapsto c\}} \mid [\![y \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}]\!]_{f,\{y\mapsto c\}} \mid [\![R]\!]_f) \\
   &\quad \stackrel{\text{def}}{=} \quad (\nu c)\,[(\nu c')(c!\langle l_j(B)_c'\rangle.[\![Q]\!]_{f,\{x\mapsto c, c\mapsto c'\}}) \mid \\
   &\qquad\qquad c?(z).\ \mathbf{case}\ z\ \mathbf{of}\ \{l_i(X_i \leq B_i)_c' \triangleright [\![P_i]\!]_{f,\{y\mapsto c, c\mapsto c'\}}\}_{i\in I} \mid [\![R]\!]_f] \\
   &\quad \to \quad (\nu c)\,[(\nu c')([\![Q]\!]_{f,\{x\mapsto c, c\mapsto c'\}} \mid \\
   &\qquad\qquad \mathbf{case}\ l_j(B)_c'\ \mathbf{of}\ \{l_i(X_i \leq B_i)_c' \triangleright [\![P_i]\!]_{f,\{y\mapsto c, c\mapsto c'\}}\}_{i\in I} \mid [\![R]\!]_f)] \\
   &\quad \to \quad (\nu c)\,[(\nu c')([\![Q]\!]_{f,\{x\mapsto c, c\mapsto c'\}} \mid \\
   &\qquad\qquad [\![P_j]\!]_{f,\{y\mapsto c, c\mapsto c'\}}[B/X_j] \mid [\![R]\!]_f)] \\
   &\quad \equiv \quad (\nu c)\,[(\nu c')([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid \\
   &\qquad\qquad [\![P_j]\!]_{f,\{x,y\mapsto c'\}}[B/X_j] \mid [\![R]\!]_{f,\{x,y\mapsto c'\}})] \\
   &\quad = \quad (\nu c')([\![Q]\!]_{f'} \mid [\![P_j]\!]_{f'}[B/X_j] \mid [\![R]\!]_{f'})
   \end{aligned}$$

   Where $f' = f, \{x \mapsto c', y \mapsto c'\}$. On the other hand

   $$\begin{aligned}
   &[\![P']\!]_f \\
   &\quad = \quad [\![(\nu xy)(Q \mid P_j[B/X_j] \mid R)]\!]_f \\
   &\quad \stackrel{\text{def}}{=} \quad (\nu c')([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}}[B/X_j] \mid [\![R]\!]_{f,\{x,y\mapsto c'\}}) \\
   &\quad \stackrel{\text{def}}{=} \quad (\nu c')([\![Q]\!]_{f'} \mid [\![P_j]\!]_{f'}[B/X_j] \mid [\![R]\!]_{f'})
   \end{aligned}$$

   Where we have used Lemma 8.1.1 and equation (E-BPolyB). Which means

   $$[\![P]\!]_f \to\!\hookrightarrow [\![P']\!]_f$$

2. The proof is done by induction on the structure of the bounded polymorphic session process P. The only case to consider is added to case 2c

in Theorem 6.3.14 Let $P_1 \mid P_2 = x \triangleleft l_j(B).P'_1 \mid y \triangleright \{l_i(X_i \leq B_i) : P''_i\}_{i\in I}$. By the encodings, equations (E-Composition), (E-BPolySelection) and (E-BPolyBranching), we have

$$\begin{aligned}
&[\![P_1]\!]_f \mid [\![P_2]\!]_f \\
&\qquad \stackrel{\text{def}}{=} \ (\nu c) f_x!\langle l_j(B)_c\rangle.[\![P'_1]\!]_{f,\{x\mapsto c\}} \mid \\
&\qquad\qquad f_y?(z).\ \mathbf{case}\ z\ \mathbf{of}\ \{l_i(X_i \leq B_i)_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I} \\
&\qquad \to \ [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid \mathbf{case}\ l_j(B)_c\ \mathbf{of}\ \{l_i(X_i \leq B_i)_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I} \\
&\qquad \to \ [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid [\![P''_j]\!]_{f,\{y\mapsto c\}}[B/X_j] \equiv Q
\end{aligned}$$

We now show that there exist $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P_1 \mid P_2] \to \mathcal{E}[P']$ and $Q \to^* \equiv [\![P']\!]_{f'}$. Let $\mathcal{E}[\cdot] = (\nu xy)[\cdot]$, then by (R-BPolySel) we have

$$\begin{aligned}
\mathcal{E}[P_1 \mid P_2] \ &= \\
&\qquad (\nu xy)\left(x \triangleleft l_j(B).P'_1 \mid y \triangleright \{l_i(X_i \leq B_i) : P''_i\}_{i\in I}\right) \\
&\qquad \to \ (\nu xy)(P'_1 \mid P''_j[B/X_j]) \\
&\qquad = \ \mathcal{E}[P']
\end{aligned}$$

Trivially, by (E-Composition) and by Lemma 8.1.1 we have $[\![P']\!]_{f'} = [\![P'_1 \mid P''_j[B/X_j]]\!]_{f'} \stackrel{\text{def}}{=} [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid [\![P''_j]\!]_{f,\{y\mapsto c\}}[B/X_j] \equiv Q$ where we let $f' = f \cup \{x \mapsto c, y \mapsto c\}$.

□

CHAPTER 9

Higher-Order

Higher-order π-calculus (HOπ) models mobility of processes that can be sent and received and thus can be run locally [101]. Higher-order in the π-calculus with sessions has the same benefits as in the standard π-calculus, in particular, it models code mobility in a distributed scenario. In this chapter, we use HOπ to provide sessions with higher-order by extending the encoding, in the same way as with subtyping and polymorphism, in order to accommodate higher-order constructs. Let us now consider higher-order sessions [90] and higher-order π-calculus.

9.1 Syntax

In this section we present the syntax of types and terms for both π-calculus with and without sessions. Since the type constructs and the term constructs for higher-order, added to both calculi are the same, we present them with the same grammar. We will distinguish them by the context in which they are used and in particular, we will often refer to the standard π-calculus constructs as the encoded constructs of the π-calculus with sessions.

The syntax of types and terms of HOπ with sessions and standard HOπ is given in Fig. 9.1. Let $\diamond$ denote the type of a process and let σ range over a general type T in the π-calculus with and without sessions, and on the type of processes $\diamond$. We add to the syntax of types T the type `Unit`, the functional type $T \to \sigma$, assigned to

$$
\begin{array}{rlll}
\sigma ::= & T & & \text{general type} \\
& \Diamond & & \text{process type} \\
\\
T ::= & \texttt{Unit} & & \text{unit type} \\
& T \to \sigma & & \text{functional type} \\
& T \overset{1}{\to} \sigma & & \text{linear functional type} \\
\\
P ::= & PQ & & \text{application} \\
& v & & \text{values} \\
\\
v ::= & \lambda x : T.P & & \text{abstraction} \\
& \star & & \text{unit value}
\end{array}
$$

Figure 9.1: Syntax of higher-order constructs

a functional term that can be used without any restriction and the linear functional type $T \overset{1}{\to} \sigma$, assigned to a term that should be used *exactly once*. The reason for the linear functional type is privacy and communication safety properties that we want to be guaranteed in the session types. In particular a function may contain free session channels, hence it should necessarily be used at least once, in order to complete the session and so to ensure communication safety and on the other hand it should not be used more than once, so not to violate privacy. As long as terms are concerned, they include constructs borrowed from the λ-calculus the *abstraction* and the *application*, to enable mobility not only of values but also of processes. A process can be the application PQ of a process P, typically being a functional value, to a process Q, or a value v. The latter can be an abstraction $\lambda x : T.P$ having exactly the same meaning as in λ-calculus, where variable x is bound with scope P, or a unit value $\star$ typically having `Unit` type. Note that the above type and term constructs where first added to the standard π-calculus and then they were also adopted in the π-calculus with sessions.

9.2 Semantics

In this section we present the new reduction rules added to the existing ones presented in Section 5.2 for sessions and in Section 4.2 for standard π-calculus, respectively.

The new reduction rules are given in Fig. 9.2. We will distinguish the rules

$$\text{(R}[\pi]\text{-Beta)} \qquad (\lambda x : T.P)v \to P[v/x]$$

$$\text{(R}[\pi]\text{-ApplLeft)} \qquad \frac{P \to P'}{PQ \to P'Q}$$

$$\text{(R}[\pi]\text{-ApplRight)} \qquad \frac{P \to P'}{vP \to vP'}$$

Figure 9.2: Semantics of higher-order constructs

for sessions from the ones for standard π-calculus by the presence of $[\pi]$ in the rule name. Rule (R$[\pi]$-Beta) states that the application of a λ-abstraction $\lambda x : T.P$ on a value v reduces to P where x is substituted by v. Rules (R$[\pi]$-ApplLeft) and (R$[\pi]$-ApplRight) state that the application process reduces if one of its parts reduces as well.

9.3 Typing Rules

In this section we present the typing rules for the HOπ with and without sessions. Typing judgements are of the form $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$. For simplicity, in case P is a process and not a value, we use $\Phi; \Gamma; \mathcal{S} \vdash P$ instead $\Phi; \Gamma; \mathcal{S} \vdash P : \diamond$. We start with session typing.

9.3.1 HOπ Session Typing

The typing contexts are defined as follows:

$\Phi ::=$	$\emptyset \mid \Phi, x : \texttt{Bool} \mid \Phi, x : \texttt{Unit}$		
	$\Phi, x : T \to \sigma \mid \Phi, x : T \overset{1}{\to} \sigma$		general type environment
$\Gamma ::=$	$\emptyset \mid \Gamma, x : \mathsf{lin} p \mid \Gamma, x : \mathsf{end}$		session type environment
$\mathcal{S} ::=$	$\emptyset \mid \mathcal{S} \cup \{x\}$		linear functional variables

Φ associates value types, except session types, to identifiers. Γ associates linear pretypes or terminated channel types, namely session types, to channels. $\mathcal{S}$ denotes the set of linear functional variables. The context split $\circ$ is defined as in Fig. 5.6. We state that a typing judgement is well-formed if $\mathcal{S} \subseteq \mathsf{dom}(\Phi)$ and $\mathsf{dom}(\Phi) \cap \mathsf{dom}(\Gamma) = \emptyset$.

$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma,x:T;\emptyset\vdash x:T}\ (\text{T-HoSess}) \qquad \frac{T\neq T'\overset{1}{\to}\sigma \qquad \mathsf{un}(\Gamma)}{\Phi,x:T;\Gamma;\emptyset\vdash x:T}\ (\text{T-HoVar})$$

$$\frac{\mathsf{un}(\Gamma)\quad v=\texttt{true}\ /\ \texttt{false}}{\Phi;\Gamma;\emptyset\vdash v:\texttt{Bool}}\ (\text{T-HoBool}) \qquad \frac{\mathsf{un}(\Gamma)}{\Phi,x:T\overset{1}{\to}\sigma;\Gamma;\{x\}\vdash x:T\overset{1}{\to}\sigma}\ (\text{T-HoFun})$$

$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset\vdash\star:\texttt{Unit}}\ (\text{T-HoUnit}) \qquad \frac{\Phi,x:T;\Gamma;\mathcal{S}\vdash P:\sigma \quad \text{if } T=T'\overset{1}{\to}\sigma \text{ then } x\in\mathcal{S}}{\Phi;\Gamma;\mathcal{S}-\{x\}\vdash\lambda x:T.P:T\to\sigma}\ (\text{T-HoAbs1})$$

$$\frac{\Phi;\Gamma,x:T;\mathcal{S}\vdash P:\sigma}{\Phi;\Gamma;\mathcal{S}\vdash\lambda x:T.P:T\to\sigma}\ (\text{T-HoAbs2}) \qquad \frac{\Phi;\Gamma;\mathcal{S}\vdash P:T\to\sigma}{\Phi;\Gamma;\mathcal{S}\vdash P:T\overset{1}{\to}\sigma}\ (\text{T-HoSub})$$

$$\frac{\Phi;\Gamma_1;\mathcal{S}_1\vdash P:T\overset{1}{\to}\sigma \qquad \Phi;\Gamma_2;\mathcal{S}_2\vdash Q:T \quad \text{if } T=T'\to\sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2=\emptyset}{\Phi;\Gamma_1\circ\Gamma_2;\mathcal{S}_1\cup\mathcal{S}_2\vdash PQ:\sigma}\ (\text{T-HoApp})$$

Figure 9.3: Typing rules for the HOπ with sessions: values and functions

The predicates lin and un are defined as in Section 5.3. However, since we use only linear pretypes, this means that the only unrestricted types are the ground types, like `Bool`, `Unit` ... and the terminated channel type `end`.

The typing rules for the HOπ with sessions are given in Fig. 9.3 and Fig. 9.4.

We start with the typing rules for values presented in Fig. 9.3. Rule (T-HoSess) states that a variable x has session type T, if this is assumed in the type environment for channels. Rule (T-HoVar) states that a variable has not linear type, namely different from session type and different from linear functional type, if it is assumed so in the first type environment. On the contrary, rule (T-HoFun) states that a variable is of a linear functional type if this is assumed in the first type environment and x is put in the set $\mathcal{S}$. Rule (T-HoBool) states that a boolean value `true` or `false`, is of type `Bool` where Γ is unrestricted and $\mathcal{S}=\emptyset$. Rule (T-HoUnit) is similar to (T-HoBool). There are two typing rules for abstractions,

depending on the type of x being the binder in the λ-abstraction. Rule (T-HoAbs1) states that $\lambda x : T.P$ is of type $T \to \sigma$ if process P is of type σ in x having value type. In case x is a linear functional variable, then it is in $\mathcal{S}$ and since λ is a binder, then in typing $\lambda x : T.P$ we remove x from $\mathcal{S}$. Rule (T-HoAbs2) is similar to the previous one, but in this case x is a session type and is in the second type environment. Rule (T-HoSub) is a subsumption typing rule. It states that an arrow type can be lifted to a linear arrow type. Rule (T-HoApp) states that the application of process P to Q has type σ if P is a linear function of type $T \xrightarrow{1} \sigma$ and Q is of type T. Notice that in case P is of type $T \to \sigma$, then it can be lifted to the linear one by subsumption. In case the type of Q is a standard arrow type, then Q does not have any session channel, given by condition $\mathsf{un}(\Gamma_2)$, or any linear functional variables, given by condition $\mathcal{S}_2 = \emptyset$, otherwise this would violate linearity.

We comment now on the typing rules for processes given in Fig. 9.4. As previously stated, for simplicity, we omit the process type $\diamond$ from the typing judgements. Namely, $\Phi; \Gamma; \mathcal{S} \vdash P$ is a short form of $\Phi; \Gamma; \mathcal{S} \vdash P : \diamond$. Rule (T-Inact) states that the terminated process is well-typed in a typing environment where neither session type nor linear functional type assumptions are present. Rule (T-HoPar) is straightforward, it uses context split $\circ$ and union of sets $\cup$ of linear functional variables. Rules (T-HoRes) and (T-HoIf) are similar to (T-Res) and (T-If), except for the separation of type environments in three parts. There are two typing rules for the input process, depending on the type of the object of the input prefix. Rule (T-HoInp1) is similar to (T-Inp) where the type of the bound variable y is a session type. Rule (T-HoInp2) states the well-typedness of the input process where y is of value type, in particular it can be of a functional type. In case y is a linear functional variable, then it is in set $\mathcal{S}$ and when typing the input process, we remove y from $\mathcal{S}$ since the input prefix is a binder. Rule (T-HoOut) states the well-typedness of the output process by using the context split operator and the union of sets of linear functional variables present in v and P. This rule is used both when a session channel is sent, and in that case it can be read as (T-Out), or when a value, in particular functional value, is sent. In the latter case, if the value v is of a standard functional type, then it does not contain either free session channels or linear functional variable. This condition is the same as for (T-HoApp). Rules (T-HoBrch) and (T-HoSel) are the same as the standard ones, the only difference is in the type environments which are split in three parts.

9.3.2 HOπ Typing

The typing contexts are defined as follows:

$$
\begin{array}{lll}
\Phi ::= & \emptyset \mid \Phi, x : \texttt{Bool} \mid \Phi, x : \texttt{Unit} & \\
& \Phi, x : \langle l_i \_ T_i \rangle_{i \in I} \mid \Phi, x : T \to \sigma & \\
& \Phi, x : T \overset{1}{\to} \sigma & \text{general type environment} \\
\Gamma ::= & \emptyset \mid \Gamma, x : \tau & \text{channel type environment} \\
\mathcal{S} ::= & \emptyset \mid \mathcal{S} \cup \{x\} & \text{linear functional variables}
\end{array}
$$

Φ associates value types, except channel types, namely τ types, to identifiers. Γ associates τ types to channels. $\mathcal{S}$ denotes the set of linear functional variables. The $\uplus$ operation is defined as in Fig. 4.5. We state that a typing judgement is well-formed if $\mathcal{S} \subseteq \mathsf{dom}(\Phi)$ and $\mathsf{dom}(\Phi) \cap \mathsf{dom}(\Gamma) = \emptyset$. The predicates lin and un are defined as in Section 4.3. However, since we use only linear channel types, this means that the only unrestricted types are the ground types, like $\texttt{Bool}, \texttt{Unit} \ldots$ and the type of a channel with no capabilities $\emptyset[]$.

Typing rules in the standard HOπ are given in Fig. 9.5 and Fig. 9.6, for values and processes, respectively. Most of the rules follow the same line as the corresponding ones in HOπ with sessions. In the following we comment only on the typing rules that are new or different with respect to the ones previously presented. Rule (Tπ- HoLVal) is the same as (Tπ- LVal), the only difference is the type environments which are composed by three parts. There are two typing rules for restriction, as in the standard π-calculus typing rules presented in Section 4.4. Rule (Tπ- HoCase) is similar to (Tπ- Case). In addition, it uses a set of linear functional variables that comes from the union of the linear ones in the guard v and the linear ones in P_i. In the same way as for the branching process, the set of linear variables for every P_i is the same set $\mathcal{S}_2$ since only one of the processes will be executed.

9.4 Encoding

The encoding of HOπ types is an homomorphism and is given in Fig. 9.7.
We move now to the encoding of terms. As long as the new terms construct are concerned, the encoding is an homomorphism, namely Star, abstraction and application in session π-calculus are encoded respectively as star, abstraction and

application in standard π- calculus. However, since values contain processes – the case of abstraction – then we have to encode also the transmitted values. In particular, we change the encoding of the output process $x!\langle v\rangle.P$, where instead of sending v, we send the encoding f_v. Notice that this is a difference wrt to the original encoding of the output process, given in Fig 6.2.
Last thing we give the encoding of typing contexts, which is defined as follows:

Definition 9.4.1 (Encoding of type environments)**.**

$$[\![\Phi;\Gamma;\mathcal{S}]\!]_f \stackrel{def}{=} [\![\Phi]\!];[\![\Gamma]\!]_f;\mathcal{S}$$

where

$$[\![\emptyset]\!]_f \stackrel{def}{=} \emptyset$$
$$[\![\Phi, x:T]\!]_f \stackrel{def}{=} [\![\Phi]\!], x:[\![T]\!]$$
$$[\![\Gamma, x:T]\!]_f \stackrel{def}{=} [\![\Gamma]\!]_f \uplus f_x:[\![T]\!]$$

9.5 Properties of the Encoding

In this section we present the correctness of the extended encoding to HOπ constructs with respect to typing derivations for values and processes. However, since processes include also values, we present the result under the same theorem statement, without splitting in two separate theorems as we did so far. We prove the soundness of typing derivations for processes, namely if a session HOπ process has type σ, then the encoding of the session process has type the encoding of σ. We prove the completeness of typing derivations for processes, namely the opposite implication of the former also holds.

9.5.1 Typing HOπ Processes by Encoding

Before introducing the main results, we give the following auxiliary lemma that is a weakening lemma. Notice that, since Γ contains only channel type assumptions, and channel types we deal with are only linear, and $\mathcal{S}$ is the set of linear functional variables, then the only weakening that can be done is on Φ.

Lemma 9.5.1 (Weakening in HOπ)**.** *If* $\Phi;\Gamma;\mathcal{S} \vdash P$*, then* $\Phi';\Gamma;\mathcal{S} \vdash P$*, where* $\Phi' \supseteq \Phi$.

We are ready now to present the main contribution, namely the correctness of the encoding on higher-order constructs. We start with soundness.

Theorem 9.5.2 (Soundness: Typing HOπ Processes)**.** *If* $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$*, then* $[\![\Phi; \Gamma; \mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$.

Proof. The proof is done by induction on the derivation $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$, by analysing the last typing rule applied. In all the cases we use Definition 9.4.1 for the encoding of higher-order type environments.

- Cases (T-HoSess), (T-HoVar), (T-HoUnit) and (T-HoSub) follow immediately by the encoding of terms. These cases are trivial since the encoding is an homomorphism and the corresponding typing rules in the standard HOπ are the same. Cases (T-HoBool), (T-HoInact), (T-HoPar), (T-HoRes), (T-HoIf) follow exactly the same line as the corresponding ones in Theorem 6.3.10.

- Case (T-HoFun):

$$\frac{\mathsf{un}(\Gamma)}{\Phi, x : T \overset{1}{\to} \sigma; \Gamma; \{x\} \vdash x : T \overset{1}{\to} \sigma}$$

  To prove that $[\![\Phi]\!], x : [\![T \overset{1}{\to} \sigma]\!]; [\![\Gamma]\!]_f; \{x\} \vdash x : [\![T \overset{1}{\to} \sigma]\!]$. Since $\mathsf{un}(\Gamma)$, then also $\mathsf{un}([\![\Gamma]\!]_f)$. By (E-LinFunType) and by rule (Tπ-HoFun) we conclude the case.

- Case (T-HoAbs1):

$$\frac{\begin{array}{c}\Phi, x : T; \Gamma; \mathcal{S} \vdash P : \sigma \\ \text{if } T = T' \overset{1}{\to} \sigma \text{ then } x \in \mathcal{S}\end{array}}{\Phi; \Gamma; \mathcal{S} - \{x\} \vdash \lambda x : T.P : T \to \sigma}$$

  To prove $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} - \{x\} \vdash [\![\lambda x : T.P]\!]_f : [\![T \to \sigma]\!]$. By (E-Abstraction) and (E-FunType), it means that $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} - \{x\} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]$. By induction hypothesis we have $[\![\Phi]\!], x : [\![T]\!]; [\![\Gamma]\!]_{f'}; \mathcal{S} \vdash [\![P]\!]_{f'} : [\![\sigma]\!]$ and if $[\![T]\!] = [\![T' \overset{1}{\to} \sigma]\!]$ then $x \in \mathcal{S}$. Then, by letting $f = f'$ and by applying (Tπ-HoAbs1) we obtain $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} - \{x\} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : T \to \sigma$.

- Case (T-HoAbs2):

$$\frac{\Phi; \Gamma, x : T; \mathcal{S} \vdash P : \sigma}{\Phi; \Gamma; \mathcal{S} \vdash \lambda x : T.P : T \to \sigma}$$

To prove $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash [\![\lambda x : T.P]\!]_f : [\![T \to \sigma]\!]$. By (E-Abstraction) and (E-FunType), it means that $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]$. By induction hypothesis we have $[\![\Phi]\!]; [\![\Gamma]\!]_{f'}, f'_x : [\![T]\!]; \mathcal{S} \vdash [\![P]\!]_{f'} : [\![\sigma]\!]$, where $\mathsf{dom}(f') = \mathsf{dom}(\Gamma) \cup \{x\}$. Then, choose $f = f' - \{x \mapsto f'_x\}$. Then, by applying (T$\pi$-HoAbs2) we obtain $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]$.

- Case (T-HoApp):

$$\frac{\Phi; \Gamma_1; \mathcal{S}_1 \vdash P : T \xrightarrow{1} \sigma \qquad \Phi; \Gamma_2; \mathcal{S}_2 \vdash Q : T \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S}_1 \cup \mathcal{S}_2 \vdash PQ : \sigma}$$

To prove that $[\![\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S}_1 \cup \mathcal{S}_2]\!]_f \vdash [\![PQ]\!]_f : [\![\sigma]\!]$. By (E-Application) and Lemma 6.3.5 it means that $[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \mathcal{S}_1 \cup \mathcal{S}_2 \vdash [\![P]\!]_f [\![Q]\!]_f : [\![\sigma]\!]$. By induction hypothesis we have $[\![\Phi]\!]; [\![\Gamma_1]\!]_{f'}; \mathcal{S}_1 \vdash [\![P]\!]_{f'} : [\![T]\!] \xrightarrow{1} [\![\sigma]\!]$ and $[\![\Phi]\!]; [\![\Gamma_2]\!]_{f''}; \mathcal{S}_2 \vdash [\![Q]\!]_{f''} : [\![T]\!]$, and if $[\![T]\!] = [\![T' \to \sigma']\!]$ then $\mathsf{un}([\![\Gamma_2]\!]_f)$ and $\mathcal{S}_2 = \emptyset$. Now let $f = f' \cup f''$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Then, the induction hypothesis becomes $[\![\Phi]\!]; [\![\Gamma_1]\!]_f; \mathcal{S}_1 \vdash [\![P]\!]_f : [\![T]\!] \xrightarrow{1} [\![\sigma]\!]$ and $[\![\Phi]\!]; [\![\Gamma_2]\!]_f; \mathcal{S}_2 \vdash [\![Q]\!]_f : [\![T]\!]$. Then, by applying (T$\pi$-HoApp) we conclude the case.

- Case (T-HoInp1):

$$\frac{\Phi; \Gamma_1; \emptyset \vdash x : ?T.U \qquad \Phi; \Gamma_2, x : U, y : T; \mathcal{S} \vdash P}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} \vdash x?(y).P}$$

To prove that $[\![\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S}]\!]_f \vdash [\![x?(y).P]\!]_f$ for some function f such that $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. By induction hypothesis $[\![\Phi]\!]; [\![\Gamma_1]\!]_{f'}; \emptyset \vdash f'_x : [\![?T.U]\!]$ which by (E-Inp) means $[\![\Phi]\!]; [\![\Gamma_1]\!]_{f'}; \emptyset \vdash f'_x : \ell_{\mathsf{i}}[[\![T]\!], [\![U]\!]]$, for some function f' and $[\![\Phi]\!]; [\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![U]\!] \uplus f''_y : [\![T]\!]; \mathcal{S} \vdash [\![P]\!]_{f''}$ for some function f''. Let $f''(x) = c, f''(y) = y$ and let $f = f' \cup f'' - \{x \mapsto c, y \mapsto y\}$. Then, by rewriting the previous $[\![\Phi]\!]; [\![\Gamma_1]\!]_f; \emptyset \vdash f_x : \ell_{\mathsf{i}}[[\![T]\!], [\![U]\!]]$ and $[\![\Phi]\!]; [\![\Gamma_2]\!]_f, y : [\![T]\!], c : [\![U]\!]; \mathcal{S} \vdash [\![P]\!]_{f,\{x \mapsto c\}}$. By applying (E-Input), (Tπ-HoInp) and Lemma 6.3.5 we have $[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \mathcal{S} \vdash$

$f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$.

- Case (T-HoInp2):

$$\frac{\Phi;\Gamma_1;\emptyset \vdash x : ?T.U \quad \Phi, y:T;\Gamma_2, x:U;\mathcal{S} \vdash P \quad \text{if } T = T' \overset{1}{\to} \sigma \text{ then } y \in \mathcal{S}}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} - \{y\} \vdash x?(y).P}$$

To prove that $[\![\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} - \{y\}]\!]_f \vdash [\![x?(y).P]\!]_f$ for some function f such that $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. By induction hypothesis $[\![\Phi]\!];[\![\Gamma_1]\!]_{f'};\emptyset \vdash f'_x : [\![?T.U]\!]$ which by (E-Inp) means $[\![\Phi]\!];[\![\Gamma_1]\!]_{f'};\emptyset \vdash f'_x : \ell_{\mathsf{i}}[[\![T]\!],[\![U]\!]]$, for some function f' and $[\![\Phi]\!];[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![U]\!] \uplus f''_y : [\![T]\!];\mathcal{S} \vdash [\![P]\!]_{f''}$, for some function f''. Let $f''(x) = c, f''(y) = y$ and let $f = f' \cup f'' - \{x \mapsto c, y \mapsto y\}$. Then, by rewriting the previous we have $[\![\Phi]\!];[\![\Gamma_1]\!]_f;\emptyset \vdash f_x : \ell_{\mathsf{i}}[[\![T]\!],[\![U]\!]]$ and $[\![\Phi]\!], y : [\![T]\!];[\![\Gamma_2]\!]_f, c : [\![U]\!];\mathcal{S} \vdash [\![P]\!]_{f,\{x\mapsto c\}}$. Then, by applying (E-Input), (Tπ- HoInp) and Lemma 6.3.5 we obtain $[\![\Phi]\!];[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f;\mathcal{S} \vdash f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$, where the condition if $T = T' \overset{1}{\to} \sigma$ then $y \in \mathcal{S}$ becomes if $[\![T]\!] = [\![T']\!] \overset{1}{\to} [\![\sigma]\!]$ then $y \in \mathcal{S}$.

- Case (T-HoOut):

$$\frac{\Phi;\Gamma_1;\emptyset \vdash x : !T.U \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash v : T \quad \Phi;\Gamma_3, x:U;\mathcal{S}_3 \vdash P \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi;\Gamma_1 \circ \Gamma_2 \circ \Gamma_3;\mathcal{S}_2 \cup \mathcal{S}_3 \vdash x!\langle v\rangle.P}$$

To prove that $[\![\Phi;\Gamma_1 \circ \Gamma_2 \circ \Gamma_3;\mathcal{S}_2 \cup \mathcal{S}_3]\!]_f \vdash [\![x!\langle v\rangle.P]\!]_f$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) \cup \mathsf{dom}(\Gamma_3)$. By (E-Output) and Lemma 6.3.5 it means that $[\![\Phi]\!];[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f;\mathcal{S}_2 \cup \mathcal{S}_3 \vdash (\nu c) f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$. By induction hypothesis we have that $[\![\Phi]\!];[\![\Gamma_1]\!]_{f'};\emptyset \vdash [\![x : !T.U]\!]_{f'}$ which by (E-Out) means that $[\![\Phi]\!];[\![\Gamma_1]\!]_{f'};\emptyset \vdash f'_x : \ell_{\mathsf{o}}[[\![T]\!],[\![\overline{U}]\!]]$, and $[\![\Phi]\!];[\![\Gamma_2]\!]_{f''};\mathcal{S}_2 \vdash f''_v : [\![T]\!]$, and $[\![\Phi]\!];[\![\Gamma_3]\!]_{f'''}, f'''_x : [\![U]\!];\mathcal{S}_3 \vdash [\![P]\!]_{f'''}$. Let $f'''_x = c$ and let $f = f' \cup f'' \cup f''' - \{x \mapsto c\}$. Then, by rewriting the previous we have $[\![\Phi]\!];[\![\Gamma_1]\!]_f;\emptyset \vdash f_x : \ell_{\mathsf{o}}[[\![T]\!],[\![\overline{U}]\!]]$ and $[\![\Phi]\!];[\![\Gamma_2]\!]_f;\mathcal{S}_2 \vdash f_v : [\![T]\!]$ and $[\![\Phi]\!];[\![\Gamma_3]\!]_f, c : [\![U]\!];\mathcal{S}_3 \vdash [\![P]\!]_{f,\{x\mapsto c\}}$. By applying rule (T$\pi$- HoOut) and (T$\pi$- HoSess) for deriving $c : [\![\overline{U}]\!]$ and by using Lemma 6.3.7 and

Lemma 6.3.4 to obtain $c : \ell_\sharp[W]$, where $[\![U]\!] = \ell_\alpha W$, we have the following:

$$\frac{\begin{array}{c}[\![\Phi]\!]; c : [\![\overline{U}]\!]; \emptyset \vdash c : [\![\overline{U}]\!] \\ [\![\Phi]\!]; [\![\Gamma_1]\!]_f; \emptyset \vdash f_x : \ell_o[[\![T]\!], [\![\overline{U}]\!]] \\ [\![\Phi]\!]; [\![\Gamma_2]\!]_f; \mathcal{S}_2 \vdash f_v : [\![T]\!] \\ [\![\Phi]\!]; [\![\Gamma_3]\!]_f, c : [\![U]\!]; \mathcal{S}_3 \vdash [\![P]\!]_{f,\{x \mapsto c\}}\end{array}}{[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f, c : \ell_\sharp[W]; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}}$$

Then by applying (Tπ-HoRes1) we have the following:

$$\frac{[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f, c : \ell_\sharp[W]; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}}{[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash (\nu c) f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}}$$

which concludes this case.

- Case (T-HoBrch):

$$\frac{\begin{array}{c}\Phi; \Gamma_1; \emptyset \vdash x : \&\{l_i : T_i\}_{i \in I} \\ \Phi; \Gamma_2, x : T_i; \mathcal{S} \vdash P_i \quad \forall i \in I\end{array}}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} \vdash x \triangleright \{l_i : P_i\}_{i \in I}}$$

To prove that $[\![\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S}]\!]_f \vdash [\![x \triangleright \{l_i : P_i\}_{i \in I}]\!]_f$ By (E-Branching) and Lemma 6.3.5 $[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \mathcal{S} \vdash f_x?(y).$ **case** y **of** $\{l_i\_c \triangleright [\![P_i]\!]_{f,\{x \mapsto c\}}\}_{i \in I}$. By induction hypothesis we have $[\![\Phi]\!]; [\![\Gamma_1]\!]_{f'}; \emptyset \vdash [\![x : \&\{l_i : T_i\}_{i \in I}]\!]_{f'}$ which by the encoding of branch type means $[\![\Phi]\!]; [\![\Gamma_1]\!]_{f'}; \emptyset \vdash f'_x : \ell_i[\langle l_i\_[\![T_i]\!]\rangle_{i \in I}]$, and $[\![\Phi]\!]; [\![\Gamma_2]\!]_{f''}, f''_x : [\![T_i]\!]; \mathcal{S} \vdash [\![P_i]\!]_{f''}\ \forall i \in I$. Let $f''_x = c$ and $f = f' \cup f'' - \{x \mapsto c\}$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Then, be rewriting the previous we have $[\![\Phi]\!]; [\![\Gamma_1]\!]_f; \emptyset \vdash f_x : \ell_i[\langle l_i\_[\![T_i]\!]\rangle_{i \in I}]$, and $[\![\Phi]\!]; [\![\Gamma_2]\!]_f, c : [\![T_i]\!]; \mathcal{S} \vdash [\![P_i]\!]_{f,\{x \mapsto c\}}\ \forall i \in I$. By applying Lemma 9.5.1, also $[\![\Phi]\!], y : \langle l_i\_[\![T_i]\!]\rangle_{i \in I}; [\![\Gamma_2]\!]_f, c : [\![T_i]\!]; \mathcal{S} \vdash [\![P_i]\!]_{f,\{x \mapsto c\}}\ \forall i \in I$. Then, by applying (T$\pi$-HoCase) and (T$\pi$-HoVar) for deriving $y : \langle l_i\_[\![T_i]\!]\rangle_{i \in I}$:

$$\frac{\begin{array}{c}[\![\Phi]\!], y : \langle l_i\_[\![T_i]\!]\rangle_{i \in I}; \emptyset; \emptyset \vdash y : \langle l_i\_[\![T_i]\!]\rangle_{i \in I} \\ [\![\Phi]\!], y : \langle l_i\_[\![T_i]\!]\rangle_{i \in I}; [\![\Gamma_2]\!]_f, c : [\![T_i]\!]; \mathcal{S} \vdash [\![P_i]\!]_{f,\{x \mapsto c\}}\ \forall i \in I\end{array}}{[\![\Phi]\!], y : \langle l_i\_[\![T_i]\!]\rangle_{i \in I}; [\![\Gamma_2]\!]_f; \mathcal{S} \vdash \textbf{case}\ y\ \textbf{of}\ \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x \mapsto c\}}\}_{i \in I}}$$

Then, by applying (Tπ-HoInp) we have:

$$\frac{\begin{array}{c}[\![\Phi]\!];[\![\Gamma_1]\!]_f;\emptyset \vdash f_x : \ell_{\mathsf{i}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \\ [\![\Phi]\!], y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I};[\![\Gamma_2]\!]_f;\mathcal{S} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}\end{array}}{[\![\Phi]\!];[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f;\mathcal{S} \vdash f_x?(y).\, \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

which concludes this case.

- Case (T-HoSel):

$$\frac{\begin{array}{c}[\![\Phi]\!];\Gamma_1;\emptyset \vdash x : \oplus\{l_i : T_i\}_{i\in I} \\ [\![\Phi]\!];\Gamma_2, x : T_j;\mathcal{S} \vdash P \quad j \in I\end{array}}{[\![\Phi]\!];\Gamma_1 \circ \Gamma_2;\mathcal{S} \vdash x \triangleleft l_j.P}$$

To prove that $[\![\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S}]\!]_f \vdash [\![x \triangleleft l_j.P]\!]_f$. By (E-Selection) and Lemma 6.3.5 it means that $[\![\Phi]\!];[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f;\mathcal{S} \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$. By induction hypothesis we have that $[\![\Phi]\!];[\![\Gamma_1]\!]_{f'};\emptyset \vdash [\![x : \oplus\{l_i : T_i\}_{i\in I}]\!]_{f'}$ which by the encoding of select type means that $[\![\Phi]\!];[\![\Gamma_1]\!]_{f'};\emptyset \vdash f'_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$, and $[\![\Phi]\!];[\![\Gamma_2]\!]_{f''}, f''_x : [\![T_j]\!];\mathcal{S} \vdash [\![P]\!]_{f''}\ j \in I$. Let $f''_x = c$ and $f = f' \cup f'' - \{x \mapsto c\}$, where $\mathsf{dom}(f) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$. Then the induction hypothesis become $[\![\Phi]\!];[\![\Gamma_1]\!]_f;\emptyset \vdash f_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$, and $[\![\Phi]\!];[\![\Gamma_2]\!]_f, c : [\![T_j]\!];\mathcal{S} \vdash [\![P]\!]_{f,\{x\mapsto c\}}\ j \in I$. Then, by applying rule (Tπ-HoSess), to derive $c : [\![\overline{T_j}]\!]$, and rule (T$\pi$-HoLVal) we have:

$$\dfrac{\dfrac{}{[\![\Phi]\!];c : [\![\overline{T_j}]\!];\emptyset \vdash c : [\![\overline{T_j}]\!]}\ (\text{T}\pi\text{-HoSes})}{[\![\Phi]\!];c : [\![\overline{T_j}]\!];\emptyset \vdash l_j\_c : \langle l_i\_[\![\overline{T_j}]\!]\rangle_{i\in I}}\ (\text{T}\pi\text{-HoLVal})$$

Then, by rule (Tπ-HoOut):

$$\frac{\begin{array}{c}[\![\Phi]\!];[\![\Gamma_1]\!]_f;\emptyset \vdash f_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \\ [\![\Phi]\!];c : [\![\overline{T_j}]\!];\emptyset \vdash l_j\_c : \langle l_i\_[\![\overline{T_j}]\!]\rangle_{i\in I} \\ [\![\Phi]\!];[\![\Gamma_2]\!]_f, c : [\![T_j]\!];\mathcal{S} \vdash [\![P]\!]_{f,\{x\mapsto c\}}\ j \in I\end{array}}{[\![\Phi]\!];[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus c : \ell_\sharp[W];\mathcal{S} \vdash f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

We have used Lemma 6.3.7 and Lemma 6.3.4 to obtain $c : \ell_\sharp[W]$, where

$[\![T_j]\!] = \ell_\alpha[W]$ and $[\![\overline{T_j}]\!] = \ell_{\overline{\alpha}}[W]$. By applying (T$\pi$-HoRes1):

$$\frac{[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus c : \ell_\sharp[W]; \mathcal{S} \vdash f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Phi]\!]; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \mathcal{S} \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

we conclude this case.

□

We show now the completeness of typing derivations for processes by using the encoding.

Theorem 9.5.3 (Completeness: Typing HOπ Processes)**.** *If* $[\![\Phi; \Gamma; \mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$*, then* $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$.

Proof. The proof is done by induction on the structure of P.

- Case $\lambda x : T.P$:
  By equation (E-Abstraction) and (E-FunType) we have $[\![\lambda x : T.P]\!]_f \stackrel{\texttt{def}}{=} \lambda x : [\![T]\!].[\![P]\!]_f$ and $[\![T \to \sigma]\!] \stackrel{\texttt{def}}{=} [\![T]\!] \to [\![\sigma]\!]$, and assume $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]$. Then, either rule (T$\pi$-HoAbs1) or rule (T$\pi$-HoAbs2) is applied. We consider both cases in the following:

  - Rule (Tπ-HoAbs1) is applied:

    $$\frac{\begin{array}{c}[\![\Phi]\!], x : [\![T]\!]; [\![\Gamma]\!]_f; \mathcal{S}' \vdash [\![P]\!]_f : [\![\sigma]\!] \\ \text{if } [\![T]\!] = T_1^\pi \stackrel{1}{\to} \sigma_1^\pi \text{ then } x \in \mathcal{S}'\end{array}}{[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S}' - \{x\} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]}$$

    where $\mathcal{S} = \mathcal{S}' - \{x\}$. By induction hypothesis $\Phi, x : T; \Gamma; \mathcal{S}' \vdash P : \sigma$. Then, we obtain the result by applying rule (T-HoAbs1).

  - Rule (Tπ-HoAbs2) is applied:

    $$\frac{[\![\Phi]\!]; [\![\Gamma]\!]_f, x : [\![T]\!]; \mathcal{S} \vdash [\![P]\!]_f : [\![\sigma]\!]}{[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]}$$

    By induction hypothesis $\Phi; \Gamma, x : T; \mathcal{S} \vdash P : \sigma$. Then, we obtain the result by applying rule (T-HoAbs1).

The cases for other values, different from the lambda abstraction, are trivial, as the encoding is an homomorphism and the typing rules on both π-calculus with and without sessions, follow the same line.t

- Case PQ:
  By equation (E-Application) we have $[\![PQ]\!]_f \stackrel{\texttt{def}}{=} [\![P]\!]_f[\![Q]\!]_f$ and assume $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash [\![P]\!]_f[\![Q]\!]_f : [\![\sigma]\!]$. Then, rule (T$\pi$-HoApp) is applied:

$$\frac{[\![\Phi]\!]; \Gamma_1^\pi; \mathcal{S}_1 \vdash [\![P]\!]_f : T^\pi \stackrel{1}{\to} [\![\sigma]\!] \quad [\![\Phi]\!]; \Gamma_2^\pi; \mathcal{S}_2 \vdash [\![Q]\!]_f : T^\pi \quad \text{if } T^\pi = T_1^\pi \to \sigma_1^\pi \text{ then } \mathsf{un}(\Gamma_2^\pi) \text{ and } \mathcal{S}_2 = \emptyset}{[\![\Phi]\!]; \Gamma_1^\pi \uplus \Gamma_2^\pi; \mathcal{S}_1 \cup \mathcal{S}_2 \vdash [\![P]\!]_f[\![Q]\!]_f : [\![\sigma]\!]}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By induction hypothesis $\Phi; \Gamma_1; \mathcal{S}_1 \vdash P : T \stackrel{1}{\to} \sigma$ where $T^\pi = [\![T]\!]$, and $\Phi; \Gamma_2; \mathcal{S}_2 \vdash Q : T$. Then, the result follows by applying rule (T-HoApp) on the induction hypothesis.

- Cases $\mathbf{0}$, $P \mid Q$, **if** v **then** P **else** Q, $(\nu xy)P$ follow exactly the same line as the corresponding ones in Theorem 6.3.11.

- Case $x?(y).P$:
  By equation (E-Input) we have $[\![x?(y).P]\!]_f \stackrel{\texttt{def}}{=} f_x?(y,c).[\![P]\!]_{f,\{x \mapsto c\}}$ and assume $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash f_x?(y,c).[\![P]\!]_{f,\{x \mapsto c\}}$. Then, rule (T$\pi$-HoInp) is applied:

$$\frac{[\![\Phi]\!]; \Gamma_1^\pi; \emptyset \vdash f_x : \ell_{\mathrm{i}}[T^\pi, U^\pi] \quad ([\![\Phi]\!]; \Gamma_2^\pi), y : T^\pi, c : U^\pi; \mathcal{S} \vdash [\![P]\!]_{f,\{x \mapsto c\}} \quad \text{if } T^\pi = T_1^\pi \stackrel{1}{\to} \sigma^\pi, \text{ then } y \in \mathcal{S}}{[\![\Phi]\!]; \Gamma_1^\pi \uplus \Gamma_2^\pi; \mathcal{S}' - \{y\} \vdash f_x?(y,c).[\![P]\!]_{f,\{x \mapsto c\}}}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$; and $\mathcal{S} = \mathcal{S}' - \{y\}$. By induction hypothesis $\Phi; \Gamma_1; \emptyset \vdash x : ?T.U$ and depending on whether y is a channel variable or not we have either $\Phi; \Gamma_2, x : U, y : T; \mathcal{S} \vdash P$ or $\Phi, y : T; \Gamma_2, x : U; \mathcal{S} \vdash P$ where $T^\pi = [\![T]\!]$, $U^\pi = [\![U]\!]$. Then, we apply either rule (T-HoInp1) or rule (T-HoInp2). We consider both cases in the following:

– Rule (T-HoInp1) is applied:

$$\frac{\Phi; \Gamma_1; \emptyset \vdash x : ?T.U \qquad \Phi; \Gamma_2, x : U, y : T; \mathcal{S} \vdash P}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} \vdash x?(y).P}$$

– Rule (T-HoInp2) is applied:

$$\frac{\begin{array}{c}\Phi; \Gamma_1; \emptyset \vdash x : ?T.U \\ \Phi, y : T; \Gamma_2, x : U; \mathcal{S} \vdash P \\ \text{if } T = T' \xrightarrow{1} \sigma \text{ then } y \in \mathcal{S}\end{array}}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} - \{y\} \vdash x?(y).P}$$

since $T^\pi = [\![T]\!]$, then $T_1^\pi \xrightarrow{1} \sigma^\pi = [\![T' \xrightarrow{1} \sigma]\!]$.

- Case $x!\langle v\rangle.P$:
  By equation (E-Output) we have $[\![x!\langle v\rangle.P]\!]_f \stackrel{\text{def}}{=} (\nu c) f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Phi]\!]; [\![\Gamma]\!]_f; \mathcal{S} \vdash (\nu c) f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ which by rules (Tπ-HoRes1), since c is present in $[\![P]\!]_{f,\{x\mapsto c\}}$, and (Tπ-HoOut), and (Tπ-HoSess) to derive $c : \ell_\alpha[W]$ means:

$$\frac{\begin{array}{c}[\![\Phi]\!]; \Gamma_1^\pi; \emptyset \vdash f_x : \ell_o[T^\pi, U^\pi] \qquad [\![\Phi]\!]; \Gamma_2^\pi; \mathcal{S}_2 \vdash f_v : T^\pi \\ [\![\Phi]\!]; \Gamma_3^\pi, c : \ell_{\overline{\alpha}}[W]; \mathcal{S}_3 \vdash [\![P]\!]_{f,\{x\mapsto c\}} \qquad [\![\Phi]\!]; c : \ell_\alpha[W]; \emptyset \vdash c : \ell_\alpha[W] \\ \text{if } T^\pi = T_1^\pi \to \sigma_1^\pi, \text{ then } \mathsf{un}(\Gamma_2^\pi) \text{ and } \mathcal{S}_2 = \emptyset\end{array}}{\dfrac{[\![\Phi]\!]; [\![\Gamma]\!]_f, c : \ell_\sharp[T^\pi, U^\pi]; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Phi]\!]; \Gamma_1^\pi \uplus \Gamma_2^\pi \uplus \Gamma_3^\pi; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash (\nu c) f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}}$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi \uplus \Gamma_3^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ and $\Gamma_3^\pi = [\![\Gamma_3]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$; and $\mathcal{S} = \mathcal{S}_2 \cup \mathcal{S}_3$. By induction hypothesis we have $\Phi; \Gamma_1; \emptyset \vdash x : !T.U$ where $\ell_o[T^\pi, U^\pi] = [\![!T.U]\!]$, which by (E-Out) means that $T^\pi = [\![T]\!]$ and $U^\pi = [\![\overline{U}]\!] = \ell_\alpha[W]$, and $\Phi; \Gamma_2; \mathcal{S}_2 \vdash v : T$, and $\Phi; \Gamma_3, x : U; \mathcal{S}_3 \vdash P$, where $[\![U]\!] = \ell_{\overline{\alpha}}[W]$, by Lemma 6.3.7. By applying rule (T-HoOut) on the induction hypothesis and translating "if $T^\pi = T_1^\pi \to \sigma_1^\pi$, then $\mathsf{un}(\Gamma_2^\pi)$ and $\mathcal{S}_2 = \emptyset$" in "if $T = T_1 \to \sigma_1$ then $\mathsf{un}(\Gamma_2)$ and $\mathcal{S}_2 = \emptyset$" we obtain $\Phi; \Gamma_1 \circ \Gamma_2 \circ \Gamma_3; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash x!\langle v\rangle.P$

- Case $x \triangleright \{l_i : P_i\}_{i\in I}$:
  By equation (E-Branching) $[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f = f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright$

$[\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$ and assume $[\![\Phi]\!];[\![\Gamma]\!]_f \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$ which by rules (Tπ- HoInp) and (Tπ- HoCase) and (Tπ- HoVar) to derive $y : \langle l_i\_T_i^\pi\rangle_{i\in I}$ and Lemma 6.3.3 we have the following:

$$\dfrac{\begin{array}{c}(\text{T}\pi\text{-HoCase})\\ [\![\Phi]\!], y : \langle l_i\_T_i^\pi\rangle_{i\in I}; \emptyset; \emptyset \vdash y : \langle l_i\_T_i^\pi\rangle_{i\in I}\\ [\![\Phi]\!]; \Gamma_2^\pi, c : T_i^\pi; \mathcal{S} \vdash [\![P_i]\!]_{f,\{x\mapsto c\}} \quad \forall i \in I\end{array}}{[\![\Phi]\!], y : \langle l_i\_T_i^\pi\rangle_{i\in I}; \Gamma_2^\pi; \mathcal{S} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

and

$$\dfrac{\begin{array}{c}(\text{T}\pi\text{-HoInp})\\ [\![\Phi]\!]; \Gamma_1^\pi; \emptyset \vdash f_x : \ell_{\text{i}}[\langle l_i\_T_i^\pi\rangle_{i\in I}]\\ [\![\Phi]\!], y : \langle l_i\_T_i^\pi\rangle_{i\in I}; \Gamma_2^\pi; \mathcal{S} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}\end{array}}{[\![\Phi]\!];[\![\Gamma]\!]_f; \mathcal{S} \vdash f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By induction hypothesis we have $\Phi; \Gamma_1; \emptyset \vdash x : \&\{l_i : T_i\}_{i\in I}$ where $[\![\&\{l_i : T_i\}_{i\in I}]\!]_f = \ell_{\text{i}}[\langle l_i\_T_i^\pi\rangle_{i\in I}]$ by applying (E-Branch) and hence $\forall i\ [\![T_i]\!] = T_i^\pi$. We have also $\Phi; \Gamma_2, x : T_i; \mathcal{S} \vdash P_i\ \forall i \in I$. By applying rule (T-HoBrch) we obtain $\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} \vdash x \triangleright \{l_i : P_i\}_{i\in I}$.

- Case $x \triangleleft l_j.P$:
  By equation (E-Selection) we have $[\![x \triangleleft l_j.P]\!]_f = (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Phi]\!];[\![\Gamma]\!]_f; \mathcal{S} \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ which by rules (Tπ-HoRes1), since c is present in $[\![P]\!]_{f,\{x\mapsto c\}}$, and (T$\pi$-HoOut) and (T$\pi$-HoLval) means:

$$\dfrac{\dfrac{}{[\![\Phi]\!]; c : T_j^\pi; \emptyset \vdash c : T_j^\pi \quad j \in I}\ (\text{T}\pi\text{-HoSess})}{[\![\Phi]\!]; c : T_j^\pi; \emptyset \vdash l_j\_c : \langle l_i\_T_i^\pi\rangle_{i\in I}}\ (\text{T}\pi\text{-HoLval})$$

$$\dfrac{\begin{array}{c}(\text{T}\pi\text{-HoRes1})\\ (\text{T}\pi\text{-HoOut})\\ [\![\Phi]\!]; \Gamma_1^\pi; \emptyset \vdash f_x : \ell_{\text{o}}\ [\langle l_i\_T_i^\pi\rangle_{i\in I}]\end{array}}{\dfrac{\dfrac{[\![\Phi]\!]; \Gamma_2^\pi, c : \overline{T_j^\pi}; \mathcal{S} \vdash [\![P]\!]_{f,\{x\mapsto c\}} \quad [\![\Phi]\!]; c : T_j^\pi; \emptyset \vdash l_j\_c : \langle l_i\_T_i^\pi\rangle_{i\in I}}{[\![\Phi]\!];[\![\Gamma]\!]_f, c : \ell_\sharp\,[W]; \mathcal{S} \vdash f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}}{[\![\Phi]\!];[\![\Gamma]\!]_f; \mathcal{S} \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.6 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. Notice that c is a channel and one capability of c is sent along l_j_c whether the other is used in the continuation $[\![P]\!]_{f,\{x\mapsto c\}}$,

hence $\ell_\sharp\,[W] = T_j^\pi \uplus \overline{T_j^\pi}$. By induction hypothesis we have $\Phi; \Gamma_1; \emptyset \vdash x : \oplus\{l_i : T_i\}_{i\in I}$ where by (E-Select) $\ell_\mathsf{o}\,[\langle l_i\_T_i^\pi\rangle_{i\in I}] = [\![\oplus\{l_i : T_i\}_{i\in I}]\!]$ and $T_i^\pi = [\![\overline{T_i}]\!]\ \forall i \in I$. We also have $\Phi; \Gamma_2, x : T_j; \mathcal{S} \vdash P$. By applying rule (T-HoSel) we obtain $\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} \vdash x \triangleleft l_j.P$.

□

9.5.2 Operational Correspondence for HOπ

In this section we present the operational correspondence that relates the extended encoding on higher-order constructs and the reduction relation in the session HOπ and the standard HOπ .

Before presenting the main theorem, we recall the encoding of a substitution in a session process, given by Definition 6.3.13. Since the syntax of processes for HOπ includes also values, which in case of abstraction use processes in the body, we need to modify Definition 6.3.13 to accommodate this feature. Formally, the new definition is as follows.

Definition 9.5.4. *Let Q be a session process HOπ and $z \in FV(Q)$ and let $Q[v/z]$ denote process Q where variable z is substituted by value v. Then,*

$$[\![Q[v/z]]\!]_f \stackrel{def}{=} [\![Q]\!]_f[f_v/z]$$

We are ready now to give the operational correspondence for HOπ constructs. The following theorem is an extension of the main Theorem 6.3.14 with the reductions for the λ-abstractions.

Theorem 9.5.5 (OC for HOπ Constructs)**.** *Let P be a HOπ process with sessions. Then,*

1. *If $P \to P'$ then $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$,*
2. *If $[\![P]\!]_f \to\equiv Q$ then, $\exists\, P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \to^*\equiv [\![P']\!]_{f'}$, where f' is the updated f after the communication and $f_x = f_y$ for all $(\boldsymbol{\nu} xy) \in \mathcal{E}[\cdot]$.*

Proof. We consider both cases in the following.

1. The proof is done by induction on the length of the derivation $P \to P'$.

- Case (R-Beta):

$$(\lambda x : T.Q)v \to Q[v/x]$$

By the encoding of HOπ with sessions we have

$$\begin{aligned}
&[\![P]\!]_f \\
&\quad = \quad [\![(\lambda x : T.Q)v]\!]_f \stackrel{\texttt{def}}{=} \lambda x : [\![T]\!].[\![Q]\!]_f f_v \\
&\quad \to \quad [\![Q]\!]_f[f_v/x]
\end{aligned}$$

On the other hand

$$[\![P']\!]_f = [\![Q[v/x]]\!]_f = [\![Q]\!]_f[f_v/x]$$

Where we use Definition 9.5.4. Which means that $[\![P]\!]_f \to\equiv [\![P']\!]_f$.

- Case (R-AppLeft):

$$\frac{P \to P'}{PQ \to P'Q}$$

By induction hypothesis $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$. We conclude by applying (Rπ-AppLeft) and (Rπ-Struct).

- Case (R-AppRight):

$$\frac{P \to P'}{vP \to vP'}$$

This case follows exactly the same line as the previous one.

2. The proof is done by induction on the structure of the HOπ process with sessions, P. The only new case to consider is the application $P = P_1P_2$. By the encoding (E-Application), we have $[\![P_1]\!]_f[\![P_2]\!]_f \to\equiv Q$. We have to show that $\exists P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \to^*\equiv [\![P']\!]_f$. There are the following 3 cases to be considered:

   (a) $P = (\lambda x : T.Q')v$ and an application occurs. Hence, by equation (E-Abstraction), $[\![\lambda x : T.Q']\!]_f[\![v]\!]_f \stackrel{\texttt{def}}{=} (\lambda x : [\![T]\!].[\![Q']\!]_f)f_v \to [\![Q']\!]_f[f_v/x] \equiv Q$. We show that $\exists P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \to^*\equiv [\![P']\!]_{f'}$. Let $\mathcal{E}[\cdot] = [\cdot]$, and let $f' = f$. Then, $P \to Q'[v/x] = P'$ and $[\![P']\!]_f = [\![Q']\!]_f[f_v/x]$ and $Q \equiv [\![Q']\!]_f[f_v/x]$, where we use Definition 9.5.4 to encode the substitution that occurs in Q'.

(b) Only $[\![P_1]\!]_f$ reduces: it means that $Q \equiv R[\![P_2]\!]_f$. By induction hypothesis, since P_1 is a subprocess of P and $[\![P_1]\!]_f \rightarrow R$ then there exist $P_1', \mathcal{E}'[\cdot]$ such that $\mathcal{E}'[P_1] \rightarrow \mathcal{E}'[P_1']$ and $R \rightarrow^* \equiv [\![P_1']\!]_f$. Then, by (R-AppLeft) $\mathcal{E}'[P_1]P_2 \rightarrow \mathcal{E}'[P_1']P_2$. Let $\mathcal{E}[\cdot] = \mathcal{E}'[\cdot]$ and hence $\mathcal{E}[P] = \mathcal{E}'[P_1]P_2 \rightarrow \mathcal{E}'[P_1']P_2 = \mathcal{E}[P']$, by applying structural congruence and so $P' = P_1'P_2$. Then, $R[\![P_2]\!]_f \rightarrow^* \equiv [\![P_1']\!]_f[\![P_2]\!]_f = [\![P']\!]_f$.

(c) Only $[\![P_2]\!]_f$ reduces: this case follows the same line as the previous one, by changing P_1 with P_2 and by using (R-AppRight) instead of (R-AppLeft) and structural congruence.

□

$$\text{(T-HoInact)}\ \frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset \vdash \mathbf{0}} \qquad \text{(T-HoPar)}\ \frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash P \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash P \mid Q}$$

$$\text{(T-HoRes)}\ \frac{\Phi;\Gamma, x : T, y : \overline{T};\mathcal{S} \vdash P}{\Phi;\Gamma;\mathcal{S} \vdash (\boldsymbol{\nu} xy)P} \qquad \text{(T-HoInp1)}\ \frac{\Phi;\Gamma_1;\emptyset \vdash x : ?T.U \quad \Phi;\Gamma_2, x : U, y : T;\mathcal{S} \vdash P}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} \vdash x?(y).P}$$

$$\text{(T-HoIf)}\ \frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash v : \mathtt{Bool} \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash P \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q}$$

$$\text{(T-HoInp2)}\ \frac{\Phi;\Gamma_1;\emptyset \vdash x : ?T.U \quad \Phi, y : T;\Gamma_2, x : U;\mathcal{S} \vdash P \quad \text{if } T = T' \overset{1}{\to} \sigma \text{ then } y \in \mathcal{S}}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} - \{y\} \vdash x?(y).P}$$

$$\text{(T-HoOut)}\ \frac{\Phi;\Gamma_1;\emptyset \vdash x : !T.U \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash v : T \quad \Phi;\Gamma_3, x : U;\mathcal{S}_3 \vdash P \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi;\Gamma_1 \circ \Gamma_2 \circ \Gamma_3;\mathcal{S}_2 \cup \mathcal{S}_3 \vdash x!\langle v \rangle.P}$$

$$\text{(T-HoBrch)}\ \frac{\Phi;\Gamma_1;\emptyset \vdash x : \&\{l_i : T_i\}_{i \in I} \quad \Phi;\Gamma_2, x : T_i;\mathcal{S} \vdash P_i \quad \forall i \in I}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} \vdash x \triangleright \{l_i : P_i\}_{i \in I}}$$

$$\text{(T-HoSel)}\ \frac{\Phi;\Gamma_1;\emptyset \vdash x : \oplus\{l_i : T_i\}_{i \in I} \quad \Phi;\Gamma_2, x : T_j;\mathcal{S} \vdash P \quad j \in I}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} \vdash x \triangleleft l_j.P}$$

Figure 9.4: Typing rules for the HOπ with sessions: processes

$$\text{(T}\pi\text{-HoSess)}\ \frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma, x : T;\emptyset \vdash x : T} \qquad \text{(T}\pi\text{-HoVar)}\ \frac{T \neq T' \overset{1}{\to} \sigma \qquad \mathsf{un}(\Gamma)}{\Phi, x : T;\Gamma;\emptyset \vdash x : T}$$

$$\text{(T}\pi\text{-HoBool)}\ \frac{\mathsf{un}(\Gamma) \quad v = \texttt{true} \,/\, \texttt{false}}{\Phi;\Gamma;\emptyset \vdash v : \texttt{Bool}} \qquad \text{(T}\pi\text{-HoFun)}\ \frac{\mathsf{un}(\Gamma)}{\Phi, x : T \overset{1}{\to} \sigma;\Gamma;\{x\} \vdash x : T \overset{1}{\to} \sigma}$$

$$\text{(T}\pi\text{-HoUnit)}\ \frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset \vdash \star : \texttt{Unit}} \qquad \text{(T}\pi\text{-HoAbs1)}\ \frac{\Phi, x : T;\Gamma;\mathcal{S} \vdash P : \sigma \quad \text{if } T = T' \overset{1}{\to} \sigma \text{ then } x \in \mathcal{S}}{\Phi;\Gamma;\mathcal{S} - \{x\} \vdash \lambda x : T.P : T \to \sigma}$$

$$\text{(T}\pi\text{-HoAbs2)}\ \frac{\Phi;\Gamma, x : T;\mathcal{S} \vdash P : \sigma}{\Phi;\Gamma;\mathcal{S} \vdash \lambda x : T.P : T \to \sigma} \qquad \text{(T}\pi\text{-HoSub)}\ \frac{\Phi;\Gamma;\mathcal{S} \vdash P : T \to \sigma}{\Phi;\Gamma;\mathcal{S} \vdash P : T \overset{1}{\to} \sigma}$$

$$\text{(T}\pi\text{-HoLVal)}\ \frac{\Phi;\Gamma;\mathcal{S} \vdash v : T_j \quad j \in I}{\Phi;\Gamma;\mathcal{S} \vdash l_j\_v : \langle l_i\_T_i \rangle_{i \in I}} \qquad \text{(T}\pi\text{-HoApp)}\ \frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash P : T \overset{1}{\to} \sigma \qquad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q : T \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash PQ : \sigma}$$

Figure 9.5: Typing rules for the standard HOπ: values and functions

$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset \vdash \mathbf{0}}\ (\text{T}\pi\text{-HoInact}) \qquad \frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash P \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash P \mid Q}\ (\text{T}\pi\text{-HoPar})$$

$$\frac{\Phi;\Gamma, x : \ell_\alpha[\widetilde{T}];\mathcal{S} \vdash P}{\Phi;\Gamma;\mathcal{S} \vdash (\nu x)P}\ (\text{T}\pi\text{-HoRes1}) \qquad \frac{\Phi;\Gamma;\mathcal{S} \vdash P}{\Phi;\Gamma;\mathcal{S} \vdash (\nu x)P}\ (\text{T}\pi\text{-HoRes2})$$

$$\frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash v : \texttt{Bool} \qquad \Phi;\Gamma_2;\mathcal{S}_2 \vdash P \qquad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q}\ (\text{T}\pi\text{-HoIf})$$

$$\frac{\Phi;\Gamma_1;\emptyset \vdash x : \ell_{\mathsf{i}}[\widetilde{T}] \qquad (\Phi;\Gamma_2), \tilde{y} : \widetilde{T};\mathcal{S} \vdash P \qquad \text{if } \exists\, \tilde{\tilde{y}} : \widetilde{\widetilde{T}} \subseteq \tilde{y} : \widetilde{T} \text{ s.t. } T_i = T' \xrightarrow{1} \sigma,\ T_i \in \widetilde{\widetilde{T}} \text{ then } \tilde{\tilde{y}} \in \mathcal{S}}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S} - \{\tilde{\tilde{y}}\} \vdash x?(\tilde{y}).P}\ (\text{T}\pi\text{-HoInp})$$

$$\frac{\Phi;\Gamma_1;\emptyset \vdash x : \ell_{\mathsf{o}}[\widetilde{T}] \qquad \Phi;\widetilde{\Gamma_2};\widetilde{\mathcal{S}_2} \vdash \tilde{v} : \widetilde{T} \qquad \Phi;\Gamma_3;\mathcal{S}_3 \vdash P \qquad \text{if } T_i = T' \to \sigma', \text{ then } \mathsf{un}(\Gamma_{2i}) \text{ and } \mathcal{S}_{2i} = \emptyset}{\Phi;\Gamma_1 \uplus \widetilde{\Gamma_2} \uplus \Gamma_3;\widetilde{\mathcal{S}_2} \cup \mathcal{S}_3 \vdash x!\langle\tilde{v}\rangle.P}\ (\text{T}\pi\text{-HoOut})$$

$$\frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash v : \langle l_i \_ T_i\rangle_{i\in I} \qquad \Phi;\Gamma_2, x_i : T_i;\mathcal{S}_2 \vdash P_i \quad \forall i \in I}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash \textbf{case } v \textbf{ of } \{l_i \_ x_i \triangleright P_i\}_{i\in I}}\ (\text{T}\pi\text{-HoCase})$$

Figure 9.6: Typing rules for the standard HOπ: processes

$$[\![T \xrightarrow{1} \sigma]\!] \stackrel{\texttt{def}}{=} [\![T]\!] \xrightarrow{1} [\![\sigma]\!] \qquad \text{(E-LinFunType)}$$

$$[\![T \to \sigma]\!] \stackrel{\texttt{def}}{=} [\![T]\!] \to [\![\sigma]\!] \qquad \text{(E-FunType)}$$

Figure 9.7: Encoding of HOπ types

$$
\begin{array}{rcll}
[\![\star]\!]_f & \stackrel{\text{def}}{=} & \star & \text{(E-Star)} \\
[\![\lambda x : T.P]\!]_f & \stackrel{\text{def}}{=} & \lambda x : [\![T]\!].[\![P]\!]_f & \text{(E-Abstraction)} \\
[\![PQ]\!]_f & \stackrel{\text{def}}{=} & [\![P]\!]_f [\![Q]\!]_f & \text{(E-Application)} \\
[\![x!\langle v\rangle.P]\!]_f & \stackrel{\text{def}}{=} & (\nu c) f_x!\langle f_v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} & \text{(E-Output)}
\end{array}
$$

Figure 9.8: Encoding of HOπ terms

CHAPTER 10

Recursion

So far we have presented processes that have finite behaviours. – in particular we have omitted the replication process $!P$ which is present in the π-calculus [95, 101]. – However, another way of modelling infinite behaviours is by using *recursion*. In fact, recursion is widely known and used not only in process calculi (and in CCS), but also in other programming paradigms, like functional or imperative programming. Replication is more specific to process algebras and is a simple form of recursion, that states what is exactly needed, for example in representing data and functions [101]. However, there is a strong relation between recursion and replication. In [101] it is shown that recursion definitions can be represented by replication and replication is redundant in the presence of recursion. Moreover, [95] shows via encoding the strong relation between the two constructs.

In this chapter we add *recursion* and *recursive types* both to the π-calculus with sessions, given in Chapter 5 and to the standard π-calculus, given in Chapter 4.

10.1 Syntax

In this section we present the syntax of types and the syntax of terms, of the π-calculus with sessions and the standard π-calculus.

Recursion in π-calculus with sessions The syntax of recursive types and recursive processes in the π-calculus with sessions is given in Fig. 10.1.

$T ::=$	$\mathbf{t}$	type variable
	$\mu\mathbf{t}.T$	recursive type
	$\cdots$	other type constructs
$P ::=$	X	process variable
	$\mathbf{rec}X.P$	recursive process
	$\cdots$	other process constructs

Figure 10.1: Syntax of recursive session constructs

Recursion in standard π-calculus The syntax of recursive types and recursive processes in the standard π-calculus is given in Fig. 10.2.

$m_\alpha ::=$	ℓ_α	linear qualifier used in α
	α	unrestricted qualifier used in α
$\tau ::=$	$m_\alpha[\widetilde{T}]$	channel type used in m
	$\emptyset[\widetilde{T}]$	channel with no capability
$T ::=$	τ	channel type
	$\mathbf{t}$	type variable
	$\mu\mathbf{t}.T$	recursive type
	$\cdots$	other type constructs
$P ::=$	X	process variable
	$\mathbf{rec}X.P$	recursive process
	$\cdots$	other process constructs

Figure 10.2: Syntax of recursive standard π constructs

Some comments on recursive types follow. They hold for both recursive types in sessions and in standard π- calculus. The μ and **rec** operators are binders of the type and process variables, respectively. Recursive expressions are finite representations of infinite objects. Consequently, a recursive type can be seen as a finite representation of a (possibly infinite) type expression. So, the type $\mu\mathbf{t}.T$ is the solution to the equation $\mathbf{t} = T$, which is obtained by replacing the free occurrence of the type variable $\mathbf{t}$ in T with T itself. In order to avoid meaningless types, like $\mu\mathbf{t}.\mathbf{t}$, we require that our recursive types, on both calculi with and without sessions, satisfy the constraint that the type variable $\mathbf{t}$ of the $\mu\mathbf{t}.T$ expression should

be *guarded* in the type T, which means that can occur free only underneath at least one of the other type constructs in the syntax. Moreover, recursive types are *contractive*, namely do not contain subexpressions like $\mu\mathbf{t}_1 \ldots \mu\mathbf{t}_n.T$. To conclude, in this work we adopt the following notion of *well-formedness* of recursive types: recursive variables do not occur in a message position. For example, the type $\mu\mathbf{t}.!\mathbf{t}$ is considered to be ill-formed, and hence it is ruled out. We will explain the reason for this choice in the next paragraphs.

10.2 Semantics

The reduction rule for the recursive process is the same for both π- calculi with and without sessions and is given in the following.

$$\text{(R}[\pi]\text{-Rec)} \qquad \frac{P[\mathbf{rec}X.P/X] \rightarrow P'}{\mathbf{rec}X.P \rightarrow P'}$$

The rest of the reduction rules are the same as in the corresponding sections where the operational semantics is given.

10.3 Typing Rules

Operation on types for π-calculus with sessions Type duality for session types is defined as follows. It extends the original inductive type duality for finite types, to accommodate recursive types.

$$\begin{aligned} \overline{\mathbf{t}} &= \mathbf{t} \\ \overline{\mu\mathbf{t}.T} &= \mu\mathbf{t}.\overline{T} \end{aligned}$$

However, type duality for recursive session types is a delicate matter. Recent work [8, 9] has shown that inductive duality is not complete. In particular, in the presence of recursive types having a type variable as a carried type, for example $\mu\mathbf{t}.!\mathbf{t}$, it is unsafe to adopt inductive duality, since the latter does not commute with unfolding. In order to overcome this problem, we follow the standard way adopted in the literature, namely considering the above type to be ill-formed. We let the exploration of more accurate duality relations as future work.

Operation on types for standard π-calculus Type duality for standard π-types is defined as follows:

$$\begin{aligned}\overline{m_{\mathsf{i}}\,[\widetilde{T}]} &= m_{\mathsf{o}}\,[\widetilde{T}]\\ \overline{m_{\mathsf{o}}\,[\widetilde{T}]} &= m_{\mathsf{i}}\,[\widetilde{T}]\\ \overline{\emptyset[\widetilde{T}]} &= \emptyset[\widetilde{T}]\end{aligned}$$

The combination of types is defined as follows:

$$\begin{aligned}m_{\mathsf{o}}\,[\widetilde{T}] \uplus m_{\mathsf{i}}\,[\widetilde{T}] &\stackrel{\mathtt{def}}{=} m_{\sharp}\,[\widetilde{T}] &\\ T \uplus T &\stackrel{\mathtt{def}}{=} T & \text{if } \mathsf{un}(T)\\ T \uplus S &\stackrel{\mathtt{def}}{=} \mathtt{undef} & \text{otherwise}\end{aligned}$$

In particular, the second equation implies $\emptyset[\widetilde{T}] \uplus \emptyset[\widetilde{T}] = \emptyset[\widetilde{T}]$.

The lin and un predicates are again as follows:

$$\begin{array}{ll}\mathsf{lin}(T) & \text{if } T = \ell_\alpha\,[\widetilde{T}] \text{ or } (T = \langle l_i\_T_i\rangle_{i\in I} \text{ and } \exists j \in I.\ \mathsf{lin}(T_j))\\ \mathsf{un}(T) & \text{otherwise}\end{array}$$

Typing contexts The typing context Γ is defined for both the π-calculi as follows:

$$\Gamma ::= \emptyset \mid \Gamma, x : T$$

In addition to this typing context, we introduce a new typing context Θ, used to accommodate the recursion variables, namely:

$$\Theta ::= \emptyset \mid \Theta, X : \Gamma$$

Then, the typing judgements for both the π-calculi with recursion constructs have the form:

$$\Theta; \Gamma \vdash P$$

The generalisation of the lin and un predicates to typing contexts is the same as in the previous sections.

Type equality An important notion related to the recursive types is that of *type equality* denoted with $\sim_{\mathsf{type}}$. Following [101] we write $T_1 \sim_{\mathsf{type}} T_2$ to mean that the underlying (possibly infinite) trees of T_1 and T_2 are the same. To formalise it, we say that $\sim_{\mathsf{type}}$ is a congruence and satisfies the following:

$$\frac{}{\mu \mathbf{t}.T \sim_{\mathsf{type}} T[\mu \mathbf{t}.T/\mathbf{t}]}\ (\text{Eq-Unfold})$$

Typing rules for π-calculus with and without sessions The typing rule for the recursive process added both to the π-calculi with and without session is given in the following Fig. 10.3. The rest of the typing rules are the same as in Section 5.4 for the π-calculus with sessions and Section 4.4 for the standard π-calculus, respectively where the typing judgements are merely augmented with Θ.

$$\frac{\Theta(X) = \Gamma}{\Theta; \Gamma \vdash X}\ (\text{T}[\pi]\text{-RecVar}) \qquad \frac{\Theta, X : \Gamma; \Gamma \vdash P}{\Theta; \Gamma \vdash \mathbf{rec} X.P}\ (\text{T}[\pi]\text{-RecProc})$$

$$\frac{\Theta, \Gamma \vdash v : T \quad T \sim_{\mathsf{type}} S}{\Theta, \Gamma \vdash v : S}\ (\text{T}[\pi]\text{-EqVal})$$

Figure 10.3: Typing rules for recursive constructs

Rule (T[π]-RecVar) states that a process variable X is well-typed in $\Theta; \Gamma$ if it is assumed in Θ that X has 'type' Γ . Rule (T[π]-RecProc) states that the recursive process $\mathbf{rec}X.P$ is well-typed in $\Theta; \Gamma$ if process P is well-typed in a typing context where X is associated with Γ. Rule (T[π]-EqVal) is a subsumption rule for the equality relation $\sim_{\mathsf{type}}$ on infinite recursive types. It states that a value v is of type S if it has type T by the premise of the typing rule and $T \sim_{\mathsf{type}} S$.

10.4 Encoding

The encodings of recursive types, recursive processes and typing contexts, are given in Fig. 10.4.

The encoding of processes is the one presented in Section 6.2, with the addition of two equations for recursion, namely equation (E-Pvar) and equation (E-Prec). The encoding of process variable and recursive process is an homomorphism.

The encoding of types is a conservative extension of the one presented in Section 6.1, the latter being the encoding of only the linear pretypes, $\mathsf{lin}p$. Here, we give the encoding of both the linear and the unrestricted pretypes as well as the recursive types that we added at the beginning of this chapter, namely we encode the types produced by the syntax $T ::= q\ p \mid \mathsf{end} \mid \mathbf{t} \mid \mu \mathbf{t}.T$. The encoding of

Types Encoding:

$$
\begin{array}{rcll}
[\![\mathsf{end}]\!] & \stackrel{\mathsf{def}}{=} & \emptyset[] & \text{(E-qEnd)} \\
[\![q!T.U]\!] & \stackrel{\mathsf{def}}{=} & m_{\mathsf{o}}[[\![T]\!], [\![\overline{U}]\!]] & \text{(E-qOut)} \\
[\![q?T.U]\!] & \stackrel{\mathsf{def}}{=} & m_{\mathsf{i}}[[\![T]\!], [\![U]\!]] & \text{(E-qInp)} \\
[\![q \oplus \{l_i : T_i\}_{i\in I}]\!] & \stackrel{\mathsf{def}}{=} & m_o[\langle l_i \_ [\![\overline{T_i}]\!]\rangle_{i\in I}] & \text{(E-qSelect)} \\
[\![q\&\{l_i : T_i\}_{i\in I}]\!] & \stackrel{\mathsf{def}}{=} & m_i[\langle l_i \_ [\![T_i]\!]\rangle_{i\in I}] & \text{(E-qBranch)} \\
[\![\mathbf{t}]\!] & \stackrel{\mathsf{def}}{=} & \mathbf{t} & \text{(E-TVar)} \\
[\![\mu\mathbf{t}.T]\!] & \stackrel{\mathsf{def}}{=} & \mu\mathbf{t}.[\![T]\!] & \text{(E-TRec)}
\end{array}
$$

Terms Encoding:

$$
\begin{array}{rcll}
[\![X]\!]_f & \stackrel{\mathsf{def}}{=} & X & \text{(E-PVar)} \\
[\![\mathbf{rec}X.P]\!]_f & \stackrel{\mathsf{def}}{=} & \mathbf{rec}X.[\![P]\!]_f & \text{(E-PRec)}
\end{array}
$$

Typing Context Encoding:

$$
\begin{array}{rcll}
[\![\emptyset]\!]_f & \stackrel{\mathsf{def}}{=} & \emptyset & \text{(E-Empty)} \\
[\![\Gamma, x : T]\!]_f & \stackrel{\mathsf{def}}{=} & [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] & \text{(E-Gamma)} \\
[\![\Theta, X : \Gamma]\!]_f & \stackrel{\mathsf{def}}{=} & [\![\Theta]\!]_f, X : [\![\Gamma]\!]_f & \text{(E-Theta)} \\
[\![\Theta; \Gamma]\!]_f & \stackrel{\mathsf{def}}{=} & [\![\Theta]\!]_f; [\![\Gamma]\!]_f & \text{(E-CtxRec)}
\end{array}
$$

Figure 10.4: Encoding of recursive types, terms and typing contexts

linear pretypes is exactly as in Section 6.1, by letting the lin qualifier be interpreted as ℓ_α, where α is the action that follow the qualifier. The encoding of unrestricted pretypes follows the same idea as for the linear ones, by letting the un qualifier be interpreted as α, the latter being the action that follow the qualifier. Put together, the encoding of a q pretype is a channel type with action α and multiplicity m, linear or unrestricted, namely m_α. The encoding of recursive type constructs is an homomorphism and is given by equations (E-TVar) and (E-TRec), respectively for the recursive type variable $\mathbf{t}$ and for the recursive type $\mu\mathbf{t}.T$.

10.5 Properties of the Encoding

In this section we present the main properties related to the encoding of recursive types and terms added to both π- calculus with sessions and the standard one. Notice in the following that the typing judgements are different with respect to the ones in the original theorem, in that they are augmented with Θ to accommodate assumptions on recursive variables. However, this is an extension performed to every case of the original theorem.

We start with the correctness of the encoding for values. In particular, the following lemma is an extension of Lemma 6.3.8 and Lemma 6.3.9

Lemma 10.5.1 (Correctness of Typing Values)**.** $\Theta; \Gamma \vdash v : T$ *if and only if* $[\![\Theta; \Gamma]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$.

Proof. The only case to consider is rule (T[π]-EqVal), which is added on both calculus with and without sessions. Both directions of the lemma are trivial. □

The following theorem states the correctness of the encoding for typing derivations for processes: a recursive session process is well-typed if and only if the encoded recursive process is well-typed. Notice that this is a case accommodating recursion and is integrated to the main theorems on correctness of typing processes via encoding, namely Theorem 6.3.10 and Theorem 6.3.11.

Theorem 10.5.2 (Correctness of Typing Recursion)**.** $\Theta; \Gamma \vdash \mathbf{rec}X.P$ *if and only if* $[\![\Theta; \Gamma]\!]_f \vdash [\![\mathbf{rec}X.P]\!]_f$.

Proof. We split the proof as follows:

- (*only if*):
  This case is done by induction on the last typing rule applied for deriving $\Theta; \Gamma \vdash \mathbf{rec}X.P$. The last rule applied must be (T-RecProc). Hence, by induction hypothesis we have: $[\![\Theta]\!]_{f'}, X : [\![\Gamma]\!]_{f'}; [\![\Gamma]\!]_{f'} \vdash [\![P]\!]_{f'}$. By letting $f = f'$ and by applying rule (Tπ-RecProc) we obtain $[\![\Theta]\!]_f; [\![\Gamma]\!]_f \vdash [\![\mathbf{rec}X.P]\!]_f$.

- (*if*):
  This case is done by induction on the structure of the recursion process. By (E-PRec) we have $[\![\mathbf{rec}X.P]\!]_f \stackrel{\mathtt{def}}{=} \mathbf{rec}X.[\![P]\!]_f$ and assume that $[\![\Theta]\!]_f; [\![\Gamma]\!]_f \vdash [\![\mathbf{rec}X.P]\!]_f$. This means that the last rule applied must have been (Tπ-RecProc). As in the previous case, we conclude by induction hypothesis, letting $f' = f$ and applying (T-RecProc).

□

The following theorem states the operational correspondence for recursive processes. We first start with an auxiliary definition.

Definition 10.5.3. *Let Q be a session process and X be free in Q and let $Q[\mathbf{rec}X.Q/X]$ denote process Q where process variable X is substituted by $\mu X.Q$. Then,*

$$[\![Q[\mathbf{rec}X.Q/X]]\!]_f \stackrel{def}{=} [\![Q]\!]_f[\mathbf{rec}X.[\![Q]\!]_f/X]$$

We prove the following theorem which is an extension to the original operational correspondence theorem, in order to accommodate recursive processes.

Theorem 10.5.4 (OC for Recursive Process)**.** *Let P be a process in the π-calculus with sessions. Then,*

1. *If $P \to P'$ then $[\![P]\!]_f \to\equiv [\![P']\!]_f$,*
2. *If $[\![P]\!]_f \to\equiv Q$ then, $\exists\, P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \to^*\equiv [\![P']\!]_{f'}$, where f' is the updated f after the communication and $f_x = f_y$ for all $(\nu xy) \in \mathcal{E}[\cdot]$.*

Proof. The two cases are as follows

1. The proof is done by induction on the length of the derivation $P \to P'$.

   - Case (R-Rec):

   $$\frac{P[\mathbf{rec}X.P/X] \to P'}{P \to P'}$$

   By induction hypothesis we have that $[\![P[\mathbf{rec}X.P/X]]\!]_f \to\equiv [\![P']\!]_f$. By applying (R$\pi$-Rec) and (R$\pi$-Struct) we conclude that $[\![P]\!]_f \to\equiv [\![P']\!]_f$.

2. The proof is done by induction on the last reduction rule applied to obtain $[\![P]\!]_f \to\equiv Q$. The only case to be considered is the following:

   - Case (Rπ-Rec):
     We have that $[\![P]\!]_f \to R$ and $R \equiv Q$. By the premise of rule (Rπ-Rec) this means that $[\![P]\!]_f[\mathbf{rec}X.[\![P]\!]_f/X] \to R$, which by Definition 10.5.3 we have that $[\![P[\mathbf{rec}X.P/X]]\!]_f = [\![P]\!]_f[[\![\mathbf{rec}X.P]\!]_f/X]$, namely $[\![P[\mathbf{rec}X.P/X]]\!]_f \to\equiv Q$. We conclude by induction hypothesis and rule (R-Rec) and by letting $\mathcal{E}[\cdot] = [\cdot]$ and $Q \equiv [\![P']\!]_f$.

□

CHAPTER 11

From π-Types to Session Types

11.1 Further Considerations

As explained in the previous sections, a session type is interpreted as a linear channel type, which in turn carries a linear channel. In order to satisfy this linearity, a fresh channel is created at any step of communication and is sent to the communicating peer together with the message to be transmitted. The channel sent is then used to continue the rest of the communication. Session types are structured as opposed to standard π-types which are not. However, the creation and transmission of a channel simulates the structure of a session type.

There are two processes in the encoding presented in Chapter 6 that create a new channel, the output process and the selection process, the latter being a generalisation of the former. Namely

$$[\![x!\langle v\rangle.P]\!]_f \stackrel{\texttt{def}}{=} (\nu c) f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$$
$$[\![x \triangleleft l_j.P]\!]_f \stackrel{\texttt{def}}{=} (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$$

Once can argue that there is an overhead in creating at every output action a new channel to simulate the continuation of the communication. In the following, we show that the transmission of a new channels is not necessary. What we propose is to modify the encoding in order to mimic even more a session type. In this

optimised approach we reuse the same channel. But then, since channel variables have linear types, doing so would lead to a typing problem, since the process would not be well-typed, as it obviously violates linearity. In order to overcome this problem, we modify the typing rules as in the following paragraphs, for both the output and the selection processes.

Output Consider the output process $x!\langle v\rangle.P$ in the session π- calculus, which again is encoded as:

$$[\![x!\langle v\rangle.P]\!]_f \stackrel{\texttt{def}}{=} (\nu c) f_x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \tag{11.1}$$

The optimised encoding is as follows:

$$[\![x!\langle v\rangle.P]\!] \stackrel{\texttt{def}}{=} x!\langle v, x\rangle.[\![P]\!] \tag{11.2}$$

In order to overcome the linearity violation, we modify the type system by introducing the following typing rule for the output:

$$\frac{\Gamma_1 \vdash x : \ell_{\mathsf{o}}\,[\widetilde{T}] \qquad \widetilde{\Gamma_2}, x : \ell_\alpha\,[\widetilde{S}] \vdash \tilde{v} : \widetilde{T} \qquad \Gamma_3, x : \ell_{\overline{\alpha}}\,[\widetilde{S}] \vdash P}{\Gamma_1 \uplus \widetilde{\Gamma_2} \uplus \Gamma_3 \vdash x!\langle \tilde{v}\rangle.P}\ (\text{T}\pi\text{-OutBis})$$

The above typing rule states that the output process $x!\langle\tilde{v}\rangle.P$ is well-typed if The variable x is a linear channel used in output to transmit values of type $\widetilde{T}$, and the sequence of values $\tilde{v}$ is of the expected sequence of types $\widetilde{T}$. Notice that the typing context $\widetilde{\Gamma}$, differently from the original (Tπ- Out), is augmented with the type assumption of x having type $\ell_\alpha\,[\widetilde{S}]$. Since this is a linear type, it implies that $x \in \tilde{v}$. In addition, process P is well-typed under the assumption that x has the dual type of the type it has when transmitted, namely $\ell_{\overline{\alpha}}\,[\widetilde{S}]$.

Selection Consider the selection process $x \triangleleft l_j.P$ in the session π- calculus, which again is encoded as:

$$[\![x \triangleleft l_j.P]\!]_f \stackrel{\texttt{def}}{=} (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \tag{11.3}$$

The optimised encoding is as follows:

$$[\![x \triangleleft l_j.P]\!] \stackrel{\texttt{def}}{=} x!\langle l_j\_x\rangle.[\![P]\!] \tag{11.4}$$

By using (Tπ-LVAL)

$$\frac{\Gamma, x : \ell_\alpha\ [\widetilde{S}] \vdash x : \ell_\alpha\ [\widetilde{S}] = T_j \quad j \in I}{\Gamma, x : \ell_\alpha\ [\widetilde{S}] \vdash l_j\_x : \langle l_i\_T_i \rangle_{i \in I}} \text{ (T}\pi\text{-LVAL)}$$

And using (Tπ-OUTBIS), we type the encoding of the selection process.

Notice that the encoding of session types remains as in Fig. 6.1, and the encoding of session processes remains as in Fig. 6.2, except for equations 11.2 and 11.4 which substitute respectively 11.1 and 11.3.

11.2 Typed Behavioural Equivalence

In this section we show that 11.1 and 11.2 as well as 11.3 and 11.4 are *typed strong barbed congruent*. We first give a few definitions, taken from [101], that can lead us to our result. We start with the following two auxiliary definitions:

Definition 11.2.1 (Context)**.** *A* context *in the π-calculus is obtained when the hole* $[\cdot]$ *replaces an occurrence of the terminated process* **0** *in a process term produced by the grammar in Section 4.1.*

Definition 11.2.2 (Strong Barbed Bisimilarity)**.** Strong barbed bisimilarity *is the largest, symmetric relation $\sim$ such that if whenever $P \sim Q$,*

1. *If P performs an input/output action with subject x, then Q also performs an input/output action with subject x.*
2. *$P \to P'$ implies $Q \to Q'$ for some process Q' with $P' \sim Q'$.*

Two processes P, Q are strong barbed bisimilar *if $P \sim Q$.*

Definition 11.2.3 (Strong Barbed Congruence)**.** *Two processes are* strong barbed congruent *if they are strong barbed bisimilar for every arbitrary context they are placed into.*

We pass now from the definition of strong barbed congruence to the typed version of it.

Definition 11.2.4 (Typed Strong Barbed Congruence)**.** *Let $\Delta \vdash P$ and $\Delta \vdash Q$. We say that processes P, Q are* strong barbed congruent at Δ*, denoted $\Delta \rhd P \simeq^{c} Q$, if they are strong barbed congruent for every* (Γ/Δ)*-context, with* Γ *closed.*

We explain intuitively a (Γ/Δ)-context. We refer to [101] for the formal definition. A (Γ/Δ)-context, when filled with a well-typed process in Δ becomes a well-typed process in Γ.

An important result, which will act as a proof technique in the following, is the Context Lemma for the typed strong barbed congruence.

Definition 11.2.5. *Suppose $\Delta \vdash P$ and $\Delta \vdash Q$. We write $\Delta \rhd P \simeq^{\mathsf{s}} Q$ if for every closed Γ that extends Δ, for every Δ-to-Γ substitution σ and every process R such that $\Gamma \vdash R$, it holds that $R \mid \sigma(P)$ is strong barbed bisimilar to $R \mid \sigma(Q)$.*

Lemma 11.2.6 (Context Lemma)**.** *Suppose $\Delta \vdash P$ and $\Delta \vdash Q$. $\Delta \rhd P \simeq^{\mathsf{s}} Q$ if and only if $\Delta \rhd P \simeq^{\mathsf{c}} Q$.*

11.2.1 Equivalence Results for the Encoding

We are ready now to present the result on typed strong barbed congruence of the encoding of the output and the selection processes. They are as follows.

Output Let

$$\Gamma = x : \ell_{\mathsf{o}}\,[T, \ell_\alpha\,[\widetilde{S}\,]], v : T$$

$$P_1 = (\boldsymbol{\nu} c)x!\langle v, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$$

$$P_2 = x!\langle v, x\rangle.[\![P]\!]$$

then

$$\Gamma \rhd P_1 \simeq^{\mathsf{c}} P_2 \tag{11.5}$$

Selection Let

$$\Gamma = x : \ell_{\mathsf{o}}\,[\langle l_i\_T_i\rangle_{i\in I}]$$

$$P_1 = (\boldsymbol{\nu} c)x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$$

$$P_2 = x!\langle l_j\_x\rangle.[\![P]\!]$$

then

$$\Gamma \rhd P_1 \simeq^{\mathsf{c}} P_2 \tag{11.6}$$

Both 11.5 and 11.6 follow from applying the Subject Reduction Theorem 4.5.2 and the Context Lemma 11.2.6.

Conclusions, Related and Future Work for Part II and III

In Part II and III of this dissertation, we proposed an interpretation of session types into ordinary π-types, more precisely into *linear channel types* and *variant types*.

Linearity is a concept widely used in various areas of computer science. Intuitively, when linearity of a resource is enforced, it means that the resource is used *exactly* once, namely it cannot be used more than once and on the other hand it must be used at least once. Linear channel types [70, 101] assure that a channel is used exactly once for communication.

Variant types [101] are a labelled form of disjoint union of types, where the order of components does not matter and labels are all distinct. Variant types come together with specific terms in the calculus, in particular the *variant values* and the **case** process. A variant value is a labelled value and a **case** process is a process construct native of the standard π-calculus. The branching and selection processes in the session π-calculus are similar and are inspired by the **case** process, in that they offer a sequence of labelled processes from which the communicating party can choose.

In Part II we developed Kobayashi's proposal of an encoding of session types into into *linear channel types* and *variant types*. We showed that the encoding is *faithful*, in that it allows us to derive all the basic properties of session types, by exploiting the analogous properties of π- types. In Part III we showed that the encoding is *robust*, by analysing a few non-trivial extensions to session types, namely subtyping, polymorphism and higher-order. Finally, we proposed an op-

timisation of linear channels permitting the reuse of the same channel for the continuation of the communication and proved a typed barbed congruence result. This optimisation considerably simplifies the encoding, which is parametrised in function f which on some terms, like in input and output processes becomes the identity function. The encoding of session types, however is the same as before.

Contribution The encoding we presented in Part II and III has several benefits. We list them in the following.

- The elimination of the redundancy introduced both in the syntax of types and in the syntax of terms.
- The derivation of properties like Subject Reduction and Type Safety as straightforward corollaries, thus eliminating redundancy also in the proofs.
- Privacy, communication safety and session fidelity requirements in session types are handled by the check of linearity and the encoding in the standard typed π-calculus. Duality boils down to opposite outermost capabilities of linear channel types.
- The encoding is robust with respect to extensions like subtyping, polymorphism, higher-order and recursion. This permits to prove properties of extension typing features by exploiting the well-established theory of the standard typed π-calculus.

As the last point states, the encoding allows us to easily obtain extensions of the session calculus, by exploiting the theory of the π-calculus. In particular, as shown in Section 8.2 about the bounded polymorphism, our approach makes it easy even when the intended extension was not already present in the π-calculus. In these cases one can just provide the π-calculus with the intended capability and obtain the same capability in sessions. The whole process has shown to be much easier passing through π-calculus than doing it from scratch for sessions.

We conclude that session types theory is indeed derivable from the theory of π calculus. This does not mean that we believe session types are useless: on the contrary, due to their simple and intuitive structure they represent a sophisticated tool for describing and reasoning about communication protocols in distributed scenarios. Our aim was to provide a methodology for facilitating the definition of session types and their extensions, hence encouraging their study.

Related and Future Work The idea of encoding session types into linear π-types is not new. Kobayashi [66, 69] proposes such an encoding, but he did not provide any formal study of it. Demangeon and Honda [33] provide a subtyping theory for a π-calculus augmented with branch and select constructs and show an encoding of the session calculus. They prove the soundness of the encoding and the full abstraction. The main differences with respect to our work are: i) the target language is closer to the session calculus having branch and select constructs, instead we adopt the standard π-calculus where in place of branching and selection we provide the native **case** process and in place of the branch and select type we provide the standard variant type; ii) a refined subtyping theory is provided, instead we focus on encoding of the session calculus in the standard π-calculus in order to exploit its rich and well-established theory; iii) we study the encoding in a systematic way as a means to formally derive session types and all their properties, in order to provide a methodology for the treatment of session types and their extensions without the burden of establishing the underlying theory.

As long as variant types are concerned, our encoding shows that they are an essential native type construct in the typed π-calculus. This has been proved also by other works on encodings where variant types have been used, in particular, we mention the encoding of a typed object-oriented calculus into the typed π-calculus with variant types [99].

Other expressivity results regarding session types include the work by Caires and Pfenning [16]. In this paper, the authors present a type system for the π-calculus that corresponds to the standard sequent calculus proof system for dual intuitionistic linear logic (DILL). They give an interpretation of intuitionistic linear logic formulas as a form of session types. These results are complemented and strengthened with a theory of logical relations [96]. An interpretation of the simply-typed λ-calculus in the π-calculus with session is given in [105]. As stated by the authors this encoding is done in two steps: first by giving a standard embedding of simply-typed λ-calculus in a linear λ-calculus and second by a translation of linear natural deduction into linear sequent calculus.

Another work on expressivity is the one by Wadler [113], which follows the line of [16]. In this paper, the author proposes a calculus where propositions of classical linear logic correspond to session types.

Igarashi and Kobayashi [57] have developed a single generic type system (GTS) for the π-calculus from which numerous specific type systems can be obtained by varying certain parameters. A range of type systems are thus obtained as instances of the generic one. In [45] the authors define an interpretation from

session types and terms into GTS by proving operational correspondence and correctness of the encoding. However, as the authors state, the encoding they present is very complex and deriving properties of sessions passing through GTS would be more difficult than proving them directly. Instead, the encoding we present is very simple and properties of sessions are derived as straightforward corollaries from the corresponding ones in the π-calculus.

All the above works are clearly an expressivity result. The encoding we propose is an expressivity result, as well. However, in addition our encoding is a powerful means for deriving the theory of session types and its possible extensions by the well-known theory of the standard π-calculus.

As long as future work on the encoding is concerned we want to investigate a major topic, that of multiparty session types [56]. In a nutshell, multiparty session types differ from dyadic session types in the order in which session channels are used. In a dyadic scenario, the participants share one channel, by owning the opposite endpoints. In a multiparty scenario, due to the presence of many participants, various session channels are used and hence the order in which these channels are used is important to guarantee communication safety and session fidelity. Our encoding should be extended in order to accommodate this notion of *causality* of channels introduced in [56].

Part IV

Progress of Communication

Introduction to Part IV

The notion of *progress* is a fundamental characteristic of safe programs in a language model. Intuitively, it means that a safe program never gets "stuck", i.e., reach a state that is not designated as a final value and the semantics of the language does not tell how to evaluate further [97].

The notion of progress is well-understood in models such as the λ-calculus [6] and it is typically analysed in closed terms through type systems. On the other hand, we have only recently begun to scratch the surface of its meaning in models for concurrency and distributed systems.

The most basic property related to progress in concurrency is *deadlock-freedom*: "a process is deadlock-free if it can always reduce until it eventually terminates" [65, 68]. Observe that a deadlock-free process can diverge. So, said differently, a communication will eventually succeed unless the whole process diverges. Also, and more interestingly, in a deadlock-free process some subprocesses can get stuck. For instance, consider the following process:

$$P \;=\; (\boldsymbol{\nu} x)(x?(y).\mathbf{0} \mid \Omega)$$

where Ω is a diverging process executing an infinite series of internal actions. Even though the subterm $x?(y).\mathbf{0}$ will never reduce, process P is deadlock-free.

In order to cope with this limitation of the deadlock-freedom property, *lock-freedom* or *livelock-freedom* has been proposed as a stronger property that requires every input/output action to be eventually executed under fair process scheduling [65]. Said differently, a communication will eventually succeed even if the

whole process diverges. Different techniques have been proposed for guaranteeing deadlock- and lock-freedom, mostly based on type systems [22,65,68,72,89].

All the aforementioned techniques are applied to closed processes, i.e., processes that do not communicate with the environment. However, a useful application of process calculi is to model open-ended systems where participants can join the system dynamically [35, 86, 89, 92]. A recent line of work has begun investigating the meaning of progress for such open-ended systems. Intuitively, in this setting a process has the progress property if it can reduce when it is put in execution in a suitable context. This notion has been analysed when considering only the behaviour of each single channel in isolation [31, 56, 109] and of the whole system [20, 27] in the context of session types.

We observe that progress in open-ended systems is a *compositional* notion, since an open process that has progress can be composed with another compatible process to obtain a system that does not get stuck. Interestingly, this compositionality seems to lead back to the notion of lock-freedom, in that both notions inspect subprocesses of a system. Thus, we ask:

What is the relationship between the notions of lock-freedom and progress for open-ended systems?

Answering the question above would lead to a better understanding of the progress property in concurrency. Ideally, it would allow techniques and results obtained for one property to be applied to the other.

In the following we list the major contributions of Part IV.

Progress in the π-calculus with sessions We discuss the relationship between progress and lock-freedom in the setting of π-calculus with sessions (Section 14.3), by studying the properties of processes that are well-typed in the standard type system based on session types given by Vasconcelos [109]. Our first result is that for well-typed processes with no open sessions (closed processes), the progress and lock-freedom properties coincide: a well-typed closed process has progress if and only if it is lock-free (Section 14.3.1). Building on top of this result, we prove that it is possible to relate progress to lock-freedom even in the setting of processes with open sessions (Section 14.3.2): a well-typed process has progress if and only if it can be put in a context such that the composition is a well-typed closed process and lock-free. In other words we prove that, in the setting of well-typed processes in the π-calculus with sessions, *progress is a compositional form of the notion of lock-freedom*. Crucial to our development is the

definition of a new "closure" procedure for generating well-typed contexts that are guaranteed not to introduce locks, which exploits the types of the process for which the context is generated.

A static analysis for progress in the π-calculus with sessions Guided by the discovery that progress is related to lock-freedom, we show that it is possible to build a static analysis for progress in the π-calculus with sessions by reusing an analysis for lock-freedom in the standard π-calculus. Specifically, we present how Kobayashi's type system for lock-freedom [65] can be reused for establishing whether a process has progress. Reusing Kobayashi's type system for progress analysis yields a new powerful technique that captures new processes that have progress that could not be recognised by previous techniques.

Roadmap to Part IV The rest of Part IV is organised as follows. Chapter 12 gives a background on the standard π-calculus by focusing in particular in the syntax of types and in the type system for guaranteeing the lock-freedom property. Chapter 13 gives a background on the π-calculus with sessions which reports few modifications with respect to the one introduced in Part II: it includes recursion and recursive types and the choice operator is enhanced to accommodate the progress property. Chapter 14 introduces the notion of progress for the π-calculus with session, by relating it to the notion of lock-freedom for sessions. In addition it gives a static way for checking progress for sessions, by using the type system for lock-freedom given in Chapter 12.

CHAPTER 12

Background on π-types for Lock-Freedom

In this chapter we introduce Kobayashi's type system for lock-freedom [65, 67]. We start by presenting the syntax of terms and the operational semantics for the standard π-calculus, which are the same as in Chapter 4. However, for the sake of readability, we recall them briefly in this chapter. Then we introduce a new syntax of types: the *usage types*. To conclude, we give the type system with usage types which checks the lock-freedom property.

12.1 Syntax

The syntax of terms in the standard (polyadic) π-calculus is basically the same as in Section 4.1 and is given in Fig. 12.1. Processes include the output $x!\langle\tilde{v}\rangle.P$ and the input $x?(\tilde{y}).P$ processes, where a tuple of values is transmitted; conditional **if** v **then** P **else** Q; standard constructs as parallel composition $P \mid Q$, inaction $\mathbf{0}$ and restriction $(\nu x)P$; the **case** process and the new constructs wrt Section 4.1 are the process variable X and the recursive process $\mathbf{rec}X.P$. Values include variables ranged by x, ground values, in particular the boolean ones `true` and `false`, and variant value or labelled value l_v.

$P, Q ::=$	$x!\langle\tilde{v}\rangle.P$	output
	$x?(\tilde{y}).P$	input
	if v **then** P **else** Q	conditional
	$P \mid Q$	composition
	$\mathbf{0}$	inaction
	$(\nu x)P$	channel restriction
	case v **of** $\{l_i\_x_i \triangleright P_i\}_{i\in I}$	case
	X	process variable
	rec$X.P$	recursive process
$v ::=$	x	variable
	`true | false`	boolean values
	l_v	variant value

Figure 12.1: Syntax of the standard π-calculus

12.2 Semantics

The (most important) reduction rules are presented in Fig. 12.2. They are a combination of the rules presented in Section 4.2 and in Section 10.2. We do not present the reduction rules for context closure under composition, restriction and structural congruence, the latter being the standard one.

$$\text{(R}\pi\text{-Com)} \qquad x!\langle\tilde{v}\rangle.P \mid x?(\tilde{z}).Q \to P \mid Q[\tilde{v}/\tilde{z}]$$

$$\text{(R}\pi\text{-Case)} \quad \textbf{case}\, l_j\_v\, \textbf{of}\, \{l_i\_x_i \triangleright P_i\}_{i\in I} \to P_j[v/x_j] \quad j \in I$$

$$\text{(R}\pi\text{-Rec)} \qquad \frac{P[\textbf{rec}X.P/X] \to P'}{\textbf{rec}X.P \to P'}$$

Figure 12.2: Semantics of the standard π-calculus

12.3 Types for Lock-Freedom

The syntax of types is given in Fig. 12.3 and is inspired by Kobayashi's works on lock-freedom [65, 67]. Let α range over actions, U over usages and T over types. An action α can be an input action '`i`', or an output action '`o`', (we omit the connection action $\sharp$, since it can be simulated by `|`, as explained in the following). Usages U build up channel types. A usage can be an empty usage $\emptyset$, denoting a channel that cannot be used at all for communication, in the same way as in

$\alpha ::=$	i	input action
	o	output action
$U ::=$	$\emptyset$	not usable
	$\alpha_c^o.U$	used once for α and then for U
	$(U_1 \mid U_2)$	used in parallel
	$\mathbf{t}$	usage variable
	$\mu\mathbf{t}.U$	recursive usage
$m\,U ::=$	ℓ_U	linear usage
	U	unrestricted usage
$T ::=$	$[\widetilde{T}]\,m\,U$	channel types
	$\langle l_i\_T_i\rangle_{i\in I}$	variant type
	$\mathtt{Bool}$	boolean type
	$\cdots$	other constructs
o, c	$\in Nat$	o, c in natural numbers

Figure 12.3: Syntax of usage types

Section 4.3; we will often omit it when not necessary. Usage $\alpha_c^o.U$ describes a channel used once for input or output, depending on the action α, and then used according to U. The annotations o and c in the action α are natural numbers and are called *obligation level* and *capability level* of an action, respectively. We will commonly refer to them as *tags* or *attributes* and we will comment on these numbers in the following. Usage $U_1 \mid U_2$ describes a channel used according to U_1 and U_2 in parallel. Usage variable $\mathbf{t}$ is combined with the recursive usage $\mu\mathbf{t}.U$ which is used according to $U[\mu\mathbf{t}.U/\mathbf{t}]$. A type T can be a channel type $[\widetilde{T}]\,mU$, used according usage U to transmit a sequence of values of types $\widetilde{T}$. As we can see, the usages describe a channel used in structured way, which recalls the session types. However, the main difference with session types is that the carried type associated to a usage is always the same $\widetilde{T}$. In the same way as for session types, where one has linear or unrestricted pretypes, we associate to a usage U the qualifier ℓ for linear usage, otherwise the usage is unrestricted. We use mU to range over linear usages ℓ_U or unrestricted ones U. A type can also be variant type $\langle l_i\_T_i\rangle_{i\in I}$ or a ground type like $\mathtt{Bool}$. Other type constructs, like $\mathtt{Int}, \mathtt{String}, \ldots$, or product types or record types etc., can be added to the syntax of types, in the same way as stated in Section 4.3.

Let us now explain the meaning of tags. They are thought of and defined as abstract representations of *time tags* or *reduction steps* and are found in [65, 67, 68].

The reason for this vague interpretation of tags is that what matters is their *relative* meaning and how tags are ordered among them, rather than their *absolute* meaning. They capture the inter-channel dependencies in communications. Intuitively, the obligation level o of an action (input or output) denotes the *necessity* of the action to be executed, namely when the action is ready to be performed; the capability level c of an action denotes the *guarantee* for success of the action, namely how long does it take for the action to find its co-action. For example, suppose that a channel has usage $\mathsf{i}^o_c.U$. Its obligation o means that a process can perform actions having capability levels *less than* o; said differently, the process becomes ready to use this channel for input within o steps, or time o. Instead, its capability c means that the success of the input on this channel is guaranteed if the corresponding co-action has an obligation level *less than or equal to* c; said differently, it succeeds to find its co-action in at most c steps, or time c. It is important to notice that in the original works tags may range also over ∞. It means that the success of the action is not guaranteed, or even that the action itself need not be executed at all. In this work, since we are considering processes that correspond to the encoding of a session process and we want that every action eventually takes place and succeeds, we exclude our tags to range over ∞. The relation between obligations and capabilities in a process is: (i) the obligation of an action is smaller than or equal to the capability of its co-action; (ii) the obligation of an an action is greater than the capabilities of every action prefixing it. We illustrate the usage of tags with two examples, reported below. The first example shows how tags work on a deadlocked process and the second example shows how tags work on a livelocked process.

Example 12.3.1. The process $(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(x?().y!\langle\rangle \mid y?().x!\langle\rangle)$ is deadlocked. Suppose that the usages are $\mathsf{i}^{o_1}_{c_1} \mid \mathsf{o}^{o_2}_{c_2}$ for x and $\mathsf{i}^{o_3}_{c_3} \mid \mathsf{o}^{o_4}_{c_4}$ for y. Since $x?()$ must wait for the corresponding output $x!\langle\rangle$ to be executed, it must be the case that $o_2 \leq c_1$; for the same reason $o_4 \leq c_3$. Moreover, from the left part of the parallel composition we know that y is used for output only after the input on x succeeds, which yields $c_1 < o_4$; for the same reason $c_3 < o_2$. From these inequations we have $o_2 \leq c_1 < o_4 \leq c_3 < o_2$, which is a contradiction.

Example 12.3.2. The following process is deadlock-free but livelocked: $(\boldsymbol{\nu} x)(x?(w) \mid (\boldsymbol{\nu} y)(y!\langle x\rangle \mid y?(z).\mathbf{rec}X.(y!\langle z\rangle \mid y?(z).X)))$ Suppose y sends x having usage $\mathsf{o}^{o_1}_{c_1}$. For a message sent on y it takes to be received by its counterpart $o_3 (> 0)$ steps. The subprocess $y?(z).\mathbf{rec}X.(y!\langle z\rangle \mid y?(z).X)$ receives z of usage $\mathsf{o}^{o_1}_{c_1}$ and so it is supposed to use it in time o_1 for output. Then, z is resent again on y

which means it needs o_3 steps to be received, as previously stated. So, it means that $o_1 + o_3 \leq o_1$, which is a contradiction. Informally speaking, this example shows a process which is never stuck, because of infinite sendings, however the first input on x will be never executed.

12.4 Typing Rules for Lock-Freedom

In this section we present the type system for lock-freedom which is an extension of the type system in [65], since that latter does not include **case** process or variant values. Before doing so we give few auxiliary definitions that lead to the definition of lock-freedom for the standard π-calculus taken from [65].

Definition 12.4.1 (Reduction Sequence)**.** *A set of processes $\{P_i\}_{i\in I}$ for $I \subseteq Nat$ is called a* reduction sequence *and is denoted as $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots$, if $P_{i-1} \rightarrow P_i \; \forall i \in I \setminus \{0\}$.*
A reduction sequence is normal *if i) $\forall i \in I \; P_i$ is in normal form and ii) the sequence of the restricted channels of P_{i-1} is a prefix of the sequence of the restricted channels of P_i.*
A reduction sequence is complete *if either $I = Nat$ or $I = [n]$ and $P_n \not\rightarrow$.*

Definition 12.4.2 (Fair Reduction Sequence)**.** *A normal, complete reduction sequence $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots$ is* fair *if the following conditions hold.*

1. *If there exists an infinite increasing sequence $n_0 < n_1 < \cdots$ of natural numbers such that $P_{n_j} \equiv (\nu\widetilde{x_j})(x!\langle v\rangle.Q \mid x?(z).Q_j \mid R_j)$, $\forall n_j$, then there exists $n \geq n_0$ such that $P_n \equiv (\nu\widetilde{x})(x!\langle v\rangle.Q \mid x?(z).Q' \mid R')$ and $(\nu\widetilde{x})(Q \mid Q'[v/z] \mid R') \equiv P_{n+1}$.*

2. *If there exists an infinite increasing sequence $n_0 < n_1 < \cdots$ of natural numbers such that $P_{n_j} \equiv (\nu\widetilde{x_j})(x?(z).Q \mid x!\langle v\rangle.Q_j \mid R_j)$, $\forall n_j$, then there exists $n \geq n_0$ such that $P_n \equiv (\nu\widetilde{x})(x?(z).Q \mid x!\langle v\rangle.Q' \mid R')$ and $(\nu\widetilde{x})(Q[v/z] \mid Q' \mid R') \equiv P_{n+1}$.*

We are ready now to give the definition of the lock-freedom property in the standard π-calculus.

Definition 12.4.3 (Lock-Freedom for Standard π)**.** *A process P_0 is* lock-free *under fair scheduling, if for any fair reduction sequence $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots$ the following hold*

1. *if* $P_i \equiv (\boldsymbol{\nu}\tilde{x})(x!\langle v\rangle.Q \mid R)$ *(for* $i \geq 0$*), implies that there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\tilde{x})(x!\langle v\rangle.Q \mid x?(z).R_1 \mid R_2)$ *and* $P_{n+1} \equiv (\boldsymbol{\nu}\tilde{x})(Q \mid R_1[v/z] \mid R_2)$*;*

2. *if* $P_i \equiv (\boldsymbol{\nu}\tilde{x})(x?(z).Q \mid R)$ *for some* $i \geq 0$*, there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\tilde{x})(x?(z).Q \mid x!\langle v\rangle.R_1 \mid R_2)$ *and* $P_{n+1} \equiv (\boldsymbol{\nu}\tilde{x})(Q[v/z] \mid R_1 \mid R_2)$

Before commenting on the typing rules we first explain the following operations on types.

The unary operation $\uparrow^t$ applied on a usage U lifts its obligation level *up to t*, and is defined inductively as:

$$\begin{aligned}
\uparrow^t \emptyset &= \emptyset \\
\uparrow^t \alpha_c^o.U &= \alpha_c^{max(o,t)}.U \\
\uparrow^t (U_1 \mid U_2) &= (\uparrow^t U_1 \mid \uparrow^t U_2) \\
\uparrow^t \mathbf{t} &= \mathbf{t} \\
\uparrow^t \mu\mathbf{t}.U &= \mu\mathbf{t}.\uparrow^t U
\end{aligned}$$

The $\uparrow^t$ is extended to types and typing context as follows:

$$\begin{aligned}
\uparrow^t [\widetilde{T}]\, m\, U &= [\widetilde{T}]\, m \uparrow^t U \\
\uparrow^t T &= \texttt{undef} \text{ otherwise} \\
(\uparrow^t \Gamma)(x) &= \uparrow^t (\Gamma(x))
\end{aligned}$$

The binary operation $\mid$ applied on usages U_1 and U_2 returns the usage $U_1 \mid U_2$. It is extended to types and typing context as follows:

$$\begin{aligned}
\texttt{Bool} \mid \texttt{Bool} &= \texttt{Bool} \\
[\widetilde{T}]\, m\, U_1 \mid [\widetilde{T}]\, m\, U_2 &= [\widetilde{T}]\, m\, (U_1 \mid U_2) \\
\langle l_1_T_1 \ldots l_n_T_n\rangle \mid \langle l_1_T_1 \ldots l_n_T_n\rangle &= \langle l_1_T_1 \ldots l_n_T_n\rangle \\
T \mid T' &= \texttt{undef} \text{ otherwise}
\end{aligned}$$

$$x : T \in \Gamma_1 \mid \Gamma_2 \text{ iff } \begin{cases} x : T_1 \in \Gamma_1 \text{ and } x : T_2 \in \Gamma_2 \\ \qquad\qquad \text{and } T = T_1 \mid T_2 \\ x : T \in \Gamma_1 \text{ and } x \notin \mathsf{dom}(\Gamma_2) \\ x : T \in \Gamma_2 \text{ and } x \notin \mathsf{dom}(\Gamma_1) \end{cases}$$

Notice that the parallel operator $\mid$ is defined similarly to the combination operator

$\uplus$ given in Section 4.4. As previously stated, we remove the connection $\sharp$ from the syntax of actions as it is simulated by $\mid$ present in the syntax of usages. In particular, $\sharp$ and $\mid$ on types (as well as $\uplus$ and $\mid$ on typing contexts) denote channels capable of both input and output actions *possibly* in parallel.

The operator $\dagger$ is defined on typing contexts. $\Delta = x : [T]\ \alpha_c^o \dagger \Gamma$ is such that the following holds:

$$\mathsf{dom}(\Delta) = \{x\} \cup \mathsf{dom}(\Gamma)$$

$$\Delta(x) = \begin{cases} [\widetilde{T}]\alpha_c^o.U & \text{if } \Gamma(x) = [\widetilde{T}]\ U \\ [\widetilde{T}]\alpha_c^o & \text{if } x \notin dom(\Gamma) \end{cases}$$

$$\Delta(y) = \uparrow^{c+1} \Gamma(y) \qquad \text{if } y \neq x$$

The last auxiliary notion we introduce before the typing rules is the *reliability* of a usage U, which is given by the predicate $rel(U)$. For a formal definition of the reliability predicate we refer to [65, 68]. Here we explain it in a more intuitive and informal way. A usage U is said to be reliable, denoted with $rel(U)$, if after any reduction step, whenever it contains an action (input or output) having capability level c, it also contains the co-action with an obligation level at most c. In particular, when U is a parallel usage, the predicate $rel(U)$, checks whether the obligations and capabilities of actions in U respect the conditions previously stated in a 'crossed' way. This check is performed by the type system as we will see in the following.

The typing rules are reported in Fig. 12.4. The typing judgements are of the form $\Gamma \vdash_{\mathrm{LF}} v : T$ for values and $\Gamma \vdash_{\mathrm{LF}} P$ for processes. We use $\vdash_{\mathrm{LF}}$ instead of just $\vdash$ in order to distinguish the type system for lock-freedom from the linear type system for the standard π-calculus. Rules (LF-Var), (LF-Val) and (LF-LVal) for values are the same as the corresponding ones in Section 4.4, where linear channel types are used. Rules (LF-Inact), (LF-If),(LF-Par) and (LF-Case) are the same as the corresponding ones in Section 4.4, where instead of the $\uplus$ operator on linear types we use the $\mid$ operator on usages. Rule (LF-In) states that the input process $x?(\tilde{y}).P$ is well-typed if x is a channel used in input with obligation level 0. Moreover, the obligations of other channels in Γ are raised by the auxiliary operator $\dagger$, to reflect in the types that the actions inside process P are prefixed by the input action that is being typed and will thus become available later. Rule (LF-Out) states that the output process $x!\langle\tilde{v}\rangle.P$ is well-typed and ready for execution if x is a channel used in output and has obligation level 0. Moreover, the obligation level

$$\frac{\mathsf{un}(\Gamma)}{\Theta;\Gamma, x : T \vdash_{\mathrm{LF}} x : T}\ (\text{LF-VAR}) \qquad \frac{\mathsf{un}(\Gamma) \quad v = \mathtt{true} \,/\, \mathtt{false}}{\Theta;\Gamma \vdash_{\mathrm{LF}} v : \mathtt{Bool}}\ (\text{LF-VAL})$$

$$\frac{\Theta;\Gamma \vdash_{\mathrm{LF}} v : T}{\Theta;\Gamma \vdash_{\mathrm{LF}} l\_v : \langle l\_T \rangle}\ (\text{LF-LVAL}) \qquad \frac{\mathsf{un}(\Gamma)}{\Theta;\Gamma \vdash_{\mathrm{LF}} \mathbf{0}}\ (\text{LF-INACT})$$

$$\frac{\Theta;\Gamma_1 \vdash_{\mathrm{LF}} v : \mathtt{Bool} \quad \Theta;\Gamma_2 \vdash_{\mathrm{LF}} P \quad \Theta;\Gamma_2 \vdash_{\mathrm{LF}} Q}{\Theta;\Gamma_1 \mid \Gamma_2 \vdash_{\mathrm{LF}} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}\ (\text{LF-IF})$$

$$\frac{\Theta;\Gamma, \widetilde{y} : \widetilde{T} \vdash_{\mathrm{LF}} P}{\Theta; x : [\widetilde{T}]\, m\, \mathsf{i}^0_c \ \dagger\ \Gamma \vdash_{\mathrm{LF}} x?(\tilde{y}).P}\ (\text{LF-IN}) \qquad \frac{\Theta;\Gamma_1 \vdash_{\mathrm{LF}} \tilde{v} :\uparrow \widetilde{T} \quad \Theta;\Gamma_2 \vdash_{\mathrm{LF}} P}{\Theta; x : [\widetilde{T}]\, m\, \mathsf{o}^0_c \dagger (\Gamma_1 \mid \Gamma_2) \vdash_{\mathrm{LF}} x!\langle\tilde{v}\rangle.P}\ (\text{LF-OUT})$$

$$\frac{\Theta;\Gamma_1 \vdash_{\mathrm{LF}} P \quad \Theta;\Gamma_2 \vdash_{\mathrm{LF}} Q}{\Theta;\Gamma_1 \mid \Gamma_2 \vdash_{\mathrm{LF}} P \mid Q}\ (\text{LF-PAR}) \qquad \frac{\Theta;\Gamma, x : [\widetilde{T}]\, m_U \vdash_{\mathrm{LF}} P \quad rel(U)}{\Theta;\Gamma \vdash_{\mathrm{LF}} (\nu x)P}\ (\text{LF-RES})$$

$$\frac{\Theta;\Gamma_1 \vdash_{\mathrm{LF}} v : \langle l\_T \rangle_{i\in I} \quad \Theta;\Gamma_2, x_i : T_i \vdash_{\mathrm{LF}} P_i \quad \forall i \in I}{\Theta;\Gamma_1 \mid \Gamma_2 \vdash_{\mathrm{LF}} \mathbf{case}\ v\ \mathbf{of}\ \{l_i\_x_i \triangleright P_i\}_{i\in I}}\ (\text{LF-CASE})$$

$$\frac{\Theta(X) = \Gamma}{\Theta;\Gamma \vdash X}\ (\text{LF-RECVAR}) \qquad \frac{\Theta, X : \Gamma;\Gamma \vdash P}{\Theta;\Gamma \vdash \mathbf{rec}X.P}\ (\text{LF-RECPROC})$$

$$\frac{\Theta;\Gamma \vdash v : T \quad T \sim_{\mathsf{type}} S}{\Theta;\Gamma \vdash v : S}\ (\text{LF-EQVAL})$$

Figure 12.4: Typing rules for π calculus with usage types

of the values $\tilde{v}$ is decremented by 1, by applying the operation $\uparrow \widetilde{T}$ in the premise of the rule: this is to reflect the fact that the actions on these values will become available one time step later, since they have to be transmitted first through the output action that is being typed. Finally, the obligations of channels in $\Gamma_1 \mid \Gamma_2$ are raised by $\dagger$ for the same reasons as in rule (LF-IN). As stated in [66, 69] the typing rule for the output process is what differs the type system for deadlock-freedom from the type system for lock-freedom. The decrement operation on the obligation level avoids infinite sendings and hence livelocks, as shown in Example 12.3.2. Rule (LF-RES) is the key rule for establishing lock-freedom; it states that the restriction of a name x in a process P is well-typed if x is used reliably

in P. The notion of reliability is checked by the predicate $rel(U)$ which we previously introduced. Rules (LF-RecVar), (LF-RecProc) and (LF-EqVal) are the same as the ones presented in Section 10.3.

The type system we described guarantees the lock-freedom property. We state the following theorem which can be found in [65].

Theorem 12.4.4 (Lock-Freedom)**.** *If* $\Gamma \vdash_{\mathtt{LF}} P$ *and* $rel(\Gamma)$*, then* P *is lock-free.*

Corollary 12.4.5 (Lock-Freedom for Closed Processes)**.** *If* $\emptyset \vdash_{\mathtt{LF}} P$*, then* P *is lock-free.*

CHAPTER 13

Background on Session Types for Progress

In this chapter we recall the π-calculus with sessions given in Chapter 5, by performing few modifications to the syntax of types and terms in order to deal with progress property.

13.1 Syntax

The syntax of processes and values of the π-calculus with sessions is presented in the following Fig. 13.1. Processes include the output $x!\langle v\rangle.P$ and the input $x?(y).P$ processes, conditional **if** v **then** P **else** Q, parallel composition $P \mid Q$ and inaction **0** are standard. Process $(\nu xy)P$ is the restriction of co-variables and X and **rec**$X.P$ model recursion. Branching is the standard one $x \triangleright \{l_i : P_i\}_{i\in I}$ as in Section 5.1. As long as selection is concerned, we adopt a more general notion of selection $x \triangleleft \{l_i : P_i\}_{i\in I}$ which substitutes the standard selection $x \triangleleft l_j.P$. The reason for this modification is to accommodate the notion of progress for sessions, and will be clearer in the next sections.

13.2 Semantics

The reduction rules are presented in Fig. 13.2 and are the same as the ones given in Section 5.2. We report only the most important ones: rules (R-Com), (R-Sel) and

$$
\begin{array}{rll}
P, Q ::= & x!\langle v\rangle.P & \text{output} \\
 & x?(y).P & \text{input} \\
 & x \triangleleft \{l_i : P_i\}_{i\in I} & \text{selection} \\
 & x \triangleright \{l_i : P_i\}_{i\in I} & \text{branching} \\
 & \textbf{if } v \textbf{ then } P \textbf{ else } Q & \text{conditional} \\
 & P \mid Q & \text{composition} \\
 & \mathbf{0} & \text{inaction} \\
 & (\nu xy)P & \text{session restriction} \\
 & X & \text{process variable} \\
 & \textbf{rec}X.P & \text{recursive process} \\
v ::= & x & \text{variable} \\
 & \texttt{true} \mid \texttt{false} & \text{boolean values}
\end{array}
$$

Figure 13.1: Syntax of π-calculus with sessions: enhanced

$$
\begin{array}{rc}
\text{(R-Com)} & (\nu xy)(x!\langle v\rangle.P \mid y?(z).Q \mid R) \to (\nu xy)(P \mid Q[v/z] \mid R) \\
\text{(R-Sel)} & (\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I} \mid R) \to (\nu xy)(P \mid P_j \mid R) \quad j \in I \\
\text{(R-SelNorm)} & x \triangleleft \{l_i : P_i\}_{i\in I} \to x \triangleleft l_j.P_j \quad j \in I \\
\text{(R-Rec)} & \dfrac{P[\textbf{rec}X.P/X] \to P'}{\textbf{rec}X.P \to P'}
\end{array}
$$

Figure 13.2: Semantics of session types: enhanced

(R-Rec), were explained in details in the previous chapters. We add a *selection normalisation*, rule (R-SelNorm) stating that the new selection process $x \triangleleft \{l_i : P_i\}_{i\in I}$ reduces to the old selection process $x \triangleleft l_j.P$ being j one of the labels in I. We omit the context closure rules for parallel composition, restriction and structural congruence, and the reader can refer to Section 5.2 for a detailed presentation.

13.3 Types

The syntax of types of the π-calculus with sessions is presented in Fig. 13.3 and is an extension of the syntax of types presented in Section 5.3, since it includes $\mathbf{t}$ and $\mu\mathbf{t}.T$ to model recursive types.

q	::=	$\mathsf{lin} \mid \mathsf{un}$	qualifiers
p	::=	$!T.T$	send
		$?T.T$	receive
		$\oplus\{l_i : T_i\}_{i \in I}$	select
		$\&\{l_i : T_i\}_{i \in I}$	branch
T	::=	$q\ p$	qualified pretype
		$\mathtt{end}$	termination
		$\mathtt{Bool}$	boolean type
		$\mathbf{t}$	type variable
		$\mu\mathbf{t}.T$	recursive type

Figure 13.3: Syntax of session types: enhanced

13.4 Typing Rules

The typing judgements, since recursion is adopted, have the following form:

$$\Theta; \Gamma \vdash P$$

where

$$\Gamma ::= \emptyset \mid \Gamma, x : T$$

$$\Theta ::= \emptyset \mid \Theta, X : \Gamma$$

The typing rules are presented in the following Fig. 13.4. The only difference is in rule (T-Sel), which well-types the new selection process. This typing rule is very similar to the typing rule for branching, since the selection process chooses over a set of labels and not only one label.

$$\frac{\mathsf{un}(\Gamma)}{\Theta; \Gamma, x : T \vdash x : T}\ (\text{T-Var}) \qquad \frac{\mathsf{un}(\Gamma) \quad v = \texttt{true} \,/\, \texttt{false}}{\Theta; \Gamma \vdash v : \texttt{Bool}}\ (\text{T-Val})$$

$$\frac{\mathsf{un}(\Gamma)}{\Theta; \Gamma \vdash \mathbf{0}}\ (\text{T-Inact}) \qquad \frac{\Theta; \Gamma_1 \vdash P \quad \Theta; \Gamma_2 \vdash Q}{\Theta; \Gamma_1 \circ \Gamma_2 \vdash P \mid Q}\ (\text{T-Par})$$

$$\frac{\Theta; \Gamma, x : T, y : \overline{T} \vdash P}{\Theta; \Gamma \vdash (\boldsymbol{\nu} xy)P}\ (\text{T-Res}) \qquad \frac{\Theta; \Gamma_1 \vdash v : \texttt{Bool} \quad \Theta; \Gamma_2 \vdash P \quad \Theta; \Gamma_2 \vdash Q}{\Theta; \Gamma_1 \circ \Gamma_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q}\ (\text{T-If})$$

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

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

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

$$\frac{\Theta; \Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I} \quad \Theta; \Gamma_2 + x : T_i \vdash P_i \quad \forall i \in I}{\Theta; \Gamma_1 \circ \Gamma_2 \vdash x \triangleleft \{l_i : P_i\}_{i \in I}}\ (\text{T-Sel})$$

$$\frac{\Theta(X) = \Gamma}{\Theta; \Gamma \vdash X}\ (\text{T-RecVar}) \qquad \frac{\Theta, X : \Gamma; \Gamma \vdash P}{\Theta; \Gamma \vdash \textbf{rec} X.P}\ (\text{T-RecProc})$$

$$\frac{\Theta, \Gamma \vdash v : T \quad T \sim_{\mathsf{type}} S}{\Theta, \Gamma \vdash v : S}\ (\text{T-EqVal})$$

Figure 13.4: Typing rules for the π-calculus with sessions: enhanced

CHAPTER 14

Progress as Compositional Lock-Freedom

In this chapter we present our main results related to progress and lock-freedom in the π-calculus with sessions. We start by giving the definition of lock-freedom for sessions, which is an adaptation of the corresponding definition in the standard π-calculus and we give a a relation between lock-freedom and the notion of progress already defined for sessions [20, 27].

14.1 Lock-Freedom for Sessions

In order to formally define the lock-freedom property, we give in the following a few definitions and auxiliary lemmas [65], which are originally stated for the standard π- calculus an are adapted here for the π- calculus with session, taking care in particular about the choice processes.

Definition 14.1.1 (Reduction Sequence)**.** *A set of processes $\{P_i\}_{i\in I}$ for $I \subseteq Nat$ is called a* reduction sequence *and is denoted as $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots$, if $P_{i-1} \rightarrow P_i \; \forall i \in I \setminus \{0\}$.*
A reduction sequence is normal *if i) $\forall i \in I \; P_i$ is in normal form and ii) the sequence of the restricted channels of P_{i-1} is a prefix of the sequence of the restricted channels of P_i.*
A reduction sequence is complete *if either $I = Nat$ or $I = [n]$ and $P_n \not\rightarrow$.*

Definition 14.1.2 (Fair Reduction Sequence)**.** *A normal, complete reduction sequence* $P_0 \to P_1 \to P_2 \to \dots$ *is* fair *if the following conditions hold.*

1. *If there exists an infinite increasing sequence* $n_0 < n_1 < \cdots$ *of natural numbers such that* $P_{n_j} \equiv (\boldsymbol{\nu} \widetilde{x_j y_j})(x!\langle v\rangle.Q \mid y?(z).Q_j \mid R_j)$, $\forall n_j$, *then there exists* $n \geq n_0$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x!\langle v\rangle.Q \mid y?(z).Q' \mid R')$ *and* $(\boldsymbol{\nu}\widetilde{xy})(Q \mid Q'[v/z] \mid R') \equiv P_{n+1}$.

2. *If there exists an infinite increasing sequence* $n_0 < n_1 < \cdots$ *of natural numbers such that* $P_{n_j} \equiv (\boldsymbol{\nu} \widetilde{x_j y_j})(x?(z).Q \mid y!\langle v\rangle.Q_j \mid R_j)$, $\forall n_j$, *then there exists* $n \geq n_0$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x?(z).Q \mid y!\langle v\rangle.Q' \mid R')$ *and* $(\boldsymbol{\nu}\widetilde{xy})(Q[v/z] \mid Q' \mid R') \equiv P_{n+1}$.

3. *If there exists an infinite increasing sequence* $n_0 < n_1 < \cdots$ *of natural numbers such that* $P_{n_j} \equiv (\boldsymbol{\nu} \widetilde{x_j y_j})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft \{l_i : Q_i\}_{i\in I} \mid R_j)$, $\forall n_j$, *then there exists* $n \geq n_0$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft \{l_i : Q'_i\}_{i\in I} \mid R')$ *and* $(\boldsymbol{\nu}\widetilde{xy})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft l_k.Q'_k \mid R') \equiv P_{n+1}$ *for some* $k \in I$ *and* $(\boldsymbol{\nu}\widetilde{xy})(P_k \mid Q'_k \mid R') \equiv P_{n+2}$.

4. *If there exists an infinite increasing sequence* $n_0 < n_1 < \cdots$ *of natural numbers such that* $P_{n_j} \equiv (\boldsymbol{\nu} \widetilde{x_j y_j})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R_j)$, $\forall n_j$, *then there exists* $n \geq n_0$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid y \triangleright \{l_i : Q'_i\}_{i\in I} \mid R')$ *and* $(\boldsymbol{\nu}\widetilde{xy})(x \triangleleft l_k.P_k \mid y \triangleright \{l_i : Q'_i\}_{i\in I} \mid R') \equiv P_{n+1}$ *for some* $k \in I$ *and* $(\boldsymbol{\nu}\widetilde{xy})(P_k \mid Q'_k \mid R') \equiv P_{n+2}$.

Now we are ready to give the definition of lock-freedom. Intuitively, a process is lock-free if for any fair reduction sequence a process which is trying to perform a communication will eventually succeed.

Remark 14.1.3. *Note that in the original work [65], the lock-freedom property states that a process annotated with a mark* c*, eventually succeeds; for the non marked processes it is not required such a constraint. In our framework, we drop the mark and proceed as if all processes were marked, since we want all processes to satisfy the lock-freedom property, and hence eventually communicate.*

In order to define lock-freedom, we assume, as in the original work, a *strongly fair scheduling* [28, 41], which intuitively means that every process enabled to participate in a communication infinitely many times, will eventually do so.

Definition 14.1.4 (Lock-Freedom for Sessions)**.** *A process* P_0 *is* lock-free *under fair scheduling, if for any fair reduction sequence* $P_0 \to P_1 \to P_2 \to \cdots$ *we have the following*

1. *if* $P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(x!\langle v\rangle.Q \mid R)$ *(for* $i \geq 0$*), implies that there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x!\langle v\rangle.Q \mid y?(z).R_1 \mid R_2)$ *and* $P_{n+1} \equiv (\boldsymbol{\nu}\widetilde{xy})(Q \mid R_1[v/z] \mid R_2)$*;*

2. *if* $P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(x?(z).Q \mid R)$ *for some* $i \geq 0$*, there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x?(z).Q \mid y!\langle v\rangle.R_1 \mid R_2)$ *and* $P_{n+1} \equiv (\boldsymbol{\nu}\widetilde{xy})(Q[v/z] \mid R_1 \mid R_2)$*;*

3. *if* $P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid R)$ *for some* $i \geq 0$*, there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid S) \rightarrow (\boldsymbol{\nu}\widetilde{xy})(x \triangleleft l_j.P_j \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid S) \equiv P_{n+1}$ *and* $P_{n+2} \equiv (\boldsymbol{\nu}\widetilde{xy})(P_j \mid Q_j \mid S)$ *for* $j \in I$*;*

4. *if* $P_i \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleright \{l_i : P_i\}_{i\in I} \mid R)$ *for some* $i \geq 0$*, there exists* $n \geq i$ *such that* $P_n \equiv (\boldsymbol{\nu}\widetilde{xy})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft \{l_i : Q_i\}_{i\in I} \mid S) \rightarrow (\boldsymbol{\nu}\widetilde{xy})(x \triangleleft l_j.P_j \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid S) \equiv P_{n+1}$ *and* $P_{n+2} \equiv (\boldsymbol{\nu}\widetilde{xy})(P_j \mid Q_j \mid S)$ *for* $j \in I$*.*

14.2 Progress for Sessions

Progress property is fundamental in a concurrent or distributed scenario. It has been studied in session-based systems by adopting cumbersome definitions and type systems for guaranteeing it. Progress is a property checked for closed as well as open processes. Intuitively, it states that each session, once started, is guaranteed to satisfy all the requested interactions. In particular this means that progress property is a stronger property than deadlock-freedom.

Before giving the formal definition of progress, we first need to introduce some auxiliary definitions. We start with the definition of characteristic process. A *characteristic process* is the simplest process that can inhabit a type and is formally given by Definition 14.2.1. Notice that this is an extension and modification of the original definition given in [20], where there are no recursive types.

Definition 14.2.1 (Characteristic Process)**.** *Given a type T, its* characteristic pro-

cess $[\![T]\!]_f^x$ *is inductively defined on the structure of* T *as:*

$$
\begin{array}{rl}
(\text{inVal}) & [\![q?\texttt{Bool}.U]\!]_f^x = x?(y).[\![U]\!]_f^x \\
(\text{outVal}) & [\![q!\texttt{Bool}.U]\!]_f^x = x!\langle\texttt{true}\rangle.[\![U]\!]_f^x \\
(\text{inSess}) & [\![q'?(qp).U]\!]_f^x = x?(y).([\![U]\!]_f^x \mid [\![qp]\!]_f^y) \\
(\text{outSess}) & [\![q'!(qp).U]\!]_f^x = (\boldsymbol{\nu} zw)(x!\langle z\rangle.([\![U]\!]_f^x \mid [\![\overline{qp}]\!]_f^w)) \\
(\text{inSum}) & [\![q\&\{l_i : (q_i p_i)_i\}_{i\in I}]\!]_f^x = x \triangleright \{l_i : [\![q_i p_i]\!]_f^x\}_{i\in I} \\
(\text{outSum}) & [\![q \oplus \{l_i : (q_i p_i)_i\}_{i\in I}]\!]_f^x = x \triangleleft \{l_i : [\![q_i p_i]\!]_f^x\}_{i\in I} \\
(\text{end}) & [\![\texttt{end}]\!]_f^x = \mathbf{0} \\
(\text{recVar}) & [\![\mathbf{t}]\!]_f^x = f(\mathbf{t}) \\
(\text{rec}) & [\![\mu\mathbf{t}.T]\!]_f^x = \mathbf{rec}X.[\![T]\!]_{f,\{\mathbf{t}\mapsto X\}}^x
\end{array}
$$

Recall from Section 6.3.4 that an *evaluation context*, or simply a context, is a process with a hole $[\cdot]$ and is produced by the following grammar:

$$\mathcal{E}[\cdot] ::= \quad [\cdot] \mid P \quad | \quad (\boldsymbol{\nu} xy)\mathcal{E}[\cdot] \quad | \quad \mathcal{E}[\cdot] \mid \mathcal{E}[\cdot] \quad | \quad \mathbf{rec}X.\mathcal{E}[\cdot]$$

By using the above definition we now define catalysers, inspired by [27]. A *catalyser* is a context with only characteristic processes:

Definition 14.2.2 (Catalyser)**.** *A* catalyser $C[\cdot]$ *is a context produced by the following grammar:*

$$C[\cdot] ::= \ [\cdot] \quad | \quad (\boldsymbol{\nu} xy)\, C[\cdot] \quad | \quad C[\cdot] \mid [\![T]\!]_f^x$$

We illustrate the catalysers by the following example:

Example 14.2.3. The following context $C[\cdot]$ is a catalyser obtained by composing the characteristic processes P_1 and P_2 respectively of the channel types $T_1 = ?(!\texttt{Bool}.\texttt{end}).\texttt{end}$ and $T_2 = \oplus\{l_1 : \texttt{end}, l_2 : !\texttt{Bool}.\texttt{end}\}$:

$$
\begin{array}{rcl}
C[\cdot] & = & (\boldsymbol{\nu} wx)(\boldsymbol{\nu} uy)([\cdot] \mid P_1 \mid P_2) \\
P_1 & = & x?(z).(z!\langle\texttt{true}\rangle.\mathbf{0} \mid \mathbf{0}) \\
P_2 & = & y \triangleleft l_2.y!\langle\texttt{true}\rangle.\mathbf{0}
\end{array}
$$

To conclude, we define the $\bowtie$ operator, a binary relation over processes, which relates processes that start with respective co-actions. This operator, differently

from the original one in [10,27], is parametrised in a pair of variables $\{x, y\}$, which are co-variables.

Definition 14.2.4 ($\bowtie$)**.** *The duality $\bowtie_{\{x,y\}}$ between input and output processes is defined as follows:*

$$x!\langle v\rangle.P \bowtie_{\{x,y\}} y?(z).Q$$
$$x \triangleleft \{l_i : P_i\}_{i\in I} \bowtie_{\{x,y\}} y \triangleright \{l_i : Q_i\}_{i\in I}$$

We are now ready to give the formal definition of progress. This definition is inspired by [10, 27], which is an improvement of the definition of progress used in [20].

Definition 14.2.5 (Progress)**.** *A process P has* progress *if for all $C[\cdot]$ such that $C[P]$ is well-typed, $C[P] \to^* \mathcal{E}[R]$ (where R is an input or an output) implies that there exist $C'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $C'[\mathcal{E}[R]] \to^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that (νxy) is a restriction in $C'[\mathcal{E}[R]]$.*

14.3 Lock-Freedom meets Progress

In this section we put together lock-freedom and progress for session π-calculus in order to understand their relation. We split this section in the following two subsections, by analysing separately the closed processes and then the open ones.

14.3.1 Properties of Closed Terms

By analysing the definitions of lock-freedom and progress, we notice that there is some similarity. In particular, for closed terms, i.e., processes with no free variables, the properties of lock-freedom and progress are tightly related. We formalise this relation in the following.

Theorem 14.3.1 (Lock-freedom $\Rightarrow$ Progress)**.** *Let P be a well-typed closed process. Then, P lock-free implies P has progress.*

Proof. The proof proceeds by contradiction. Let us assume that P does not have progress. Formally, it means that: there exists $C[\cdot]$ such that $C[P]$ is well-typed, $C[P] \to^* \mathcal{E}[R]$ where R is an input or an output, and for all $C'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' it holds that $C'[\mathcal{E}[R]] \not\to^* \mathcal{E}'[R][R']$ such that $R \bowtie_{\{x,y\}} R'$ where x and y are such that (νxy) is a restriction in $C'[\mathcal{E}[R]]$. Instead, to reach a contradiction, we show

that there exists $C'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $C'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ such that $R \bowtie_{\{x,y\}} R'$. Since P is closed it means that it does not communicate with any context $C[\cdot]$ it is inserted in. Hence, $C[P] \rightarrow^* \mathcal{E}[R]$ means that reductions have occurred either in the catalyser C or in P, separately. Let now $C'[\cdot] = [\cdot]$. We show that $\mathcal{E}[R] \rightarrow^* \mathcal{E}'[R][R']$. Since P is lock-free, by definition $P \rightarrow^* P_i$ and for some $n \geq i$, $P_i \rightarrow^* P_n$ and P_n has both action and co-action on some channels $(\boldsymbol{\nu} x'y')$. Notice that P_i is a subprocess of $\mathcal{E}[R]$ and hence P_n is a subprocess of $\mathcal{E}'[R][R']$ where R and R' are the action and co-action that have come up at the top level in the reduction under the restriction $(\boldsymbol{\nu} x'y')$ and let $x = x', y = y'$. □

What we find interesting in the case of closed processes, is that the opposite of the previous theorem is also true. We show it in the following.

Theorem 14.3.2 (Progress ⇒ Lock-freedom)**.** *Let P be a well-typed closed process. Then, P has progress implies P lock-free.*

Proof. From the definition of progress we know that for all catalysers $C[\cdot]$, $C[P] \rightarrow^* \mathcal{E}[R]$. In particular, this holds also for the empty catalyser $[\cdot]$. Hence, $P \rightarrow^* \mathcal{E}[R] \equiv P_i$. Here we can assume, without any loss of generality (the other cases are trivial) that R is an input or an output process. Furthermore, we know that there exist $C'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $C'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that $(\boldsymbol{\nu} xy)$ is a restriction in $C'[\mathcal{E}[R]]$. Since P is closed, P_i is also closed and this means that it does not communicate with any catalyser it is inserted in. Hence, $C'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ means that reductions have occurred either in the catalyser C' or in $\mathcal{E}[R]$, separately. Notice that R is part of $\mathcal{E}[R] \equiv P_i$, and since R occurs in the redex $\mathcal{E}'[R][R']$ together with its counterpart R', it means that $P_i \rightarrow^* P_n$ where P_n is a subprocess of $\mathcal{E}'[R][R']$, and the communication occurs over $(\boldsymbol{\nu} xy)$. We conclude by applying the definition of lock-freedom. □

It follows as a corollary from Theorems 14.3.1 and 14.3.2 that the lock-freedom and progress properties coincide for closed terms.

Corollary 14.3.3 (Progress ⇔ Lock-freedom)**.** *Let P be a well-typed closed process. Then P is lock-free if and only if P has progress.*

14.3.2 Properties of open terms

We switch now to a more general setting, i.e., processes that can be open. Differently from the case of the closed terms, the definitions of lock-freedom and

progress do not coincide in the case of open terms. For example, consider the following process:

$$P \;=\; x!\langle \texttt{true} \rangle.x?(z).\mathbf{0}$$

In process P, x is an open session with a missing participant. Process P has progress, by following Definition 14.2.5 but it is not lock-free because it does not respect Definition 14.1.4, since it is stuck and does not reduce.

In this section we try to reply to the question we posed in the introduction, namely trying to understand the relationship between the notions of lock-freedom and progress for open-ended systems. Although the two properties do not coincide in the case of open terms, we can still relate progress to lock-freedom.

The idea is to use catalysers in order to reduce the problem of checking progress for open terms to the problem of checking progress (and lock-freedom) for closed terms. The intuition for using catalysers is that when a process is open, its type can provide us some information about how such a process can be put in a context such that the final composition is closed. We formalise this idea with the notion of closure given below.

Definition 14.3.4 (Closure)**.** *Let P be such that $\Gamma \vdash P$. Then, the closure of P, denoted by* $\mathsf{close}(P)$*, is the process $C[P]$ where*

$$C[\cdot] = (\nu \widetilde{xy})(\,[\cdot] \mid \prod_{x_i : T_i \in \Gamma} [\![T_i]\!]_f^{y_i})$$

Notice that, in the definition above all x_i in the sequence $\widetilde{xy}$ correspond exactly to the domain of Γ. The y_i in $\widetilde{xy}$ are all different from x_i and are the variables used to create the characteristic processes from every type T_i. Below, we give an example of how the closure of a process works.

Example 14.3.5. Consider the open process previously shown

$$P = x!\langle \texttt{true} \rangle.x?(z).\mathbf{0}$$

We can type P in a typing context $\Gamma = x : !\texttt{Bool}.?\texttt{Bool}.\texttt{end}$. Then, the closure of P is defined as:

$$\mathsf{close}(P) = (\nu xy)([P] \mid y?(z).y!\langle \texttt{true} \rangle.\mathbf{0})$$

The closure procedure $\mathsf{close}(P)$ can also be applied to processes that are al-

ready closed, as shown in the following.

Example 14.3.6. Consider the closed process:

$$P \;=\; (\boldsymbol{\nu} xy)(x!\langle \mathtt{true}\rangle.\mathbf{0} \mid y?(z).\mathbf{0})$$

Since P can only be typed with the empty typing context, i.e., $\emptyset \vdash P$, in this case we have that $\mathsf{close}(P) = P$. This means that the catalyser that we can place P into, in order to close it, is the empty catalyser $[\cdot]$.

As a first property of closure, we can immediately observe that the closure operation preserves typability.

Proposition 14.3.7 (Closure preserves typability)**.** *If* $\Gamma \vdash P$ *then* $\emptyset \vdash \mathsf{close}(P)$.

Proof. It follows by the definition of characteristic process and repeated use of the typing rules (T-Par) and (T-Res).

□

We present in the following, one of the major properties of our technical development, which will be crucial in establishing our main results. The closure procedure defines a new way for checking progress: a process P has progress if its closure can always reduce to terms where an action at the top level can be matched with its co-action in a parallel subterm. We formalise this notion below.

Lemma 14.3.8 (From Closure to Progress)**.** *Let* $\Gamma \vdash P$*. Then, P has progress if and only if* $\mathsf{close}(P) \to^* \mathcal{E}[R]$ *(where R is an input or an output process) implies that there exist* $\mathcal{E}'[\cdot][\cdot]$ *and* R' *such that* $\mathcal{E}[R] \to^* \mathcal{E}'[R][R']$ *and* $R \bowtie_{\{x,y\}} R'$ *for some x and y such that* $(\boldsymbol{\nu} xy)$ *is a restriction in* $\mathcal{E}[R]$.

Proof. We split the proof into two cases.

only if. Follows immediately by the definitions of progress and $\mathsf{close}(P)$.

if. Let $C[\cdot]$ be a catalyser such that $C[P]$ is well-typed. Intuitively, any catalysers can be written by splitting the processes put in parallel with P in two: the ones that implement and the ones that do not implement the counterparts of sessions in P; formally:

$$C[\cdot] \equiv (\boldsymbol{\nu} \widetilde{xy})([\cdot] \mid Q_1 \mid Q_2)$$
$$Q_1 = \textstyle\prod_{x_j : T_j \notin \Gamma} [\![T_j]\!]_f^{y_j}$$
$$Q_2 = \textstyle\prod_{x_i : T_i \in \Gamma'} [\![T_i]\!]_f^{y_i} \qquad \text{where } \Gamma' \subseteq \Gamma$$

Moreover, from the definition of $\mathsf{close}(P)$ we know that:

$$\mathsf{close}(P) = (\nu \widetilde{xy}')(P \mid Q_2 \mid Q_3)$$
$$Q_3 = \prod_{x_i : T_i \in \Gamma \setminus \Gamma'} [\![T_i]\!]_f^{y_i}$$

Since $C[P]$ is well-typed, from the typing rules we know that Q_1 cannot interact neither with P nor with Q_2; therefore we have only three possible cases for the derivation of $C[P] \rightarrow^* \mathcal{E}''[R]$: (i) $P \rightarrow^* P'$; (ii) $(\nu \widetilde{xy})Q_1 \rightarrow^* (\nu \widetilde{xy})Q_1'$; (iii) $(\nu \widetilde{xy})(P \mid Q_2) \rightarrow^* (\nu \widetilde{xy})(P' \mid Q_2')$. We discuss the cases.

(i) For this case, we know that $\mathsf{close}(P) \rightarrow^* \mathsf{close}(P') \equiv \mathcal{E}[R]$ and $C[P] \rightarrow^* C[P']$. We now choose close as catalyser for $C[P']$; therefore: $\mathsf{close}(C[P']) \equiv C[\mathsf{close}(P')] \equiv (\nu \widetilde{xy}'')(\nu \widetilde{xy})(P' \mid Q_1 \mid Q_2 \mid Q_3)$ where $\widetilde{xy}''$ are the free names in the typing of $C[P']$. Since, by hypothesis, $\mathsf{close}(P') \rightarrow^* \mathcal{E}'[R][R']$ we also know that: $\mathsf{close}(C[P']) \rightarrow^* C[\mathcal{E}'[R][R']]$ and the thesis follows.

(ii) $(\nu \widetilde{xy})Q_1 \rightarrow^* (\nu \widetilde{xy})Q_1'$. This means that $C[P] \rightarrow^* C'[P]$, since only the catalyser reduces. We now choose close as catalyser for $C'[P]$; therefore: $\mathsf{close}(C'[P]) \equiv C'[\mathsf{close}(P)] \rightarrow^* C'[\mathcal{E}[R]]$ and the thesis follows by applying the hypothesis for $\mathsf{close}(P)$.

(iii) $(\nu \widetilde{xy})(P \mid Q_2) \rightarrow^* (\nu \widetilde{xy})(P' \mid Q_2')$. This means that $C[P] \rightarrow^* C'[P'] \equiv (\nu \widetilde{xy})(P' \mid Q_1 \mid Q_2')$, since both the catalyser and P reduce. By hypothesis, $\mathsf{close}(P) \rightarrow^* \mathcal{E}[R]$ and since the closure gives to P its missing counterpart, it means that P and Q_2 communicate, hence $\mathcal{E}[R] \equiv (\nu \widetilde{xy}')(P' \mid Q_2' \mid Q_3)$ We know that $\mathcal{E}[R] \rightarrow^* \mathcal{E}'[R][R']$ and $R \bowtie_{\{x,y\}} R'$ for some x and y such that (νxy) is a restriction in $\mathcal{E}[R]$. Let $C''[\cdot]$ be the catalyser for $C'[P']$ defined as: $C''[\cdot] \equiv (\nu \widetilde{xy}')([\cdot] \mid Q_3)$. Then $C''[C'[P']] \equiv (\nu \widetilde{xy}')(\nu \widetilde{xy})(P' \mid Q_1 \mid Q_2' \mid Q_3)$and the thesis follows by applying the hypothesis for $\mathsf{close}(P)$.

□

By applying Lemma 14.3.8, we establish that checking the progress property for a process P is equivalent to checking the progress property for its closure:

Theorem 14.3.9 (Closure Progress ⇔ Progress)**.** *If P is well-typed then* $\mathsf{close}(P)$ *has progress if and only if P has progress.*

Proof. We split the proof into two cases.

- *if*. Since P has progress, then for all catalysers we must prove that for every reachable process we can find another catalyser such that every input/output action will eventually be consumed. But then this also holds for $\mathsf{close}(P)$ by definition of close and Lemma 14.3.8.
- *only if*. Follows immediately by Lemma 14.3.8.

□

We are finally able to link the progress and the lock-freedom properties. Our main result is that the progress property of a process P and the lock-freedom property of the closure of P coincide:

Theorem 14.3.10. (Progress ⇔ Closed Lock-Free) *If P is well-typed then P has progress if and only if* $\mathsf{close}(P)$ *is lock-free.*

Proof. It follows immediately from Theorem 14.3.9 and Corollary 14.3.3. □

We summarise the main results as follows. We have proved that, for closed terms, i.e., terms with no free variables, lock-freedom and progress coincide. For open terms, i.e., terms containing free variables, we have shown that these notions do not coincide. However, we define a procedure for *closing* a process by using the notions of catalyser and characteristic process. Then, we prove that progress and lock-freedom coincide for $\mathsf{close}(P)$, which implies progress for P.

14.4 A Type System for Progress

In this section we show theoretical results that permit us to adopt the type system for lock-freedom to check the progress property for the session π-processes. Before doing so, we recall the encoding presented in Section 10.4. In order to encode qualifiers lin and un, previously encoded in m_α, we perform the following translation from m_α to mU.

$$[\![\ell_{\mathsf{i}}]\!] = \ell \mathsf{i}_c^o.\emptyset \qquad [\![\ell_{\mathsf{o}}]\!] = \ell \mathsf{o}_c^o.\emptyset$$
$$[\![\mathsf{i}]\!] = \mathsf{i}_c^o.\emptyset \qquad [\![\mathsf{o}]\!] = \mathsf{o}_c^o.\emptyset$$

The following auxiliary lemma relates the notion of lock-freedom in π-calculus with and without sessions by using the encoding presented so far.

Lemma 14.4.1. *A session process P is lock-free if and only if $[\![P]\!]_f$ is lock-free.*

Proof. The result follows by applying the Definition 14.1.4 for lock-freedom in sessions and Definition 12.4.3 for lock-freedom in the standard π-calculus and by the operational correspondence of the encoding. □

By the results obtained previously, which relate lock-freedom and progress, we show how to use the type system for lock-freedom in the standard π-calculus to derive progress in the π-calculus with sessions.

The following theorem gives the main result of this part of the dissertation.

Theorem 14.4.2 (Progress in Sessions)**.** *Let P be a session process and $\Gamma \vdash P$. If $\emptyset \vdash_{\mathrm{LF}} [\![\mathsf{close}(P)]\!]_f$, then process P has progress.*

Proof. Let $\Gamma \vdash P$ and $\emptyset \vdash_{\mathrm{LF}} [\![\mathsf{close}(P)]\!]_f$. Then, by Theorem 12.4.4 and Corollary 12.4.5 this means that $[\![\mathsf{close}(P)]\!]_f$ is lock-free. By Lemma 14.4.1 also $\mathsf{close}(P)$ is lock-free. By Theorem 14.3.10 we have that P has progress. □

We use the previous result to obtain the following (pseudo-) algorithm for checking the progress property for a session process. Kobayashi's type system comes with a reference implementation, the tool TyPiCal [106].

1: **procedure** PROGRESS(Γ, P)
2: Check $\Gamma \vdash P$
3: Build $\mathsf{close}(P)$ from Γ
4: Encode $[\![\mathsf{close}(P)]\!]_f = P'$
5: **return** TyPiCal(P')
6: **end procedure**

Conclusions, Related and Future Work for Part IV

In Part IV of the dissertation we adopted the notion of lock-freedom to the π-calculus with sessions, and studied the relationship between the notions of progress and lock-freedom, by showing that they are strongly linked, since progress can be thought of as a generalisation of lock-freedom to open processes. We proved that, for closed terms, i.e., terms with no free variables, lock-freedom and progress coincide. For open terms, i.e., terms containing free variables, we showed that these notions do not coincide. However, we defined a procedure to *close* a process by using the notions of catalyser and characteristic process. Then, we proved that progress and lock-freedom coincide for $\mathsf{close}(P)$. Guided by this discovery, we used an existing static analysis for lock-freedom, i.e., Kobayashi's type system from [65], for analysing the progress property. We show in the following that, reusing Kobayashi's technique captures new interesting cases of processes that have progress that could not be successfully recognised by previous type systems studied for the π-calculus with sessions.

Comparison with Related Work In the following we recall some examples taken from [10, 20, 27, 37, 94], show how the encoding works and compare them with our framework. For the sake of readability, we simplify the encoding by omitting the creation of fresh channels when the latter are not used in the continuation of a process.

Example 14.4.3. Consider the session process

$$(\nu ab)(\nu cd)(\; a?(z).d!\langle z\rangle \;\mid\; c?(w).b!\langle w\rangle \;)$$

which is deadlocked, and therefore does not have progress. This process is not typable in the type systems for progress in [10,27,94]. By the encoding we obtain the process:

$$(\boldsymbol{\nu} x)(\boldsymbol{\nu} y)(\ x?(z).y!\langle z\rangle \ \mid\ y?(w).x!\langle w\rangle\ )$$

where the function f in which the encoding is parametrised maps $a, b \mapsto x$ and $c, d \mapsto y$. As expected, our technique would (correctly) discard the example above since it is untypable in Kobayashi's type system. In particular, this process would fail to be typed since the *rel* predicate does not hold.

Example 14.4.4. The process

$$(\boldsymbol{\nu} ab)\Big(\ b!\langle 1\rangle \ \mid\ (\boldsymbol{\nu} cd)(\ d!\langle 1\rangle \ \mid\ c?(y).a?(z)\ )\Big)$$

is lock-free and, therefore, has also progress. However, it is rejected by [94] since the type system presented therein does not distinguish between obligation and capability tags, but uses a single tag instead. By the encoding in the π-calculus, we obtain the process

$$(\boldsymbol{\nu} k)\Big(\ k!\langle 1\rangle \ \mid\ (\boldsymbol{\nu} l)(\ l!\langle 1\rangle \ \mid\ l?(y).k?(z)\ )\ \Big)$$

This process is typed in Kobayashi's type system by letting $k : \mathsf{o}_{c_1}^{o_1} \mid \mathsf{i}_{c_2}^{o_2}$ and letting $l : \mathsf{o}_{c_3}^{o_3} \mid \mathsf{i}_{c_4}^{o_4}$, and yields the following system of inequations: $o_1 = o_3 = 0,\ o_1 \leq c_2,\ o_2 \leq c_1,\ o_3 \leq c_4,\ o_4 \leq c_3,\ o_2 > c_3$ which has the following solution $o_1 = o_3 = o_4 = c_2 = c_3 = c_4 = 0$ and $o_2 = c_1 = 1$.

Example 14.4.5. Consider the session process

$$(\boldsymbol{\nu} ab)(\boldsymbol{\nu} cd)\left(\begin{array}{l} a?(x).\ c!\langle x\rangle.\ c?(y).\ a!\langle y\rangle \\ \mid \\ b!\langle \mathtt{true}\rangle.\ d?(z).\ d!\langle \mathtt{false}\rangle.\ b?(z) \end{array}\right)$$

This process satisfies the progress property, but it is rejected by the type systems in [10] and [20]. This is because, in the two processes in parallel, there is a circular dependency between channels which such type systems cannot handle. Let us now

consider its encoding in the π-calculus, given by the following process:

$$(\nu k)(\nu l)\left(\begin{array}{l} k?(x, c_1).\ (\nu c_2)\Big(\ l!\langle x, c_2\rangle.\ c_2?(y).\ c_1!\langle y\rangle\ \Big) \\ \quad \mid \\ (\nu c_1)\Big(k!\langle \texttt{true}, c_1\rangle.\ l?(z, c_2).\ c_2!\langle \texttt{false}\rangle.\ c_1?(z)\Big) \end{array}\right)$$

This process is correctly recognised as having progress by our technique, since it is well-typed in Kobayashi's type system. The types assigned to the channels are as follows:

$$k : [\texttt{Bool}, T_1]\ \mathsf{i}_0^0 \mid \mathsf{o}_0^0 \qquad\qquad l : [\texttt{Bool}, T_2]\ \mathsf{o}_1^1 \mid \mathsf{i}_1^1$$

such that

$$T_1 = [\texttt{Bool}]\ \mathsf{o}_3^1 \mid \mathsf{i}_1^3 \qquad\qquad T_2 = [\texttt{Bool}]\ \mathsf{i}_0^2 \mid \mathsf{o}_2^0$$

Future Work. As future work, we plan to extend our approach to multiparty sessions [27, 56]. For the multiparty setting, we need to investigate an extension of the encoding to a setting where sessions are established between more than two peers and messaging is asynchronous, which is future work related to Part II and III. It is not clear whether Kobayashi's usage types are expressive enough for handling such situations. This is the case because, as long as the encoding is concerned, usage types have the same expressive power as linear types.

The works in [16, 113] use linear logic to type processes in the π-calculus with sessions. While these works guarantee lock-freedom, we conjecture that their techniques can be reused for progress, similarly to what we have done with Kobayashi's type system. We leave such an investigation as future work.

Bibliography

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Sci. Comput. Program.*, 77(12):1289–1309, 2012.

[3] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. Analysis of may-happen-in-parallel in concurrent objects. In *FMOODS/FORTE*, pages 35–51, 2012.

[4] OSGi Alliance. *Osgi Service Platform, Release 3*. IOS Press, Inc., 2003.

[5] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[6] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 2nd edition, 1984.

[7] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the 2nd European conference on Software Architecture*, EWSA'05, pages 1–17, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. To be submitted to International Conference, 2014.

[9] Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types. *CoRR*, abs/1310.6176, 2013.

[10] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, pages 418–433, 2008.

[11] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4), 1998.

[12] Paolo Di Blasio and Kathleen Fisher. A calculus for concurrent objects. In *CONCUR*, pages 655–670, 1996.

[13] Frank S. De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07), volume 4421 of LNCS*, pages 316–330. Springer-Verlag, 2007.

[14] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software - Practice and Experience*, 36(11-12), 2006.

[15] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, pages 330–349, 2013.

[16] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.

[17] Luís Caires and Hugo Torres Vieira. Conversation types. In *ESOP'09*, volume 5502 of *LNCS*, pages 285–300, Heidelberg, Germany, 2009. Springer-Verlag.

[18] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating sessions and methods in object-oriented languages with generics. *Theor. Comput. Sci.*, 410(2-3):142–167, 2009.

[19] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. To appear in Proc. of COORDINATION 2014, 2014.

[20] Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *Proc. of ICE'10*, pages 13–27, 2010.

[21] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 2–17, Heidelberg, Germany, 2007. springer.

[22] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.

[23] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Types for the ambient calculus. *Information and Computation*, 177(2):160 – 194, 2002.

[24] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[25] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. *SIGPLAN Not.*, 39(1):123–134, 2004.

[26] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[27] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress for dynamically interleaved multiparty sessions (long version), 2008. `http://www.di.unito.it/~dezani/papers/cdy12.pdf`.

[28] Gerardo Costa and Colin Stirling. Weak and strong fairness in ccs. *Information and Computation*, 73(3):207 – 244, 1987.

[29] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. In *In Proc. IASTED Software Engineering and Applications (SEA'04)*, 2004.

[30] Ornela Dardha, Elena Giachino, and Michael Lienhardt. A type system for components. In *SEFM*, pages 167–181, 2013.

[31] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, pages 139–150, 2012.

[32] Ornela Dardha, Daniele Gorla, and Daniele Varacca. Semantic subtyping for objects and classes. In *FMOODS/FORTE*, pages 66–82, 2013.

[33] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR'11*, pages 280–296, 2011.

[34] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Inf. Comput.*, 204(7):1045–1082, 2006.

[35] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *Proc. of POPL*, pages 435–446. ACM, 2011.

[36] Mariangiola Dezani-Ciancaglini and Ugo de' Liguoro. Sessions and session types: an overview. In *Proc. of the 6th International Workshop on Web Service and Formal Methods (WS-FM'09)*, LNCS, Heidelberg, Germany, 2009. springer.

[37] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC*, volume 4912 of *LNCS*, pages 257–275. springer, 2007.

[38] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *FMCO*, pages 207–245, 2006.

[39] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *Proc. of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *LNCS*, pages 328–352, Heidelberg, Germany, 2006. springer.

[40] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A distributed object-oriented language with session types. In *TGC*, pages 299–318, 2005.

[41] E. Allen Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[42] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP*, pages 364–388, 2004.

[43] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, nov 2005.

[44] Simon J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

[45] Simon J. Gay, Nils Gesbert, and António Ravara. Session types as generic process types. In *PLACES'08*, 2008.

[46] Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *CoRR*, abs/1205.5344, 2012.

[47] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[48] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *IFM*, pages 394–411, 2013.

[49] Elena Giachino and Tudor A. Lascu. Lock Analysis for an Asynchronous Object Calculus. Presented at *ICTCS*. Available at `http://www.cs.unibo.it/~giachino/`, 2012.

[50] Joseph Gil and Itay Maman. Whiteoak: introducing structural typing into java. *SIGPLAN Not.*, 43(10):73–90, 2008.

[51] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. In *CONCUR*, pages 492–507, 2008.

[52] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[53] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[54] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The geneva convention – on the treatment of object aliasing. OOPS Messenger, 1992.

[55] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, Heidelberg, Germany, 1998. springer.

[56] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008.

[57] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[58] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[59] Focus Inria. Overall objectives. `http://raweb.inria.fr/rapportsactivite/RA2012/focus/uid3.html`.

[60] Einar Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs: A core language for abstract behavioral specification. In *FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2012.

[61] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.

[62] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1-2):23–66, 2006.

[63] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.

[64] Naoki Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In *IFIP TCS*, pages 365–389, 2000.

[65] Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.

[66] Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, pages 439–453, 2002.

[67] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.

[68] Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, pages 233–247, 2006.

[69] Naoki Kobayashi. Type systems for concurrent programs. Extended version of [65], Tohoku University, 2007.

[70] Naoki Kobayashi, Benjamin Pierce, and David Turner. Linear types and π-calculus. In *POPL*, volume 21(5), pages 358–371, New York, NY, USA, 1996. ACM Press.

[71] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR*, pages 489–503, 2000.

[72] Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), 2010.

[73] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[74] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in reo using high-level replacement systems. *Sci. Comput. Program.*, 76(1):23–36, 2011.

[75] Michael Lienhardt, Mario Bravetti, and Davide Sangiorgi. An object group-based component model. In *ISoLA (1)*, pages 64–78, 2012.

[76] Michaël Lienhardt, Ivan Lanese, Mario Bravetti, Davide Sangiorgi, Gianluigi Zavattaro, Yannick Welsch, Jan Schäfer, and Arnd Poetzsch-Heffter. A component model for the abs language. In *Proceedings of the 9th international conference on Formal Methods for Components and Objects*, FMCO, pages 165–183, Berlin, Heidelberg, 2011. Springer-Verlag.

[77] Michael Lienhardt, Alan Schmitt, and Jean-Bernard Stefani. Oz/k: a kernel language for component-based open programming. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 43–52, New York, NY, USA, 2007. ACM.

[78] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, pages 47–57, New York, NY, USA, 1988. ACM.

[79] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP*, pages 260–284, 2008.

[80] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? an empirical study. In *ESOP*, pages 95–111, 2009.

[81] Sun Microsystems. JSR 220: Enterprise javabeans, version 3.0 – ejb core contracts and requirements, 2006.

[82] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[83] Robin Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[84] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, may 1999.

[85] Hugo Miranda, Alexandre S. Pinto, and Luis Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems (ICDCS 2001)*. IEEE Computer Society, 2001.

[86] Fabrizio Montesi and Marco Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.

[87] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. 2014.

[88] Fabrizio Montesi and Davide Sangiorgi. A model of evolvable components. In *TGC*, pages 153–171, 2010.

[89] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.

[90] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA*, pages 321–335, 2007.

[91] Wired News. History's worst software bugs, 2005. `http://www.wired.com/software/coolapps/news/2005/11/69355`.

[92] OASIS. Web Services Business Process Execution Language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[93] Klaus Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008.

[94] Luca Padovani. From lock freedom to progress using session types. In Proc. of PLACES, 2013.

[95] Catuscia Palamidessi and D. Valencia, Frank. Recursion vs replication in process calculi: Expressiveness. *Bulletin of the European Association for Theoretical Computer Science*, 87:105–125, 2005.

[96] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In *ESOP'12*, pages 539–558, 2012.

[97] Benjamin C. Pierce. *Types and programming languages*. MIT Press, MA, USA, 2002.

[98] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *LICS'93*, pages 376–385, 1993.

[99] Davide Sangiorgi. An interpretation of typed objects into typed pi-calculus. *Inf. Comput.*, 143(1):34–73, 1998.

[100] Davide Sangiorgi. Termination of processes. *Mathematical. Structures in Comp. Sci.*, 16(1):1–39, 2006.

[101] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[102] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: generalizing active objects to concurrent components. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[103] Clemens Szyperski. *Component Software, 2nd edition*. Addison-Wesley, 2002.

[104] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94*, pages 398–413, 1994.

[105] Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In *FoSSaCS'12*, pages 346–360, 2012.

[106] TYPICAL. Type-based static analyzer for the pi-calculus. `http://www-kb.is.s.u-tokyo.ac.jp/~koba/typical/`.

[107] Antonio Vallecillo, Vasco Thudichum Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundam. Inform.*, 73(4):583–598, 2006.

[108] Vasco Vasconcelos, António Ravara, and Simon J. Gay. Session types for functional multithreading. In *Proc. of the 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 497–511, Heidelberg, Germany, 2004. springer.

[109] Vasco T. Vasconcelos. Fundamentals of session types. *Information Computation*, 217:52–70, 2012.

[110] Vasco Thudichum Vasconcelos. Fundamentals of session types. In *SFM*, pages 158–186, 2009.

[111] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006.

[112] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In *ESOP'08*, volume 4960, pages 269–283, Heidelberg, Germany, 2008. Springer-Verlag.

[113] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.

[114] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for java. In *OOPSLA*, pages 439–453, 2005.

[115] Yannick Welsch and Jan Schäfer. Location types for safe distributed object-oriented programming. In *TOOLS (49)*, pages 194–210, 2011.

[116] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[117] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the π-calculus. *Inf. Comput.*, 191(2):145–202, 2004.

[118] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.