Type Systems for Distributed Programs: Components and Sessions

Ornela Dardha

Foreword

The Italian Chapter of the EATCS (European Association for Theoretical Computer Science) was founded in 1988, and aims at facilitating the exchange of ideas and results among Italian theoretical computer scientists, and at stimulating cooperation between the theoretical and the applied communities in Italy.

One of the major activities of this Chapter is to promote research in theoretical computer science, stimulating scientific excellence by supporting and encouraging the very best and creative young Italian theoretical computer scientists. This is done also by sponsoring a prize for the best PhD thesis. An interdisciplinary committee selects the best PhD thesis, among those defended in the previous year and dealing with one of the many themes in theoretical computer science.

In 2012 we started a cooperation with Atlantis Press so that the selected PhD theses will be published as volumes in the Atlantis Studies in Computing.

The present volume contains the thesis selected for publication in 2015:

Type Systems for Distributed Programs: Components and Sessions by Ornela Dardha (supervisor: Prof. Davide Sangiorgi, Univ. of Bologna, Italy)

The scientific committee that selected this thesis was composed of Professors Margherita Napoli (Univ. of Salerno), Paolo Santi (CNR of Pisa) and Andrea Masini (Univ. of Verona).

They gave the following motivation to justify the assignment of the award to the thesis by Ornela Dardha:

The PhD thesis "Type Systems for Distributed Programs: Components and Sessions" by Ornela Dardha deals with type-based systems for distributed programs. The goal of the thesis is the development of static techniques based on type systems aimed at dealing with consistency and safety properties related with dynamic reconfiguration and communication in complex distributed systems. The main original contributions of the thesis are:

- *the design of a type system for a realistic concurrent object-oriented calculus to statically guarantee the consistency of dynamic reconfigurations;*

- *the study of concrete, non-trivial safety properties of complex distributed systems, namely deadlock freedom, livelock freedom, and progress.*

All the theoretical proposals of the thesis are original and extremely interesting. They represent a major breakthrough in the study of type systems for concurrent languages. It is our opinion that the ideas of this thesis could also help in the design and implementation of real type systems for concrete distributed programming languages.

I would like to thank the members of the scientific committee, and I hope that this initiative will further contribute to strengthen the sense of belonging to the same community of all the young researchers that have accepted the challenges posed by any branch of theoretical computer science.

Rome, January 2016

Tiziana Calamoneri
President of the Italian Chapter of the EATCS

Preface

It is a pleasure for me to write a preface for Ornela Dardha's PhD thesis in the occasion of its publication in the Atlantis Studies in Computing, as recipient of a prize for "Best Italian 2015 PhD Thesis in Theoretical Computer Science" awarded by the Italian Chapter of EATCS.

I am happy that Ornela has obtained the prize, as a reward for the time and the energy that she has invested into research during the PhD period. Ornela's achievement is also gratifying for the Focus team and the whole Department of Computer Science of the University of Bologna, in which the thesis has been carried out. I like to think that Focus and the Department have provided a fertile environment in which her desire of learning and growing has been nourished.

The general topic of Ornela's thesis is type systems for programming languages. Type systems have been developed in sequential languages, initially with the goal of improving the efficiency of programs, and later also with the goals of ensuring certain correctness properties during execution and of specifying the inteded use of certain objects or components in a program. The application of type systems to concurrency is more recent. The field has presented, and still presents, a number of challenges: in a concurrent system new features, such as interactions, have to be taken into account; other features, such as dynamic reconfigurations, take a prominent role. Concurrency has sometimes led to the design of new type systems. A relevant example are the so-called session types, roughly types capable of specifying the protocols that a set of components should follow, in order to accomplish a certain task. The past two decades have seen a thorough investigation of session types.

Ornela's thesis shows how types can be used in presence of interactions and dynamic reconfiguration to guarantee some fundamental behavioural properties of distributed systems, such as forms of consistency, deadlock freedom, progress. Moreover, the thesis sheds light into the foundations of session types. The thesis shows that session types, at least in their most common format, are not a primitive concept, as they had been treated in the literature: they can be derived from more basic and well-known type constructs. This is important, both to understand better the concept, and to develop its metatheory.

I would like to conclude with my personal congratulations to Ornela for the work done, and my warmest wishes for her future.

Bologna, Gennaio 2016

Davide Sangiorgi

Acknowledgments

The work for the PhD thesis was carried out while I was a PhD student at the Computer Science Department of the University of Bologna and a member of the Focus team, and also during my one-year visit at the IT University of Copenhagen.

I am very grateful to my supervisor Davide Sangiorgi, who during my PhD has been of great support and guidance. I also want to thank the external reviewers of my PhD thesis: Ilaria Castellani and Vasco T. Vasconcelos for their careful work and useful feedbacks.

Currently I am a Research Associate at the School od Computing Science of the University of Glasgow, working with Simon J. Gay and supported by the UK EPSRC project *From Data Types to Session Types: A Basis for Concurrency and Distribution (ABCD)* (EP/K034413/1).

A very special thanks goes to Elena Giachino for her help and support, for the scientific and life-related advice she gave me during those years.

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.

During my one-year visit at ITU of Copenhagen, I had the pleasure to work with Marco Carbone and Fabrizio Montesi. Thank you for the very nice year at ITU and for making research a lot fun.

An enormous hug goes to all my friends around the world, especially the ones in Rome, Bologna, Copenhagen, Glasgow and London. Thank you for the great time together, for being of inspiration and support and above all for making me feel home whenever I visit you.

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, Titi e Ernesto: il tempo con voi non è mai abbastanza... Vi voglio un mondo di bene!

Last, but absolutely not least, I want to thank my Simon. During the time I was writing this book he has been very supportive, understanding and caring. You are truly a wonderful person!

Contents

Foreword iii

Preface v

Acknowledgements vii

List of Figures xiii

Introduction to the PhD Thesis 1

I Safe Dynamic Reconfiguration of Components 7

Introduction to Part I 9

1 Background on Components 13
- 1.1 Syntax . . . 13
- 1.2 Semantics . . . 17
  - 1.2.1 Runtime Syntax . . . 17
  - 1.2.2 Functions and Predicates . . . 18
  - 1.2.3 Evaluation of Pure and Guard Expressions . . . 19
  - 1.2.4 Reduction Rules . . . 21
- 1.3 Server and Client Example . . . 25

2 A Type System for Components 29
- 2.1 Typing Features . . . 29
- 2.2 Subtyping Relation . . . 30
- 2.3 Functions and Predicates . . . 32
- 2.4 Typing Rules . . . 35
- 2.5 Typing Rules for Runtime Configurations . . . 41

3 Properties of the Type System **43**
3.1 Main Results 43
3.2 Proofs 44

Related Work of Part I **53**

II Safe Communication by Encoding 57

Introduction to Part II **59**

4 Background on π-Types **63**
4.1 Syntax 63
4.2 Semantics 65
4.3 π-Types 66
4.4 π-Typing Rules 67
4.5 Main Results 71

5 Background on Session Types **73**
5.1 Syntax 75
5.2 Semantics 76
5.3 Session Types 78
5.4 Session Typing Rules 79
5.5 Main Results 83

6 Session Types Revisited **85**
6.1 Types Encoding 85
6.2 Terms Encoding 87
6.3 Properties of the Encoding 89
6.3.1 Auxiliary Results 90
6.3.2 Typing Values by Encoding 92
6.3.3 Typing Processes by Encoding 93
6.3.4 Operational Correspondence 102
6.4 Corollaries from the Encoding 109

III Advanced Features on Safety by Encoding 111

Introduction to Part III **113**

7 Subtyping **115**
7.1 Subtyping Rules . . . 115
7.2 Properties . . . 116

8 Polymorphism **121**
8.1 Parametric Polymorphism . . . 121
8.1.1 Syntax . . . 122
8.1.2 Semantics . . . 122
8.1.3 Typing Rules . . . 123
8.1.4 Encoding . . . 123
8.1.5 Properties of the Encoding . . . 124
8.2 Bounded Polymorphism . . . 127
8.2.1 Syntax . . . 127
8.2.2 Semantics . . . 129
8.2.3 Typing Rules . . . 130
8.2.4 Encoding . . . 131
8.2.5 Properties of the Encoding . . . 132

9 Higher-Order Communication **141**
9.1 Syntax . . . 141
9.2 Semantics . . . 142
9.3 Typing Rules . . . 143
9.3.1 HOπ Session Typing Rules . . . 143
9.3.2 HOπ Typing Rules . . . 146
9.4 Encoding . . . 146
9.5 Properties of the Encoding . . . 149
9.5.1 Typing HOπ Processes by Encoding . . . 149
9.5.2 Operational Correspondence for HOπ . . . 157

10 Recursion **161**
10.1 Syntax . . . 161
10.2 Semantics . . . 162
10.3 Typing Rules . . . 163
10.4 Encoding . . . 165
10.5 Properties of the Encoding . . . 166

11 From π-Types to Session Types **169**
11.1 Further Considerations . . . 169
11.2 Typed Behavioural Equivalence . . . 170
11.2.1 Equivalence Results for the Encoding . . . 171

Related Work of Part II and III **173**

IV Progress of Communication 177

Introduction to Part IV **179**

12 Background on π-types for Lock Freedom **183**
12.1 Syntax . . . 183
12.2 Semantics . . . 183
12.3 π-Types for Lock Freedom . . . 185
12.4 π-Typing Rules for Lock Freedom . . . 187

13 Background on Session Types for Progress **193**
13.1 Syntax . . . 193
13.2 Semantics . . . 193
13.3 Session Types . . . 195
13.4 Session Typing Rules . . . 195

14 Progress as Compositional Lock Freedom **197**
14.1 Lock Freedom for Sessions . . . 197
14.2 Progress for Sessions . . . 199
14.3 Lock Freedom meets Progress . . . 200
14.3.1 Properties of Closed Terms . . . 200
14.3.2 Properties of Open Terms . . . 202
14.4 A Type System for Progress . . . 205

Related Work of Part IV **207**

Bibliography 211

List of Figures

1.1 Component extension of core ABS 14
1.2 Runtime syntax 18
1.3 Evaluation of pure expressions 19
1.4 Evaluation of guard expressions 21
1.5 Reduction rules for configurations (1) 22
1.6 Reduction rules for configurations (2) 23
1.7 Reduction rules for configurations (3) 24
1.8 Reduction rules for rebinding 25
1.9 Workflow in ABS 26
1.10 Workflow in the component model 27
1.11 Client and controller objects creation 27

2.1 Subtyping relation 31
2.2 Lookup functions 33
2.3 Auxiliary functions and predicates 34
2.4 Typing rules for the functional level 36
2.5 Typing rules for expressions with side effects 37
2.6 Typing rules for statements 38
2.7 Typing rules for declarations 39
2.8 Typing the workflow example 40
2.9 Typing rules for runtime configurations 42

4.1 Syntax of the standard π-calculus 63
4.2 Structural congruence for the standard π-calculus 65
4.3 Rules for equational reasoning 65
4.4 Semantics of the standard π-calculus 66
4.5 Syntax of linear π-types 66
4.6 Combination of π-types and typing contexts 68
4.7 Type duality for linear π-types 69
4.8 Typing rules for the standard π-calculus 70

5.1 Syntax of the π-calculus with sessions 75

5.2 Structural congruence for the π-calculus with sessions 76
5.3 Semantics of the π-calculus with sessions 77
5.4 Syntax of session types 78
5.5 Type duality for session types 79
5.6 Context split and context update 80
5.7 Typing rules for the π-calculus with sessions 81

6.1 Encoding of session types 86
6.2 Encoding of session terms 87
6.3 Encoding of session typing contexts 90

7.1 Subtyping rules for the π-calculus with sessions 116
7.2 Subtyping rules for the standard π-calculus 116

8.1 Syntax of parametric polymorphic constructs 122
8.2 Typing rules for parametric polymorphic constructs 123
8.3 Encoding of parametric polymorphic constructs 124
8.4 Syntax of bounded polymorphic session constructs 128
8.5 Syntax of bounded polymorphic π-constructs 129
8.6 Typing rules for bounded polymorphic session constructs 130
8.7 Typing rules for bounded polymorphic π-constructs 131
8.8 Encoding of bounded polymorphic types 131
8.9 Encoding of bounded polymorphic terms 132

9.1 Syntax of higher-order constructs 141
9.2 Semantics of higher-order constructs 142
9.3 Typing rules for the HOπ with sessions: values 144
9.4 Typing rules for the HOπ with sessions: processes 145
9.5 Typing rules for the standard HOπ: values 147
9.6 Typing rules for the standard HOπ: processes 148
9.7 Encoding of HOπ types and terms 149

10.1 Syntax of recursive session types and terms 161
10.2 Syntax of recursive standard π-calculus types and terms 162
10.3 Typing rules for recursive constructs 164
10.4 Encoding of recursive types, terms and typing contexts 165

12.1 Syntax of the standard π-calculus: repeated 184
12.2 Semantics of the standard π-calculus: repeated 184
12.3 Syntax of usage types 185
12.4 Typing rules for the π-calculus with usage types 190

13.1 Syntax of the π-calculus with sessions: updated 194

13.2 Semantics of the π-calculus with sessions: updated 194
13.3 Syntax of session types: updated 195
13.4 Typing rules for the π-calculus with sessions: updated 196

14.1 Checking progress with TyPiCal 206

Introduction to the PhD Thesis

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 [90]. 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 of *correctness* and *safety* properties in software programs. However, 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 systems is what guides this thesis.

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) [62]. 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 contractually 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 PhD and Methodology

The aim of the PhD 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 have adopted types and type systems in our work 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 readabil-

ity. 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 [57, 58, 72, 83, 98, 99, 101, 104] 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 [36, 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 [66, 70, 73], or a stronger property, that of lock freedom [67, 68, 74] 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 of components communication [50, 51].

ii) There are several types and type system proposals for *communication*, starting from the standard channel types in the typed π- calculus [72, 83, 98, 101] to the *behavioural types* [17, 21, 38, 57, 58, 88, 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 independently. In this thesis, 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 focus on session types because they guarantee several safety properties, such as privacy of the communication channel, communication safety and session fidelity, ensuring that the type of the transmitted data and the structure of the session type are as expected. However, as previously stated, we are also interested in studying stronger properties, such as deadlock and lock freedom of communicating participants and progress of a session. Again, these properties can be guaranteed by using session types.

Contribution

The contributions of this thesis 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 session typed π-calculus into the standard typed π-calculus, by showing that the type and term primitives of the former can be obtained by using the primitives of the latter. 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 types. The importance of the encoding is foundational, since
  - The encoding is proved faithful as it allows the derivation of properties of the session π-calculus, for e.g., subject reduction, by exploiting the theory of the standard typed π-calculus.
  - The encoding is proved robust by extending it to handle non trivial features like, subtyping, polymorphism, higher-order communication and recursion and by using it to derive new properties in the session π-calculus due to these new features from the corresponding ones in the standard typed π-calculus.
  - The encoding is an expressiveness result for the standard π-calculus. There are many more expressiveness results in the untyped settings as opposed to expressiveness 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, lock freedom and progress. We study the relation among these properties and present a type system for guaranteeing the progress property by exploiting our encoding.

Structure of the Thesis

The structure of the thesis 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 [31]. 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 [32]. 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 wrt 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 [77] the authors propose to solve this problem by integrating notions coming from component models [4,11,14,29,82] within the *Abstract Behavioural Specification* programming language (ABS) [63].

We start this thesis 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 thesis) 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 [63] 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, [77] 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, [77] 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 thesis, 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 thesis, 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 Fig. 1.1 is an extension of the ABS language [63] and is mainly inspired by the component calculus in [77]. 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 [77]. On the other hand, we include in the present calculus the component primitives from [77], 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 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 [77]. 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, respectively;

P	::=	$\overline{Dl}\ \{\ s\ \}$	(program)
Dl	::=	$D \mid F \mid I \mid C$	(declaration)
T	::=	$\mathtt{V} \mid \mathtt{D}[\langle\overline{T}\rangle] \mid (\mathtt{I}, \mathbb{r}) \mid (\mathtt{C}, \mathbb{r})$	(type)
$\mathbb{r}$	::=	$\bot \mid \mathtt{G}[\overline{f : T}] \mid \alpha \mid \overline{\mu\alpha.\mathbb{r}}$	(record)
D	::=	**data** $\mathtt{D}[\langle\overline{T}\rangle] = \mathtt{Co}[(\overline{T})] \vert \overline{\mathtt{Co}[(\overline{T})]};$	(data type)
F	::=	**def** $T\ \mathtt{fun}[\langle\overline{T}\rangle](\overline{T\ x}) = e;$	(function)
I	::=	**interface** $\mathtt{I}$ [**extends** $\overline{\mathtt{I}}$] { $\overline{\textbf{port}\ T\ x}; \overline{S}$ }	(interface)
C	::=	**class** $\mathtt{C}[(\overline{T\ x})]$ [**implements** $\overline{\mathtt{I}}$] { $\overline{Fl}\ \overline{M}$ }	(class)
Fl	::=	[**port**] $T\ x$	(field declaration)
S	::=	[**critical**] $(\mathcal{G}, \mathbb{r})\ T\ \mathtt{m}(\overline{T\ x})$	(method header)
M	::=	$S\ \{\ s\ \}$	(method definition)
s	::=	**skip** $\mid s\ ;\ s \mid T\ x \mid x = z \mid$ **await** g	(statement)
	$\mid$	**if** e **then** s **else** $s \mid$ **while** $e\ \{\ s\ \} \mid$ **return** e	
	$\mid$	**rebind** $e.x = z \mid$ **suspend**	
z	::=	$e \mid$ **new** [**cog**] $\mathtt{C}\ (\overline{e}) \mid e.\mathtt{m}(\overline{e}) \mid e!\mathtt{m}(\overline{e}) \mid$ **get**(e)	(expression w/ side effects)
e	::=	$v \mid x \mid \mathtt{fun}(\overline{e}) \mid$ **case** $e\ \{\overline{p \Rightarrow e_p}\} \mid \mathtt{Co}[(\overline{e})]$	(pure expression)
v	::=	`true` $\mid$ `false` $\mid$ **null** $\mid \mathtt{Co}[(\overline{v})]$	(value)
p	::=	$\_ \mid x \mid$ **null** $\mid \mathtt{Co}[(\overline{p})]$	(pattern)
g	::=	$e \mid x? \mid \|\|e\|\| \mid g \wedge g$	(guard)

Figure 1.1: Component extension of core ABS

`V` ranges over type variables for polymorphism; `G` ranges over cog names –which will be explained in details later on; `D`, `Co` and `fun` range respectively over data type, constructor and function names; `m` and x range respectively over method names and variables, in particular x ranges over fields and ports. For readability, we will often use f for fields and p for ports, or just f for both fields and ports in order to distinguish them from other variables. There are also some special variables, like **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. Finally, for simplicity we let L be either a class name `C` or an interface name `I`. 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 [61].

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 `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` or class name `C` and a *record* $\mathbb{r}$ to type objects. The pair $(\texttt{C}, \mathbb{r})$ is only used to type object **this**. Records are a new concept and are associated to types in order to track group information. The previous calculi, neither ABS [63] nor its component extension [77] 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. We work up-to unfolding, meaning equating a record and its unfolding.

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` | `Cons`(`Int`, `IntList`), or parametric data types **data** `List`$\langle T \rangle$ = `Nil` | `Cons`(T, `List`$\langle T \rangle$), or predefined data types like **data** `Bool` = `true` | `false` ; or **data** `Int` ; or **data** `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** $e.x = z$ rebinds the port x 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; x? which is `true` when the future x 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, which is defined as a transition system on the runtime configurations. Hence, we first define the runtime configurations and then give some auxiliary functions which the operational semantics relies on.

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. 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 field. 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 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 starting from the initial state of the program.

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

Figure 1.2: Runtime syntax

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 $\mathtt{m}$ in class $\mathtt{C}$ with **this** bound to $\mathtt{o}$, **destiny** bound to $\mathtt{f}$, 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*:

$$\frac{\mathbf{class}\ \mathtt{C} \ldots \{T\ \mathtt{m}(\overline{T}x)\{\overline{T'\ x'}\ s\} \ldots\} \in N}{\mathrm{bind}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}, \mathtt{C}) = \{\overline{T\ x = v}; \overline{T'\ x' = \mathbf{null}}; \mathbf{this} = \mathtt{o} \mid s\}}\ (\text{NM-Bind})$$

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

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

$$\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i'}{\sigma \vdash \mathsf{Co}(e_1 \ldots e_i \ldots e_n) \rightsquigarrow \sigma' \vdash \mathsf{Co}(e_1 \ldots e_i' \ldots e_n)}\ (\textsc{RedCons}) \qquad \frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i'}{\sigma \vdash \mathtt{fun}(e_1 \ldots e_i \ldots e_n) \rightsquigarrow \sigma' \vdash \mathtt{fun}(e_1 \ldots e_i' \ldots e_n)}\ (\textsc{RedFunExp})$$

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

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

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

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

Figure 1.3: Evaluation of pure expressions

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

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

1.2.3 Evaluation of Pure and Guard Expressions

In this section we present the evaluation of pure and guard expressions, before introducing the operational semantics for 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

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

Rule (RedCons) states that the expression $\texttt{Co}(e_1 \dots e_i \dots e_n)$ reduces to $\texttt{Co}(e_1 \dots e'_i \dots e_n)$ 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 $\texttt{fun}(\overline{T\ x}) = e$, and we denote by $\overline{x}_{\texttt{fun}}$ the list of formal parameters $\overline{x}$ and by $\overline{e}_{\texttt{fun}}$ 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 $\texttt{fun}(\overline{v})$ in a context σ reduces to the evaluation of $\overline{e}_{\texttt{fun}}[\overline{x}_{\texttt{fun}} \mapsto \overline{y}]$ 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 $\overline{x}_{\texttt{fun}}$ 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), previously defined. 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.

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.

Rules (RedReply1) and (RedReply2) assert that the guard $x?$ evaluates to true, whenever the value associated to the evaluation of x is ready to be retrieved, namely is different from $\bot$; otherwise, it evaluates to false. Rule (RedConj) is straightforward and asserts the evaluation of boolean conjunctions of guards g_1 and g_2. Rules (RedCS1) and (RedCS2) are the new evaluation rules of the component calculus, wrt ABS. They state that, when in the object o the field $\texttt{nb}_{cr}$ is different from zero, then the object has an open critical section and hence the test

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

$$\frac{\sigma \vdash x \rightsquigarrow \sigma \vdash \mathtt{f} \quad \mathit{fut}(\mathtt{f}, \bot) \in N}{\sigma, N \vdash x? \rightsquigarrow \sigma, N \vdash \mathtt{false}} \text{(RedReply2)}$$

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

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

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

Figure 1.4: Evaluation of guard expressions

$\|e\|$ returns `true`; otherwise, if $\mathtt{nb}_{cr} = 0$ it means that no critical section is open and $\|e\|$ evaluates to `false`.

1.2.4 Reduction Rules

In this section we introduce the operational semantics for configurations. The reduction rules are given in Fig. 1.5, 1.6, 1.7 and 1.8.

We start with Fig. 1.5. Rule (Context) is straightforward. 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**. 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 entry to **idle** meaning that there is no active object in the cog. Rule (Activate), as opposed to (Suspend), selects a task p from the

$$\frac{N \to N'}{N\ N'' \to N'\ N''}\ (\textsc{Context}) \qquad \frac{}{ob(\mathsf{o}, \sigma, \{\sigma' \mid \textbf{skip}; s\}, Q) \to ob(\mathsf{o}, \sigma, \{\sigma' \mid s\}, Q)}\ (\textsc{Skip})$$

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

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

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

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

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

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

$$\frac{}{ob(\mathsf{o}, \sigma, \{\sigma' \mid \textbf{suspend}; s\}, Q) \to ob(\mathsf{o}, \sigma, \textbf{idle}, Q \cup \{\sigma' \mid s\})}\ (\textsc{Suspend})$$

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

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

Figure 1.5: Reduction rules for configurations (1)

queue of suspended processes and activates it. The process is removed from the queue and the *cog* configuration is updated accordingly.

Now we move to Fig. 1.6. 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 freshly created cog, respectively. The names of

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

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

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

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

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

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

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

Figure 1.6: Reduction rules for configurations (2)

the object and the cog created are globally unique. The object's fields are given default values by applying function atts($C, \overline{v}, \mathtt{o}', \mathtt{c}$).

We comment now on the rules in Fig. 1.7. Rule (Self-Sync-Call) looks up 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 from o to o′, by thus respecting

(Self-Sync-Call)

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

(Self-Sync-Return-Sched)

$$\frac{\sigma'(\mathbf{destiny}) = \mathtt{f}}{\mathit{ob}(\mathtt{o}, \sigma, \{\sigma'' \mid \mathbf{cont}(\mathtt{f})\}, Q \cup \{\sigma' \mid s\}) \rightarrow \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' \mid s\}, Q)}$$

(Async-Call)

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

(Cog-Sync-Call)

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

(Cog-Sync-Return-Sched)

$$\frac{\sigma''(\mathit{cog}) = \mathtt{c} \qquad \sigma'''(\mathbf{destiny}) = \mathtt{f}}{\begin{array}{c}\mathit{ob}(\mathtt{o}, \sigma, \{\sigma' \mid \mathbf{cont}(\mathtt{f})\}, Q \quad \mathit{ob}(\mathtt{o}', \sigma'', \mathbf{idle}, Q' \cup \{\sigma''' \mid s\})) \quad \mathit{cog}(\mathtt{c}, \mathtt{o}) \\ \rightarrow \mathit{ob}(\mathtt{o}, \sigma, \mathbf{idle}, Q) \quad \mathit{ob}(\mathtt{o}', \sigma'', \{\sigma''' \mid s\}, Q') \quad \mathit{cog}(\mathtt{c}, \mathtt{o}')\end{array}}$$

Figure 1.7: Reduction rules for configurations (3)

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).

Finally, we comment on the reduction rules for rebinding of ports in Fig. 1.8. 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

$$\frac{\sigma(\mathtt{nb}_{cr}) = 0 \quad \mathsf{o} = [\![e]\!]_{(\sigma\circ\sigma')} \quad v = [\![e']\!]_{(\sigma\circ\sigma')}}{ob(\mathsf{o}, \sigma, \{\ \sigma' \mid \textbf{rebind}\ e.x = e'; s\ \}, Q) \rightarrow ob(\mathsf{o}, \sigma[x \mapsto v], \{\ \sigma' \mid s\ \}, Q)} \text{(REBIND-LOCAL)}$$

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

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

Figure 1.8: Reduction rules for rebinding

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 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

```
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

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 [77].

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.

```
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 in 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 the type system for the component model. We first give a thorough explanation of the types we adopt and how the type system achieves 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 [77]. 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 the 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 $\mathtt{r}$, 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 (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 sets, namely $\mathcal{G} \uplus \mathcal{G}'$, where the empty set acts as the neutral element, namely $\mathcal{G} \uplus \emptyset = \emptyset \uplus \mathcal{G} = \mathcal{G}$. We will discuss the details in Section 2.4.

Method Signature Let us now explain the method signature $(\mathcal{G}, \mathbb{r})$ used to annotate a method header. The record $\mathbb{r}$ is used as the record of the object **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, notice that data types also may 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 may include the records of the objects.

2.2 Subtyping Relation

There are two forms of subtyping: *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. In the latter, subtyping relation is established 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. (Featherweight) Java uses nominal subtyping, lan-

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

$$\frac{f \notin \mathit{ports}(L)}{(L, \mathtt{G}[f : T; \overline{f : T}]) \leq (L, \mathtt{G}[\overline{f : T}])} \text{(S-Fields)} \qquad \frac{f \in \mathit{ports}(L)}{(L, \mathtt{G}[\overline{f : T}]) \leq (L, \mathtt{G}[f : T; \overline{f : T}])} \text{(S-Ports)}$$

$$\frac{\textbf{class } \mathtt{C}[(\overline{T\ x})] \textbf{ implements } \overline{\mathtt{I}} \{ \overline{Fl}\ \overline{M} \} \quad \mathtt{I}_i \in \overline{\mathtt{I}}}{\mathtt{C} \leq \mathtt{I}_i} \text{(S-Class)} \qquad \frac{}{T \leq T} \text{(S-Refl)}$$

$$\frac{\textbf{interface } \mathtt{I} \textbf{ extends } \overline{\mathtt{I}} \{ \overline{\textbf{port } T\ x}; \overline{S} \} \quad \mathtt{I}_i \in \overline{\mathtt{I}}}{\mathtt{I} \leq \mathtt{I}_i} \text{(S-Interface)} \qquad \frac{T \leq T' \quad T' \leq T''}{T \leq T''} \text{(S-Trans)}$$

Figure 2.1: Subtyping relation

guages like [44, 52, 81, 92] use structural subtyping. In [33] the authors integrate both nominal and structural subtyping.

The subtyping relation $\leq$ for our language is given in Fig. 2.1; we adopt both nominal and structural subtyping. Rule (S-Data) states that data types are covariant in their type parameters. Rule (S-Type) states that annotating classes and interfaces with records does not change the subtyping order. 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 modifications if the port is not available. Rules (S-Class)

and (S-Interface) use nominal subtyping and state that a class C (respectively, an interface I) is a subtype of an interface $\mathtt{I}_i$ that it implements (respectively, extends). Rules (S-Refl) and (S-Trans) are standard and state that our subtyping relation is a preorder.

2.3 Functions and Predicates

In this section we define the auxiliary functions and predicates that are used in the typing rules. We 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 [61]. 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 V to a type T returns a substitution of 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 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 context 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 contexts corresponding to patterns in $\overline{p}$. The pair $(\mathtt{I}, \mathtt{G}[\sigma \uplus \sigma'(\overline{f : (\mathtt{I}, \mathbb{r})})])$ 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. A class C implements an interface I if the ports of C are at *most* the ones of I. Instead, for methods, C may define at *least* the methods declared in I having the

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

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

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

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

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

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

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

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

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

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

Figure 2.2: Lookup functions

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

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

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

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

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

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

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

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

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

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

Figure 2.3: Auxiliary functions and predicates

same signature. The extends predicate states when an interface `I` properly extends another interface `I'` and is defined similarly to the implements predicate.

2.4 Typing Rules

A *typing context* Γ is a partial function and 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 $\mathsf{dom}(\Gamma)$ denotes the domain of the typing context Γ. We define the *composition* of typing contexts, $\Gamma \circ \Gamma'$, 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 have the following forms, where a cogset $\mathcal{G}$ indicates the set of new cogs created by the term being typed. $\Gamma \vdash g : \mathtt{Bool}$ 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; $\Gamma \vdash M$ for method declarations; $\Gamma \vdash C$ for class declarations and $\Gamma \vdash I$ for interface declarations.

Pure Expressions The typing rules for pure expressions 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-FieldR) assigns to x a type T and a record $\mathbb{r}$ fetched from the type of **this**. Rule (T-FieldBot) assigns $(T, \bot)$ to x, since x is not part of the record for **this** but is a field of `C`. 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 expressions $\overline{e}$ is of type $\sigma(T')$ whenever the constructor is of a functional type $\overline{T} \to T'$ and the expressions are of type $\overline{T'}$; where the auxiliary function tmatch applied on the formal types $\overline{T}$ and the actual ones $\overline{T'}$ returns the substitution σ. Rule (T-FunExp) is similar to the previous one for constructor expressions, namely, the application of the function `fun` to a list of expressions $\overline{e}$ is of type $\sigma(T')$ whenever the function is of a functional type $\overline{T} \to T'$ and the expressions are of type $\overline{T'}$, and again tmatch is applied to obtain 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. Rule (T-Sub) is the standard subsumption rule, which uses the subtyping relation defined in Section 2.2.

$$\text{(T-Var/Field)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-FieldR)}\ \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma(\mathbf{this}) = (\mathsf{C}, \mathsf{G}[x : (T, \mathbb{r}), \dots])}{\Gamma \vdash x : (T, \mathbb{r})}$$

$$\text{(T-FieldBot)}\ \frac{x \notin \mathsf{dom}(\Gamma) \qquad T\ x \in \mathit{fields}(\mathsf{C}) \qquad \Gamma(\mathbf{this}) = (\mathsf{C}, \mathsf{G}[\overline{x : T}]) \qquad x \notin \overline{x}}{\Gamma \vdash x : (T, \bot)} \qquad \text{(T-Null)}\ \frac{\mathbf{interface}\ \mathtt{I}\ [\cdots]\ \{\ \cdots\ \} \in CT}{\Gamma \vdash \mathbf{null} : (\mathtt{I}, \mathbb{r})} \qquad \text{(T-Wild)}\ \frac{}{\Gamma \vdash \_ : T}$$

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

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

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

Figure 2.4: Typing rules for the functional level

Guard Expressions The typing rules for guard expressions are given at the bottom of Fig. 2.4. Rule (T-FutGuard) states that if a variable x has type $\mathtt{Fut}\langle T \rangle$, 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 $\mathtt{Fut}\langle T \rangle$. Rule (T-New) assigns type T to the object

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

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

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

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

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

Figure 2.5: Typing rules for expressions with side effects

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 $\text{crec}(\texttt{G}, \texttt{C}, \sigma)$ predicate, which means that T is of the form $(\texttt{I}, \texttt{G}[\overline{f : \sigma(\texttt{I}, \mathbb{r})}])$ and $\text{implements}(\texttt{C}, \texttt{I})$ and σ is obtained by the function tmatch. Rule (T-NewCog) 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 i.e., $(\mathcal{G}, \mathbb{r})(\overline{T\ x}) \to T$. 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. 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 respectively in the substitution $\sigma(T)$ and $\texttt{Fut}\langle\sigma(T)\rangle$, with T being the formal return type. Rule (T-SCall) checks whether the group of

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

(T-Assign) $\dfrac{\Gamma(x) = T \qquad \Gamma, \mathcal{G} \vdash z : T}{\Gamma, \mathcal{G} \vdash x = z}$ (T-AssignFieldR) $\dfrac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma, \mathcal{G} \vdash z : T \qquad \Gamma(\textbf{this}) = (\mathtt{C}, \mathtt{G}[x : T, \ldots])}{\Gamma, \mathcal{G} \vdash x = z}$

(T-AssignFieldBot) $\dfrac{x \notin \mathsf{dom}(\Gamma) \qquad T\ x \in \mathit{fields}(\mathtt{C}) \qquad \Gamma(\textbf{this}) = (\mathtt{C}, \mathtt{G}[\overline{x : T}]) \qquad \Gamma, \mathcal{G} \vdash z : T \qquad x \notin \overline{x}}{\Gamma, \mathcal{G} \vdash x = z}$ (T-Await) $\dfrac{\Gamma \vdash g : \mathtt{Bool}}{\Gamma, \emptyset \vdash \textbf{await}\ g}$

(T-Cond) $\dfrac{\Gamma \vdash e : \mathtt{Bool} \qquad \Gamma, \mathcal{G}_1 \vdash s_1 \qquad \Gamma, \mathcal{G}_2 \vdash s_2}{\Gamma, \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2}$ (T-While) $\dfrac{\Gamma \vdash e : \mathtt{Bool} \qquad \Gamma, \emptyset \vdash s}{\Gamma, \emptyset \vdash \textbf{while}\ e\ \{\ s\ \}}$

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

(T-RebindBot) $\dfrac{T\ x \in \mathit{ports}(\mathtt{I}) \qquad x \notin \overline{x} \qquad \Gamma(\textbf{this}) = (\mathtt{C}, \mathtt{G}[\overline{x : T}]) \qquad \Gamma \vdash e : (\mathtt{I}, \mathbb{r}) \qquad \Gamma, \mathcal{G} \vdash z : T \qquad \mathrm{coloc}(\mathbb{r}, \Gamma(\textbf{this}))}{\Gamma, \mathcal{G} \vdash \textbf{rebind}\ e.x = z}$

Figure 2.6: Typing rules for statements

this and the group of the callee coincide, by using the auxiliary function coloc, whether this check is not performed in rule (T-ACall).

Statements The typing rules for statements are given 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

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

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

$$\text{(T-Interface)}\quad \frac{\forall \texttt{I}' \in \overline{\texttt{I}}.\ \text{extends}(\texttt{I}, \texttt{I}')}{\emptyset \vdash \textbf{interface}\ \texttt{I}\ \textbf{extends}\ \overline{\texttt{I}}\ \{\ \overline{\textbf{port}\ T\ x; S}\ \}}$$

Figure 2.7: Typing rules for declarations

uses the disjoint union $\uplus$ of the corresponding cogsets. Rule (T-Assign) states 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 z. Rule (T-AssignFieldR) and rule (T-AssignFieldBot) deal with the assignment $x = z$ when field x is not present in $\mathsf{dom}(\Gamma)$ and they follow the same idea as rules (T-FieldR) and (T-FieldBot), respectively. The main difference in the premises of rule (T-AssignFieldR) and rule (T-AssignFieldBot) is the fact that in the former rule x is in the record of **this**, whether in the latter rule x is not in the record of **this** but it is a field of the class of **this**. Rule (T-Await) asserts that **await** g is well typed whenever the guard g has type `Bool`. Rules (T-Cond) and (T-While) are quite standard, except for the presence of the linear set of cog names: the typing of the conditional statement follows the same principle as the composition of statements in rule (T-Comp); the typing of the loop uses instead an empty set of cogs. 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) types the statement **rebind** $e.x = z$ by checking that: *i*) x 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 $\text{coloc}(\mathbb{r}, \Gamma(\textbf{this}))$. Rule (T-RebindBot) is similar but it deals with the case when x is not present in the record of **this**, namely it is assigned to $\bot$.

Declarations The typing rules for declarations of methods, classes and interfaces are presented in Fig. 2.7. Rule (T-Method) states that method `m` is well typed in class `C` if the method's body s is well typed in a typing context augmented with the method's typed parameters; **destiny** being of type $\texttt{Fut}\langle\sigma(T)\rangle$ and **this** being of

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

Figure 2.8: Typing the workflow example

type $(\texttt{C}, \sigma(\mathbb{r}))$. A substitution σ is used to obtain the actual values starting from the formal ones. Rule (T-Class) states that a class C is well typed when it implements all the interfaces $\overline{\texttt{I}}$ and all its methods are well typed. Finally, rule (T-Interface) states that an interface I is well typed if it extends all interfaces in $\overline{\texttt{I}}$.

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 m does **this**.f = $\texttt{z}_1$ and n does **this**.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 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 [51], 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 $\texttt{c}_2$ to $\texttt{c}_n$. Let us now try to typecheck client $\texttt{c}_1$. If we try to typecheck the rebinding operation, we would have the following typing judgement in the premise of (T-Rebind):

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

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

2.5 Typing Rules for Runtime Configurations

In this section we present the typing rules for runtime configurations, introduced in Section 1.2. In order to prove the subject reduction property, typing rules for runtime configurations are needed and are presented 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:

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

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. A future value $\mathtt{f}$ is assigned type future $\mathtt{Fut}\langle T \rangle$ and a cog identifier $\mathtt{c}$ is assigned a cog name $\mathtt{G}$.

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**($\mathtt{f}$), which is a new statement added to the runtime syntax, is well typed whenever $\mathtt{f}$ is a future. Rule (T-Future1) states that the configuration $\mathit{fut}(\mathtt{f}, v)$ is well typed if the future $\mathtt{f}$ has type $\mathtt{Fut}\langle T \rangle$ where T is the type of v. Instead, rule (T-Future2) states that $\mathit{fut}(\mathtt{f}, \bot)$ is well typed whenever $\mathtt{f}$ 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 sets of cog names. Rule (T-Cog) asserts that a group configuration $\mathit{cog}(\mathtt{c}, \mathtt{o}_\epsilon)$ is well typed if $\mathtt{c}$ is declared to be associated to $\mathtt{G}$ in Δ. Rules (T-Empty) and (T-Idle) are straightforward. Rule (T-Object) states that an object is well typed whenever: *i*) the declared record of $\mathtt{o}$ is the same as the one associated to $\mathtt{c}$; *ii*) its fields are well typed and *iii*) its running process and process queue are well typed. Finally, (T-Invoc) states that $\mathit{invoc}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})$ is well typed under substitution σ when: *i*) callee $\mathtt{o}$ is assigned type $(\mathtt{C}, \sigma(\mathbb{r}))$; *ii*) future $\mathtt{f}$ is of type $\mathtt{Fut}\langle\sigma(T)\rangle$ and *iii*) values $\overline{v}$ are typed accordingly by applying substitution σ, namely $\overline{\sigma(T)}$.

$$\text{(T-Weak1)}\ \frac{\Gamma, \mathcal{G} \vdash z : T \qquad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R z : T} \qquad \text{(T-Weak2)}\ \frac{\Gamma, \mathcal{G} \vdash s \qquad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R s} \qquad \text{(T-Weak3)}\ \frac{\Gamma, \mathcal{G} \vdash Dl \qquad \Gamma \subseteq \Delta}{\Delta, \mathcal{G} \vdash_R Dl}$$

$$\text{(T-State)}\ \frac{\Delta(x) = T \qquad \Delta \vdash_R v : T}{\Delta, \emptyset \vdash_R T\ x\ v} \qquad \text{(T-Cont)}\ \frac{\Delta(\mathtt{f}) = \mathtt{Fut}\langle T \rangle}{\Delta, \emptyset \vdash_R \mathbf{cont}(\mathtt{f})}$$

$$\text{(T-Future1)}\ \frac{\Delta(\mathtt{f}) = \mathtt{Fut}\langle T \rangle \qquad \Delta \vdash_R v : T}{\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, v)} \qquad \text{(T-Future2)}\ \frac{\Delta(\mathtt{f}) = \mathtt{Fut}\langle T \rangle}{\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, \bot)}$$

$$\text{(T-Process-Queue)}\ \frac{\Delta, \mathcal{G} \vdash_R Q \qquad \Delta, \mathcal{G}' \vdash_R Q'}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R Q \cup Q'}$$

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

$$\text{(T-Empty)}\ \frac{}{\Delta, \emptyset \vdash_R \epsilon} \qquad \text{(T-Idle)}\ \frac{}{\Delta, \emptyset \vdash_R \mathbf{idle}}$$

$$\text{(T-Object)}\ \frac{\begin{array}{c}\Delta(\mathtt{o}) = (\mathtt{C}, \mathtt{G}[\overline{f : T}]) \qquad \Delta(\mathtt{c}) = \mathtt{G} \\ \mathit{fields}(\mathtt{C}) = \overline{T}\ \overline{f} \qquad \Delta, \overline{f} : \overline{T}, \emptyset \vdash_R \overline{T}\ \overline{f}\ \overline{v} \\ \Delta, \overline{f} : \overline{T}, \mathcal{G} \vdash_R K_{\mathbf{idle}} \qquad \Delta, \overline{f} : \overline{T}, \mathcal{G}' \vdash_R Q\end{array}}{\Delta, \mathcal{G} \uplus \mathcal{G}' \vdash_R \mathit{ob}(\mathtt{o}, \overline{T}\ \overline{f}\ \overline{v},\ \mathit{cog}\ \mathtt{c}; \theta, K_{\mathbf{idle}}, Q)}$$

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

Figure 2.9: Typing rules for runtime configurations

Chapter 3

Properties of the Type System

3.1 Main Results

In this section we present the main results regarding our type system. We start with subject reduction for expressions, then we present subject reduction for configurations and finally we conclude with the correctness theorems, the main result of this part. Intuitively the latter theorems state that well-typed programs do not perform illegal rebinding or illegal 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 (Subject Reduction for Expressions)**.** Let Γ be a typing context 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 context Γ' such that $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \sigma'$ and $\Gamma' \vdash e' : T$.

The type system is proven correct in a Wright-Felleisen style [116], namely we prove the 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 for Configurations)**.** If $\Delta, \mathcal{G} \vdash_R N$ and $N \rightarrow N'$ then $\exists\, \Delta',\ \mathcal{G}'$ such that $\Delta \subseteq \Delta'$, $\mathcal{G} \subseteq \mathcal{G}'$ and $\Delta', \mathcal{G}' \vdash_R N'$.

Theorem 3.1.3 (Correctness of Rebindings)**.** If $\Delta, \mathcal{G} \vdash_R N$, then for all objects $ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q) \in N$ and $s \equiv$ **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 (Correctness of Sync Method Calls)**.** If $\Delta, \mathcal{G} \vdash_R N$, then for all objects $ob(\mathsf{o}, \sigma, \{\ \sigma' \mid s\ \}, Q) \in N$ and $s \equiv x = e.\mathtt{m}(\overline{e}); s'$ 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

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 the typing judgements for configurations $\Delta, \mathcal{G} \vdash_R N$.

□

Proof of Lemma 3.1.1 on Subject Reductions for Exprs: Let Γ be a typing context 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 context 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 $\Gamma \vdash \mathtt{Co}(e_1 \ldots e_i \ldots e_n) : T$. Since $\Gamma \subseteq \Gamma'$, the $\Gamma' \vdash \mathtt{Co}(e_1 \ldots e'_i \ldots e_n) : T$.

- Case (RedFunExp): This case follows exactly the same line as (RedCons). By induction hypothesis $\Gamma \vdash e_i : T_i$ and since $\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e'_i$ for $1 \leq i \leq n$, the $\Gamma' \vdash e'_i : T_i$ and $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \sigma'$. By assumption we have $\Gamma \vdash \mathtt{fun}(e_1 \ldots e_i \ldots e_n) : T$. 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 rule (T-FunExp) we have $\Gamma \vdash \overline{v} : \overline{T}$ and $\Gamma(\mathtt{fun}) = \overline{T'} \rightarrow T'$, and there is a type substitution ρ such that $\overline{T} = \rho(\overline{T'})$ and $T = \rho(T')$. It is the case that $\Gamma, \overline{x}_{\mathtt{fun}} : \rho(\overline{T'}) \vdash \overline{x}_{\mathtt{fun}} : \overline{T'}$. By rule (T-FunDecl) $\Gamma, \overline{x}_{\mathtt{fun}} : \overline{T'} \vdash e_{\mathtt{fun}} : T'$. Since typing is preserved by substitution, then $\Gamma, \overline{x}_{\mathtt{fun}} : \rho(\overline{T'}) \vdash e_{\mathtt{fun}} : \rho(T')$. 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 $\Gamma \vdash \textbf{case}\ e\ \{\overline{p \Rightarrow e_p}\} : T'$. 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 $\Gamma' \vdash \textbf{case}\ e'\ \{\overline{p \Rightarrow e_p}\} : T'$.

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

- Case (RedCase3): By assumption $\Gamma \vdash \textbf{case}\ v\ \{p \Rightarrow e_p; \overline{p' \Rightarrow e'_{p'}}\} : T'$ and $\Gamma \vdash \sigma$ 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 $\Gamma' \vdash e_p[\overline{x} \mapsto \overline{y}] : T'$, which concludes the proof. □

Proof of Theorem 3.1.2 on Subject Reduction for Configs: If $\Delta, \mathcal{G} \vdash_R N$ and $N \to N'$ then $\exists\ \Delta',\ \mathcal{G}'$ such that $\Delta \subseteq \Delta'$, $\mathcal{G} \subseteq \mathcal{G}'$ and $\Delta', \mathcal{G}' \vdash_R N'$.

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

- Case (Skip): By assumption $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{skip}; s\}, Q)$; 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 3.1.1, 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 $ob(\mathsf{o}, \sigma, \{\sigma' | x = v; s\}, Q) \to ob(\mathsf{o}, \sigma, \{\sigma'[x \mapsto v] | s\}, Q)$. By applying typing 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 $ob(\mathsf{o}, \sigma, \{\sigma' | x = v; s\}, Q) \to ob(\mathsf{o}, \sigma[x \mapsto v], \{\sigma' | s\}, Q)$, then we derive $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma[x \mapsto v], \{\sigma' | s\}, Q)$.

- Case *Conditionals*: By assumption

$$\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2; s\}, Q)$$

and $[\![e]\!]_{(\sigma \circ \sigma')} = \texttt{true}$. There exists 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 $\mathcal{G}_1 = \mathcal{G}_1' \uplus \mathcal{G}_1''$ and $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2 \uplus \mathcal{G}_3$. 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 rule (T-Object) we conclude that $\Delta, \mathcal{G} \setminus \mathcal{G}_1'' \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | s_1; s\}, Q)$. 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

$$\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{ while } e \{\ s\ \}; s'\}, Q)$$

and $[\![e]\!]_{(\sigma \circ \sigma')} = \texttt{true}$. There exists 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 applying (T-Comp) and (T-While) we have $\Delta', \emptyset \vdash_R \textbf{while } e \{\ s\ \}$, and $\Delta', \mathcal{G}_1 \vdash_R s'$. 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 $\Delta', \emptyset \vdash_R s$ and $\Delta', \emptyset \vdash_R \textbf{while } e \{\ s\ \}$ then by applying (T-Comp) we obtain $\Delta', \emptyset \vdash_R s\ ; \textbf{while } e \{\ s\ \}$. By rule (T-Comp) we have that $\Delta', \mathcal{G}_1 \vdash_R s\ ; \textbf{while } e \{\ s\ \}; s'$. We conclude by (T-Object). The case for $[\![e]\!]_{(\sigma \circ \sigma')} = \texttt{false}$ and rule (While-False) is similar.

- Case *Awaits*: By assumption

$$\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{ await } g; s\}, Q) \quad N$$

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

$$\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\sigma' | \textbf{ return } e; s\}, Q) \quad fut(\mathtt{f}, \bot)$$

and $\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, \bot)$ and by reduction rule $\sigma'(\textbf{destiny}) = \mathtt{f}$ and $v = [\![e]\!]_{(\sigma \circ \sigma')}$ and $\mathit{ob}(\mathtt{o}, \sigma, \{\sigma' | \textbf{ return } e; s\}, Q)\ \mathit{fut}(\mathtt{f}, \bot) \rightarrow \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' | s\}, Q)\ \mathit{fut}(\mathtt{f}, v)$. Trivially, $\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' | s\}, Q)$. By the premises of (T-Return) we have $\Delta \vdash_R e : T$ and $\Delta(\textbf{destiny}) = \mathtt{Fut}\langle T \rangle$. By assumption $\sigma'(\textbf{destiny}) = \mathtt{f}$, hence $\Delta(\mathtt{f}) = \mathtt{Fut}\langle T \rangle$. By assumption $v = [\![e]\!]_{(\sigma \circ \sigma')}$, then by applying Lemma 3.1.1 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

$$\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' \mid x = \textbf{get}(e); s\}, Q) \quad \mathit{fut}(\mathtt{f}, v)$$

where $\Delta, \emptyset \vdash_R \mathit{fut}(\mathtt{f}, v)$, $v \neq \bot$ and $\mathtt{f} = [\![e]\!]_{(\sigma \circ \sigma')}$. By reduction rule $\mathit{ob}(\mathtt{o}, \sigma, \{\sigma' \mid x = \textbf{get}(e); s\}, Q)\ \mathit{fut}(\mathtt{f}, v) \rightarrow \mathit{ob}(\mathtt{o}, \sigma, \{\sigma' | x = v; s\}, Q)\ \mathit{fut}(\mathtt{f}, v)$. 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 $\Delta, \emptyset \vdash_R \textbf{get}(e) : T$. 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

$$\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, p, Q) \quad \mathit{invoc}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})$$

and by reduction 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 (T-Config) we have $\Delta, \mathcal{G}_1 \vdash_R \mathit{ob}(\mathtt{o}, \sigma, p, Q)$ and $\Delta, \mathcal{G}_2 \vdash_R \mathit{invoc}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v})$ such that $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By assumption $p' = \mathrm{bind}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}, \mathit{class}(\mathtt{o}))$ and let $\mathit{class}(\mathtt{o}) = \mathtt{C}$. By (T-Invoc) we have $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = (\mathcal{G}_m, \mathbb{r})(\overline{T\ x}) \rightarrow T$, $\Delta(\mathtt{o}) = (\mathtt{C}, \sigma(\mathbb{r}))$, $\Delta(\mathtt{f}) = \mathtt{Fut}\langle \sigma(T) \rangle$, and $\Delta \vdash_R \overline{v} : \overline{\sigma(T)}$ and $\mathcal{G}_2 = \sigma(\mathcal{G}_m)$. The bind function returns a process $p' = \{\overline{T\ x = v}; \overline{T'\ x' = \textbf{null}}; \textbf{this} = \mathtt{o} \mid s\}$ 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

$$\Delta, \mathcal{G} \vdash_R \mathit{ob}(\mathtt{o}, \sigma, \{\ \sigma'' \mid x = \textbf{new}\ \ \mathtt{C}\ (\overline{e}); s\ \}, Q)$$

and $\mathit{ob}(\mathtt{o}, \sigma, \{\ \sigma''\ \mid\ x\ =\ \textbf{new}\ \ \mathtt{C}\ \ (\overline{e}); s\ \}, Q) \rightarrow \mathit{ob}(\mathtt{o}, \sigma, \{\sigma'' | x = \mathtt{o}'; s\}, Q)\ \mathit{ob}(\mathtt{o}', \sigma', \textbf{idle}, \varepsilon)$. 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 $\Delta', \mathcal{G}_1 \vdash_R x = \textbf{new}\ \ \mathtt{C}\ (\overline{e}); s$ and $\mathcal{G}_2$ is the set of cogs in Q where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By (T-Comp) $\Delta', \emptyset \vdash_R x = \textbf{new}\ \ \mathtt{C}\ (\overline{e})$

and $\Delta', \mathcal{G}_1 \vdash_R s$. By rule (T-Assign) we have that $\Delta'(x) = T$ and $\Delta', \emptyset \vdash_R \textbf{new } \mathtt{C}\ (\overline{e}) : T$. By the premises of the typing rule we have that $fields(\mathtt{C}) = \overline{T\ f}$, $\Delta' \vdash \overline{x : T'}$, $\text{tmatch}(\overline{T}, \overline{T'}) = \pi$ and $T \in \text{crec}(\mathtt{G}, \mathtt{C}, \pi)$, 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 $\text{implements}(\mathtt{C}, \mathtt{I})$. Let $\mathbb{r} = \mathtt{G}[\pi \uplus \rho(\overline{f : (\mathtt{I}, \mathbb{r})})]$. Since $fresh(\mathtt{o}')$, then let $\Delta'' = \Delta, \mathtt{o}' : (\mathtt{C}, \mathbb{r})$. Then, $\Delta'', \mathcal{G}_1 \uplus \mathcal{G}_2 \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'' | x = \mathtt{o}'; s\}, Q)$, by (T-Process), (T-Comp), and (T-Object). By assumption, function $\text{atts}(\mathtt{C}, \overline{v}, \mathtt{o}', \mathtt{c})$ returns a substitution σ' that is well typed in Δ''. So, $\Delta'', \emptyset \vdash_R ob(\mathtt{o}', \sigma', \textbf{idle}, \varepsilon)$. Then, by (T-Config) we have $\Delta'', \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'' | x = \mathtt{o}'; s\}, Q)\ \ ob(\mathtt{o}', \sigma', \textbf{idle}, \varepsilon)$.

- Case (New-Cog-Object): By assumption

$$\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'' | x = \textbf{new cog } \mathtt{C}\ (\overline{e}); s\}, Q)$$

and by reduction we have $ob(\mathtt{o}, \sigma, \{\sigma'' | x = \textbf{new cog } \mathtt{C}\ (\overline{e}); s\}, Q) \rightarrow ob(\mathtt{o}, \sigma, \{\sigma'' | x = \mathtt{o}'; s\}, Q)\ \ ob(\mathtt{o}', \sigma', \textbf{idle}, \varepsilon)\ \ cog(\mathtt{c}', \mathtt{o}')$. By assumption $\overline{v} = [\![\overline{e}]\!]_{(\sigma \circ \sigma'')}$, $\text{fresh}(\mathtt{o}')$, $\sigma' = \text{atts}(\mathtt{C}, \overline{v}, \mathtt{o}', \mathtt{c})$. 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 the typing rules (T-Object) and (T-Process) we have that $\Delta', \mathcal{G}_1 \vdash_R x = \textbf{new cog } \mathtt{C}\ (\overline{e}); s$ $\Delta, \mathcal{G}_2 \vdash_R Q$, where $\mathcal{G} = \mathcal{G}_1 \uplus \mathcal{G}_2$. By rules (T-Comp) and (T-Assign) $\Delta', \{\mathtt{G}\} \vdash_R x = \textbf{new cog } \mathtt{C}\ (\overline{e})$ and $\Delta', \mathcal{G}_1 \setminus \{\mathtt{G}\} \vdash_R s$. By rule (T-Assign), $\Delta'(x) = T$ and $\Delta', \{\mathtt{G}\} \vdash_R \textbf{new cog } \mathtt{C}\ (\overline{e}) : T$. By the premise of typing rule (T-NewCog), we have that $fields(\mathtt{C}) = \overline{T\ f}$, $\Delta' \vdash \overline{x : T'}$, $\text{tmatch}(\overline{T}, \overline{T'}) = \pi$ and $T \in \text{crec}(\mathtt{G}, \mathtt{C}, \pi)$. Then, by definition of the auxiliary function crec, it means that $T = (\mathtt{I}, \mathtt{G}[\overline{f : \pi \circ \rho(\mathtt{I}, \mathbb{r})}])$ and $\text{implements}(\mathtt{C}, \mathtt{I})$. Let $\mathbb{r}$ be such that $\mathbb{r} = \mathtt{G}[\pi \circ \rho(\overline{f : (\mathtt{I}, \mathbb{r})})]$. Since $fresh(\mathtt{o}')$ and $fresh(\mathtt{c}')$, we have $\Delta'' = \Delta, \mathtt{o}' : (\mathtt{C}, \mathbb{r}), \mathtt{c}' : \mathtt{G}$. By applying typing rules (T-Process), (T-Comp), and (T-Object) $\Delta'', \mathcal{G}_1 \setminus \{\mathtt{G}\} \uplus \mathcal{G}_2 \vdash_R ob(\mathtt{o}, \sigma, \{\sigma'' | x = \mathtt{o}'; s\}, Q)$. By (T-Cog) we have $\Delta'', \{\mathtt{G}\} \vdash_R cog(\mathtt{c}', \mathtt{o}')$. By assumption, function $\text{atts}(\mathtt{C}, \overline{v}, \mathtt{o}', \mathtt{c}')$ returns a substitution σ' that is well typed in Δ''. So, $\Delta'', \emptyset \vdash_R ob(\mathtt{o}', \sigma', \textbf{idle}, \varepsilon)$. By (T-Config) we obtain the result.

- Case (Self-Sync-Call): By assumption

$$\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma' \mid x = e.\mathtt{m}(\overline{e}); s\}, Q)$$

and by reduction rule $ob(\mathtt{o}, \sigma, \{\sigma' \mid x = e.\mathtt{m}(\overline{e})\ ;\ s\}, Q) \rightarrow ob(\mathtt{o}, \sigma, \{\sigma'' \mid s'; \textbf{cont}(\mathtt{f}')\}, Q \cup \{\sigma' \mid x = \textbf{get}(\mathtt{f}); s\})\ \ fut(\mathtt{f}, \bot)$. Then, it is the case that $\mathtt{o} = [\![e]\!]_{(\sigma \circ \sigma')}$, $\overline{v} = [\![\overline{e}]\!]_{(\sigma \circ \sigma')}$, $\sigma'(\textbf{destiny}) = \mathtt{f}'$, $\text{fresh}(\mathtt{f})$, and also $\{\sigma'' | s'\} = \text{bind}(\mathtt{o}, \mathtt{f}, \mathtt{m}, \overline{v}, class(\mathtt{o}))$ and let $class(\mathtt{o}) = \mathtt{C}$. Since, by assumption class $\mathtt{C}$ 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. 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 typing rules (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 using (T-Comp), we have 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 typing rules (T-Assign) and (S-Call) we have $\Delta', \mathcal{G}'_1 \vdash_R e.\mathtt{m}(\overline{e}) : \rho(T)$ for some substitution ρ. By the premises of (T-SCall) we have $mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}_m, \mathbb{r})(\overline{T\ x}) \to T$, $\Delta' \vdash e : (\mathtt{I}, \rho(\mathbb{r}))$, $\Delta' \vdash \overline{e} : \overline{\rho(T)}$, $\mathrm{coloc}(\rho(\mathbb{r}), \Delta'(\mathbf{this}))$ and $\mathcal{G}'_1 = \rho(\mathcal{G}_m)$. Since $\mathtt{o} = [\![e]\!]_{(\sigma \circ \sigma')}$ then, $\Delta'(\mathtt{o}') : (\mathtt{C}, \rho(\mathbb{r}))$, such that $\mathrm{implements}(\mathtt{C}, \mathtt{I})$ and $mtype(\mathtt{m}, \mathtt{C}) = mtype(\mathtt{m}, \mathtt{I})$. Let $\sigma'' = \overline{T''\ x''\ w''}; \theta''$, then by (T-Method) we have $\Delta', \overline{x'' : T''}, \mathcal{G}'_1 \vdash s'$, hence $\Delta', \mathcal{G}'_1 \vdash_R \{\sigma''|s'\}$. Since $\sigma'(\mathbf{destiny}) = \mathtt{f}'$, then $\Delta', \mathcal{G}'_1 \vdash_R \{\sigma''|s'; \mathbf{cont}(\mathtt{f}')\}$. Since $fresh(\mathtt{f})$, let $\Delta'' = \Delta, \mathtt{f} : \rho(T)$, then $\Delta'', \emptyset \vdash_R x = \mathbf{get}(\mathtt{f})$. By rule (T-Comp) and Lemma 3.2.1 we have $\Delta'', \mathcal{G}''_1 \vdash_R x = \mathbf{get}(\mathtt{f}); s$. Then, $\Delta'', \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\sigma''|s'; \mathbf{cont}(\mathtt{f}')\}, Q \cup \{\sigma'|\ x = \mathbf{get}(\mathtt{f}); s\})$. 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 reduction rule $ob(\mathtt{o}, \sigma, \{\sigma''|\mathbf{cont}(\mathtt{f})\}, Q \cup \{\sigma'|s\}) \to ob(\mathtt{o}, \sigma, \{\sigma'|s\}, Q)$, 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 also $ob(\mathtt{o}, \sigma, \{\sigma' \mid x = e!\mathtt{m}(\overline{e}); s\}, Q) \to ob(\mathtt{o}, \sigma, \{\sigma'|x = \mathtt{f}; s\}, Q)\ invoc(\mathtt{o}', \mathtt{f}, \mathtt{m}, \overline{v})\ fut(\mathtt{f}, \bot)$. 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 $\Delta', \mathcal{G}'_1 \vdash_R x = e!\mathtt{m}(\overline{e})$ and $\Delta', \mathcal{G}''_1 \vdash_R s$ where $\mathcal{G}_1 = \mathcal{G}'_1 \uplus \mathcal{G}''_1$. For the first judgement, by (T-Assign) and (T-ACall), we have that $\Delta'(x) = \mathtt{Fut}\langle\rho(T)\rangle$ and $\Delta', \mathcal{G}'_1 \vdash_R e!\mathtt{m}(\overline{e}) : \mathtt{Fut}\langle\rho(T)\rangle$, for some substitution ρ. By the premise of (T-ACall) we have $mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}_m, \mathbb{r})(\overline{T\ x}) \to T$, $\Delta' \vdash_R e : (\mathtt{I}, \rho(\mathbb{r}))$ and $\Delta' \vdash_R \overline{e} : \overline{\rho(T)}$ and $\mathcal{G}'_1 = \rho(\mathcal{G}_m)$.

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 3.1.1 it means $\Delta' \vdash_R \mathtt{o}' : (\mathtt{C}, \rho(\mathbb{r}))$ for a class $\mathtt{C}$ such that implements$(\mathtt{C}, \mathtt{I})$ such that $mtype(\mathtt{m}, \mathtt{C}) = mtype(\mathtt{m}, \mathtt{I})$. Also, by Lemma 3.1.1 $\Delta' \vdash_R \overline{v} : \overline{\rho(T)}$. 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 $\Delta'', \mathcal{G} \setminus \mathcal{G}'_1 \vdash_R ob(\mathtt{o}, \sigma, \{\sigma' | x = \mathtt{f}; s\}, Q)$. By applying (T-Invoc) we have $\Delta'', \mathcal{G}'_1 \vdash_R invoc(\mathtt{o}', \mathtt{f}, \mathtt{m}, \overline{v})$. By applying (T-Future2) we have $\Delta'', \emptyset \vdash_R fut(\mathtt{f}, \bot)$. Then, we conclude by applying (T-Config).

- Case (Rebind-Local): By assumption

$$\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma, \{\ \sigma' \mid \textbf{rebind}\ e.f = e'; s\ \}, Q)$$

and $ob(\mathtt{o}, \sigma, \{\ \sigma' \mid \textbf{rebind}\ e.f = e'; s\ \}, Q) \rightarrow ob(\mathtt{o}, \sigma[f \mapsto v], \{\ \sigma' \mid s\ \}, Q)$ 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, $\Delta', \mathcal{G}_1 \vdash_R \textbf{rebind}\ e.f = e'; s$ 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$(\mathbb{r}, \Delta'(\textbf{this}))$ – meaning, belonging to the same cog. Since $v = [\![e']\!]_{(\sigma\circ\sigma')}$, then by Lemma 3.1.1 $\Delta' \vdash_R v : T$. Then, by (T-Object) $\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma[f \mapsto v], \{\ \sigma' \mid s\ \}, Q)$.

- Case (Rebind-Global): By assumption

$$\Delta, \mathcal{G} \vdash_R ob(\mathtt{o}, \sigma_\mathtt{o}, K_{\textbf{idle}}, Q) \quad ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid \textbf{rebind}\ e.f = e'; s\ \}, Q')$$

and $ob(\mathtt{o}, \sigma_\mathtt{o}, K_{\textbf{idle}}, Q)\ \ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid \textbf{rebind}\ e.f = e'; s\ \}, Q') \rightarrow ob(\mathtt{o}, \sigma_\mathtt{o}[f \mapsto v], K_{\textbf{idle}}, Q)\ \ ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid s\ \}, Q')$. By typing rule (T-Config) it means that $\Delta, \mathcal{G}_1 \vdash_R ob(\mathtt{o}, \sigma_o, K_{\textbf{idle}}, Q)$ and also $\Delta, \mathcal{G}_2 \vdash_R ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid \textbf{rebind}\ e.f = e'; s\ \}, Q')$ 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, $\Delta', \mathcal{G}'_2 \vdash_R \textbf{rebind}\ e.f = e'; s$ 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$(\mathbb{r}, \Delta'(\textbf{this}))$ – meaning, 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 3.1.1 we have that $\Delta' \vdash_R v : T$. Then, trivially $\Delta, \mathcal{G}_1 \vdash_R ob(\mathtt{o}, \sigma_\mathtt{o}[f \mapsto v], K_{\textbf{idle}}, Q)$ and $\Delta, \mathcal{G}_2 \vdash_R ob(\mathtt{o}', \sigma_{\mathtt{o}'}, \{\ \sigma'_{\mathtt{o}'} \mid s\ \}, Q')$. We conclude by (T-Config). □

Proof of Theorem 3.1.3 on Correctness of Rebindings: If $\Delta, \mathcal{G} \vdash_R N$, then for all objects $ob(\mathtt{o}, \sigma, \{\ \sigma' \mid s\ \}, Q) \in N$ and $s = (\textbf{rebind}\ e.f = e'; s')$ there exists an object $ob(\mathtt{o}', \sigma'', K_{\textbf{idle}}, Q')$ such that $[\![e]\!]_{(\sigma\circ\sigma')} = \mathtt{o}'$ 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 we have $\Delta, \mathcal{G} \vdash_R ob(\mathsf{o}, \sigma, \{\ \sigma' \mid \textbf{rebind}\ e.f = e'; s'\ \}, Q)$. 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 by (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. By Lemma 3.1.1 this means that $\Delta', \emptyset \vdash_R v : (\mathtt{I}, \mathbb{r})$. This implies that v is an object identifier o'. Then, by the reduction rules (New-Object) or (New-Cog-Object), it means that the object was already created, and moreover it is well typed. Let o' have $ob(\mathsf{o}', \sigma'', K_{\textbf{idle}}, Q')$ as its configuration. 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, 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) and (Context).

□

Proof of Theorem 3.1.4 on Correctness of 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 $\sigma(\textbf{this}) = \mathsf{o}$ and let $\sigma(\textbf{cog}) = \mathsf{c}$. 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 the typing 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 $T = \rho(T')$ for some substitution ρ of the formal return type T' to the actual return type and T. Moreover, $mtype(\mathtt{m}, \mathtt{I}) = (\mathcal{G}_m, \mathbb{r})(\overline{T\ x}) \to T'$, and $\mathcal{G}'_1 = \rho(\mathcal{G}_m)$. Since the synchronous method call is well typed, by the premise of (T-SCall) and (T-Weak1) we have that $\Delta' \vdash_R e : (\mathtt{I}, \rho(\mathbb{r}))$ and let $[\![e]\!]_{(\sigma \circ \sigma')} = v$. By Lemma 3.1.1 this means that $\Delta', \emptyset \vdash_R v : (\mathtt{I}, \rho(\mathbb{r}))$. By following the same lines as in the previous theorem, it is the case that v is an object identifier $\mathtt{o}'$. Then, by the reduction semantics rules (New-Object) or (New-Cog-Object), it means that the object was already created, and in addition it is well typed. Let $\mathtt{o}'$ have $ob(\mathtt{o}', \sigma'', K_{\mathbf{idle}}, Q')$ as its configuration. The rest of the proof follows exactly the same line as the correctness of rebinding proof where again by the premise of (T-SCall) we have that the predicate $\mathrm{coloc}(\rho(\mathbb{r}), \Delta'(\mathbf{this}))$ is true.

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

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 [77], the latter being an extension of the Abstract Behavioural Specification (ABS) language [63]. 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 [77] 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 thesis 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, 24, 54, 64, 114]. As stated in [63], the concurrency model of ABS is a generalisation of the

concurrency model of Creol [65] 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, 24, 64] 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 [63] cogs in ABS can be compared to *monitors* in [55]. However, differently from monitors, there is no explicit signalling. It is possible to encode monitors in the language, as stated in [64].

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, 79, 82, 87] 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 [75] 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 [79] and COMP [78] 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 [77], has a unified description of objects and components by exploiting the similarities between them. This brings in several benefits wrt 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 [61], 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 [80]. 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 [22] which is designed for a process calculus with *ambients* [23], 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 [22] 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* [25], 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. [56].

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 [86, 91] to check (dynamic) binding of input or output ports and guarantee consistencies of operations, or in [76] 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 [50, 51]. 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 [51], 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 [50].

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 in [57,104] and later in [58] 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 [49, 108, 111], component-based object systems [107], object-oriented languages [18,40–42,47], Web Services and Contracts, W3C-CDL a language for choreography [21,88] etc.

Since their appearance, many extensions have been made to session types. An important research direction is the one that brings from *dyadic* or *binary* sessions types [38, 57, 58, 104, 109, 118], describing communication between only two participants, to *multiparty* session types [10, 59], 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 [37] or to choreographies [21, 88]. In dyadic session types, different typing features have been added. Subtyping relation for (recursive) session types is added in [48]. Bounded polymorphism is added in [45] as a further extension to subtyping, and parametric polymorphism is added in [15]. The authors in [89] add higher-order primitives in order to allow not only the mobility of channels but also the mobility of processes.

Session types describe a protocol as a type abstraction, guaranteeing privacy, communication safety and session fidelity. Privacy requires that the session channel is 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 data 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. 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.

A fundamental notion of session types is that of *duality*. 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 interacting participants. 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, respectively 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 the dual type. As shown in the next chapter, term constructs like $(\boldsymbol{\nu} xy)$, are added to the syntax of processes and are used to create the endpoints of a session channel, on which the duality relation is then checked.

Another important feature related to session types is that of session channel transmission, namely *delegation*, where a session endpoint is send to a participant, for the latter to carry out the session.

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 [48, 58, 104, 118] (this often introduces a duplication in the typing contexts, as well).

In this part of the thesis 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 endpoint describes a structured sequence of inputs and outputs by specifying the type of messages that it can transmit. This

recalls the *linearised* channels [72], which are channels used multiple times for communication but *only* in a sequential manner. Linearised types can be encoded, as shown in [72], into linear types –i.e., channel types used *exactly once* and recursive types. Differently from session types, linearised channel types have the same carried type (or payload) 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 [99, 101].

Other analogies between session types and π-types concern session creation and duality. Session creation is modelled via the *restriction* construct, used to create and bind a new private session channel. 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 by owning the opposite capabilities.

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 types. There is a plethora of papers on session types in which session types are always taken as primitives. However, by following Kobayashi [71], we define an encoding of session types into standard π-types and by exploiting this encoding, session types and 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 payload 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 thesis, 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 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.

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

In this chapter we present the standard typed polyadic π-calculus [83–85, 101]. We start by giving the syntax of terms and the operational semantics, then we introduce the syntax of types and the typing rules.

4.1 Syntax

$P, Q ::=$	$x!\langle\tilde{v}\rangle.P$	(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 syntax of the 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. 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 processes in the remainder of the thesis. We use $\mathsf{fv}(P)$ to denote the set of free variables in P, $\mathsf{bv}(P)$ to denote the bound ones and $\mathsf{vars}(P) = \mathsf{fv}(P) \cup \mathsf{bv}(P)$ 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 $P[x/y]$ to denote process P where every occurrence of the free variable y is substituted by variable x. 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 and Substitution)**.** 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
   - **case** v **of** $\{l_i\_y_i \triangleright Q_i\}_{i\in I}$ by **case** v **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.

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

Figure 4.2: Structural congruence for the standard π-calculus

$$
\begin{array}{l}
P = P \\
P = Q \text{ implies } Q = P \\
P = Q \text{ and } Q = R \text{ implies } P = R \\
P = Q \text{ implies } C[P] = C[Q]
\end{array}
$$

Figure 4.3: Rules for equational reasoning

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.

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 $x \notin \mathsf{fv}(Q)$. 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.

Since $\equiv$ is a congruence, it means that it is closed under every *context* C, where informally a context is a process with a hole. Hence, in addition to the axioms presented in Fig. 4.2, we also need the rules in Fig. 4.3 for equational reasoning, where we read $=$ as $\equiv$.

The operational semantics of the standard π-calculus is given in Fig 4.4. 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 $(x!\langle\tilde{v}\rangle.P \mid x?(\tilde{y}).Q)$. Rule (R$\pi$-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

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

Figure 4.4: Semantics of the standard π-calculus

case normalisation 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

$$
\begin{array}{rll}
\tau ::= & \ell_{\mathsf{i}}\,[\widetilde{T}] & \text{(linear input)} \\
 & \ell_{\mathsf{o}}\,[\widetilde{T}] & \text{(linear output)} \\
 & \ell_{\sharp}\,[\widetilde{T}] & \text{(linear connection)} \\
 & \emptyset[\widetilde{T}] & \text{(no capability)} \\
 & \cdots & \text{(other channel types)} \\
T ::= & \tau & \text{(channel type)} \\
 & \langle l_i\_T_i\rangle_{i\in I} & \text{(variant type)} \\
 & \texttt{Bool} & \text{(boolean type)} \\
 & \cdots & \text{(other type constructs)}
\end{array}
$$

Figure 4.5: Syntax of linear π-types

The syntax of linear π-types is given in Fig. 4.5. Let α, β range over actions or capabilities, being 'i' input, 'o' output or '$\sharp$' connection. Let τ range over channel types and T range over types. Linear types are $\ell_{\mathsf{i}}\ [\widetilde{T}]$, $\ell_{\mathsf{o}}\ [\widetilde{T}]$ and $\ell_{\sharp}\ [\widetilde{T}]$. These types specify both the capability and the multiplicity of a channel i.e., how the channel should be used and for how many times. In particular, 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 types $\widetilde{T}$; 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 types $\widetilde{T}$, and 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 types $\widetilde{T}$. The capability $\sharp$ denotes the combination of i and o capabilities. In addition, we denote with $\emptyset[\widetilde{T}]$ the type of a channel with no capabilities, namely a channel that cannot be used for communication at all. 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 linear 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 linearity of x and y, if $x \notin \mathsf{fv}(P)$ and $y \notin \mathsf{fv}(Q)$. 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 linearity of x since it is used twice to send a value v and value w.

4.4 π-Typing Rules

A typing context is a partial function from variables to types and is defined as follows:

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

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

$\mathsf{lin}(T)$	if $T = \ell_\alpha\ [\widetilde{T}]$ or $(T = \langle l_i\_T_i\rangle_{i\in I}$ and for some $j \in I$. $\mathsf{lin}(T_j))$
$\mathsf{un}(T)$	otherwise

Combination of π-types

$$\begin{array}{rll} \ell_{\mathsf{i}}\,[\widetilde{T}] \uplus \ell_{\mathsf{o}}\,[\widetilde{T}] & \triangleq \ell_{\sharp}\,[\widetilde{T}] & \\ T \uplus T & \triangleq T & \text{if } \mathsf{un}(T) \\ T \uplus S & \triangleq \mathtt{undef} & \text{otherwise} \end{array}$$

Combination of typing contexts

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

Figure 4.6: Combination of π-types and typing contexts

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 typing contexts in the expected way:

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

We define the *combination* of types and of typing contexts in Fig. 4.6. We use $\uplus$ to denote the operator of combination. This operator is associative and hence we do not use brackets. The combination of linear types states that a linear input type combined with a linear output type results in a linear connection 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 $\emptyset[\widetilde{T}] \uplus \emptyset[\widetilde{T}] \triangleq \emptyset[\widetilde{T}]$. 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 the combination is undefined. The combination of typing contexts $\Gamma_1 \uplus \Gamma_2$ is extended to a tuple of typing contexts $\Gamma_1 \uplus \cdots \uplus \Gamma_n$ and we denote this for simplicity as $\widetilde{\Gamma}$.

We define the duality of π-types to be simply the duality on the capability of the channel. Formally, it is defined in Fig 4.7.

Typing judgements are of the following two forms: $\Gamma \vdash v : T$ stating that 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.

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

Figure 4.7: Type duality for linear π-types

The typing rules for the linear π-calculus are given in Fig 4.8. Rule (Tπ-VAR) states that a variable is of type the one assumed in the typing context. Moreover, the typing context contains only unrestricted type assumptions. Rule (Tπ-VAL) states that a boolean value, either `true` or `false`, is of type `Bool`. Again, the typing context contains only unrestricted type assumptions. Rule (Tπ-INACT) states that the terminated process **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, 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 $x : \ell_\sharp\,[\widetilde{T}]$. Since the type assumption on variable x is needed to type P and it is a linear channel type, it means that x is free 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 for communication in P. This rule is needed in the standard π-calculus to prove subject reduction stated by Theorem (see [101]). Moreover, this rule is also needed in our encoding that we present in Chapter 6. Rule (Tπ-IF) is standard, except for the combination of 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 with 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 **case** v **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 to type its branches. Again, this does not violate linearity, since only one of the branches is going to be executed.

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

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

$$\frac{\Gamma, x : \ell_\sharp\, [\widetilde{T}] \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}\ (\mathrm{T}\pi\text{-}\textsc{Res}1) \qquad \frac{\Gamma_1 \vdash v : \mathtt{Bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \uplus \Gamma_2 \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}\ (\mathrm{T}\pi\text{-}\textsc{If})$$

$$\frac{\Gamma, x : \emptyset[] \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)P}\ (\mathrm{T}\pi\text{-}\textsc{Res}2) \qquad \frac{\Gamma_1 \vdash x : \ell_{\mathsf{i}}\, [\widetilde{T}] \qquad \Gamma_2, \tilde{y} : \widetilde{T} \vdash P}{\Gamma_1 \uplus \Gamma_2 \vdash x?(\tilde{y}).P}\ (\mathrm{T}\pi\text{-}\textsc{Inp})$$

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

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

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

Figure 4.8: Typing rules for the standard π-calculus

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 for all $x \in \mathsf{dom}(\Gamma)$, then $\Gamma(x) \neq \ell_\sharp\,[\widetilde{T}]$.

In the following we give the substitution lemma for the linear π-calculus and the unrestricted weakening and strengthening lemmas.

Lemma 4.5.2 (Substitution Lemma for Linear π-Calculus)**.** Let $\Gamma, x : T \vdash P$, and let $\Gamma \uplus \Gamma'$ be defined and $\Gamma' \vdash v : T$. Then, $\Gamma \uplus \Gamma' \vdash P[v/x]$.

Lemma 4.5.3 (Unrestricted Weakening in Linear π- Calculus)**.** If $\Gamma \vdash P$, then $\Gamma, x : T \vdash P$, for all $x \notin \mathsf{fv}(P)$ and $\mathsf{un}(T)$.

Lemma 4.5.4 (Strengthening in Linear π-Calculus)**.** If $\Gamma, x : T \vdash P$ and $x \notin \mathsf{fv}(P)$ and $\mathsf{un}(T)$, then $\Gamma \vdash P$.

The following lemma states the type preservation of a linear process under structural congruence.

Lemma 4.5.5 (Type Preservation under $\equiv$ for Linear π-Calculus)**.** Let $\Gamma \vdash P$ and $P \equiv P'$, then $\Gamma \vdash P'$.

As often in the literature, in order to prove type soundness, we show first the subject reduction (or type preservation under reduction) and the type safety as stated in [97]. We start with subject reduction.

Theorem 4.5.6 (Subject Reduction for Linear π- Calculus)**.** Let Γ be a closed linear typing context. If $\Gamma \vdash P$ and $P \to P'$, then $\Gamma \vdash P'$.

By the statement of subject reduction for linear π- calculus, since the typing context is closed (it has no linear channel owning both capabilities), this means that, P reduces to P' either by a **case** normalisation or by a conditional reduction or if case a communication occurs, then it is a communication on a restricted channel which owns both capabilities of input and output. Reduction rules under a context are a generalisation of the above.

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 and using it with the same capability, like $(\boldsymbol{\nu} x)(x?(z) \mid x?(z))$.

Definition 4.5.7 (Well-Formedness for Linear π- Calculus)**.** 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 4.5.8 (Type Safety for Linear π-Calculus)**.** 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 4.5.9 (Type Soundness for Linear π- Calculus)**.** If $\vdash P$ and $P \rightarrow^* Q$, then Q is well formed.

Notice that this is one way of presenting the type soundness in the standard typed π-calculus. Another way would be by introducing the notion of `wrong` and extending the operational semantics with reductions to `wrong` and stating the type soundness as "well-typed programs do not go `wrong`". This is shown in [101].

Chapter 5

Background on Session Types

We start this chapter with an example, the "Distributed Auction System" taken from [110].

Example 5.0.10. *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 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:

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

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.

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

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:

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

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.

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

So, summing it up we have the following situation:

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

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.

This example involves three participants and uses *multiparty* session types in order to illustrate their expressiveness; however, in our formal development we focus on *dyadic* session types.

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

Figure 5.1: Syntax of the π-calculus with sessions

5.1 Syntax

The syntax of the π-calculus with sessions is given in Fig. 5.1. 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. Branching and selection indicate the external and the eternal choice, respectively. The order of labelled processes is not important and the labels are all different. Process **if** v **then** P **else** Q is the standard conditional process. Process $P \mid Q$ is the parallel composition of processes P, Q. Process $\mathbf{0}$ is the terminated process. Process $(\nu xy)P$ restrict variables x, y with scope P. This restriction is different from the standard one in the π-calculus. It 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*. 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. A variable can be *bound* or *free*, the latter holds when the variable does not occur under a restriction or as the

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

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

object of an input process. We denote with $\mathsf{bv}(P)$ the set of bound variables of process P and with $\mathsf{fv}(P)$ we denote the set of free variables of process P. Hence, we use $\mathsf{vars}(P) = \mathsf{bv}(P) \cup \mathsf{fv}(P)$ to denote the set of variables in P. We will use alpha-conversion and substitution 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 alpha-conversion to avoid the unintended capture of variables by the binders of the calculus.

Definition 5.1.1 (Alpha-convertibility and Substitution)**.** 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 of a subterm $(\boldsymbol{\nu} xy)Q$ by $(\boldsymbol{\nu} wy)Q[w/x]$ or by $(\boldsymbol{\nu} xw)Q[w/y]$ such that 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 session processes that satisfies the axioms in Fig. 5.2. The first three axioms state that the parallel composition of processes is commutative, associative and uses process $\mathbf{0}$ as the neutral

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

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

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

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

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

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

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

Figure 5.3: Semantics of the π-calculus with sessions

element. The last three axioms state 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 $x, y \notin \mathsf{fv}(Q)$ 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. As for the standard π-calculus, we need the rules for equational reasoning. They are the same as the ones given in Section 4.2.

The semantics of the π-calculus with sessions is given in terms of the reduction relation $\to$, which is a binary relation over processes, and it is defined by the rules in Fig. 5.3. We denote with $\to^*$ the reflexive and transitive closure of $\to$. 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 wrt 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 (νxy). As a consequence, communication occurs only on bound variables. After the communication the restriction still persists in order to enable further possible communications. 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 restriction (νxy). The selecting process continues as P and the branching process continues

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

Figure 5.4: Syntax of session types

as P_j where j is the selected label. Again, notice that communication occurs 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. It states that reduction is closed under structural congruence.

5.3 Session Types

The syntax of session types is given in Fig. 5.4. Let q range over type qualifiers, p over pretypes, qp 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 qp, the qualified pretype. A pretype can be $!T.U$ or $?T.U$, which respectively, is the type 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 thesis, 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 assume that the qualifier lin is used for every pretype unless it is stated otherwise.

$$\begin{array}{rcl}
\overline{\mathsf{end}} & \triangleq & \mathsf{end} \\
\overline{q!T.U} & \triangleq & q?T.\overline{U} \\
\overline{q?T.U} & \triangleq & q!T.\overline{U} \\
\overline{q \oplus \{l_i : T_i\}_{i \in I}} & \triangleq & q\&\{l_i : \overline{T}_i\}_{i \in I} \\
\overline{q\&\{l_i : T_i\}_{i \in I}} & \triangleq & q \oplus \{l_i : \overline{T}_i\}_{i \in I}
\end{array}$$

Figure 5.5: Type duality for session types

The following predicates state when a type is linear or unrestricted.

$$\begin{array}{ll}
\mathsf{lin}(T) & \text{if } T = \mathsf{lin}\ p \\
\mathsf{un}(T) & \text{otherwise}
\end{array}$$

A key notion in session types is *duality*. Type duality is standard, as in seminal works [58, 109], and is defined in Fig 5.5. Qualifiers do not influence duality of 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 type is a select type and vice versa. Duality is undefined otherwise. 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 convolution property, namely $\overline{\overline{T}} = T$.

5.4 Session Typing Rules

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 partial 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. 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. If $\mathsf{un}(T)$, then it is possible to add this type to both Γ_1 and Γ_2. The context update operator '+' is used to update the type

Context split

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

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

Context update

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

Figure 5.6: Context split and context update

of a variable with the continuation type in order to enable typing after an input (or branch) or an output (or select) has occurred. When the typing context Γ is updated with a variable having linear type, then the variable must not be present in $\mathsf{dom}(\Gamma)$, otherwise, if the variable is of unrestricted type, 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 as expected:

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

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 for the π-calculus with sessions are given in Fig. 5.7. 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)

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

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

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

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

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

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

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

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

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 $(\nu xy)P$ 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 $U = \mathsf{un}!T.U$. Rule (T-Brch) types an external choice on channel x, checking that each branch continuation P_i follows the respective continuation 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 $T_i = \mathsf{un}\&\{l_i : T_i\}_{i\in I}$ and $T_j = \mathsf{un} \oplus \{l_i : T_i\}_{i\in I}$, 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 x is used in two threads in parallel, it should have an unrestricted type, i.e., $x : \mathsf{unBool}.T$. 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 recursive types, as we will see in Chapter 10.

5.5 Main Results

In this section we present the main properties satisfied by the session type system presented in Fig. 5.7. The following lemmas and theorems are proven in [109].

Weakening allows introduction of new unrestricted channels in a typing context. It holds only for unrestricted channels, for linear ones it would be unsound, since when a linear channel is in a typing context, this means that it should be used in the process it types. 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.

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.

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 types 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, stating the type preservation of a process under structural congruence.

Lemma 5.5.4 (Type Preservation under $\equiv$ 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 $(\nu \widetilde{xy})(P \mid Q)$ 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.

Notice that well-typedness of a process does not imply 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 important result is the subject reduction property, stated 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, since communication occurs only on co-variables, if $P \to Q$ as a result of a communication, then it implies that the session channel in which the communication occurs is restricted and 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 (Type Soundness for Sessions)**.** 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 and of session processes into standard π-calculus processes. We start by giving first the encoding of types and then the encoding of terms.

6.1 Types Encoding

Recall that the syntax of types presented in Section 5.3 uses the notion of qualifiers: lin and un. Linear pretypes denote the standard session types, as 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 [32], we present the encoding of session types 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 added to the syntax of types, like $\mathtt{Int}, \mathtt{String}, \mathtt{Unit} \ldots$, is the identity function, since the same type constructs can be added to the syntax of types in the standard π-calculus, namely:

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

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

Figure 6.1: Encoding of session types

The encoding of session types into standard π-types is given in Fig. 6.1. (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. (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. (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. (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. They are interpreted as linear output and linear input channels carrying variant types with the same labels $l_1 \ldots l_n$ and types the encodings of $\overline{T_1} \ldots \overline{T_n}$ and $T_1 \ldots T_n$, 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

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

The duality on session types boils down to opposite capabilities of linear channel types. The encodings above differ only in the outermost level, that corresponds

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

Figure 6.2: Encoding of session terms

to having ℓ_i or ℓ_o types. The π-calculus 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 wrt that of the sender.

6.2 Terms Encoding

In this section we present the encoding of terms of the π- calculus with sessions into terms of the standard π- calculus. The encoding of terms is different from the encoding of types as it is parametrised by a function f, which is a partial function from variables to variables. We use $\mathsf{dom}(f)$ to denote the domain of function f. We use f_x, f_y as an abbreviation for $f(x), f(y)$, respectively. Let P be a session process. We say that function f is a *renaming function for* P, if f is used in the encoding of P, i.e., $[\![P]\!]_f$, and it satisfies the following conditions: $\mathsf{dom}(f) \supseteq \mathsf{vars}(P)$ meaning that f is defined on both free and bound variables of P; f is the identity function on the bound variables of P and it renames only the free variables of P. We assume that the set of variables used to rename the free variables of P is different from all variables in P, namely different from the set $\mathsf{vars}(P)$. During the encoding of a session process, its renaming function f is updated as in $f, \{x \mapsto c\}$ or $f, \{x, y \mapsto c\}$, where variables x and y are now associated to c, namely $f(x)$ and $f(y)$ are updated to c. The notion of renaming function is extended also to values, being ground values and variables, and is

as expected. We now explain the reason for using a renaming function f in the encoding of terms. Since we are using linear channel types to encode session types, for the linearity to be guaranteed, once a channel is used it should not be used again. However, to enable structured communication and simulate the structure of session types, at every output action a new channel is created and is sent along with the original payload, in order to be used for the continuation of the session. This is called continuation-passing style. Finally, we will often refer to a renaming function f for a session process P, simply as a function f, keeping in mind that it satisfies all the conditions previously presented.

The encoding of terms of the π-calculus with sessions is defined in Fig. 6.2. (E-Variable) states that a variable x is encoded by using a renaming function f for x, meaning that f is defined on x. (E-True) and (E-False) state that the encoding of `true` and `false` is respectively `true` and `false` under any renaming function. In particular, this holds for every ground value, like integers, strings etc., which can be added to both the π-calculus with and without sessions. (E-Inaction) states that the terminated session process is interpreted as the terminated process in the standard π-calculus by using any renaming function. The encoding of the output process, given by (E-Output), is as follows: a new channel c is created and is sent along with $[\![v]\!]_f$ on channel f_x; the encoding of the continuation process P is parametrised in f updated by mapping x to c. The encoding of the input process, given by (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 encodings of selection and branching, given by (E-Selection) and (E-Branching), are generalisations of the output and input ones, respectively. 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, with l_j being the selected label and c the channel created for the continuation of the session, and proceeds as process P encoded in the updated renaming function f. The encoding of branching is more complex: first, there is an input on f_x of a value (typically being a variant value), which is the guard of the **case** process. According to the label of the guard one of the corresponding processes $[\![P_i]\!]_{f,\{x\mapsto c\}}$ for $i \in I$, will be chosen. The encoding of conditional, given by (E-Conditional), is the conditional in the standard π-calculus where the guard v and both branches P and Q are encoded using the renaming function f. The encoding of the parallel composition of processes, given by (E-Composition), is an homomorphism, namely it is the composition of the encodings of the subprocesses. The encoding of the restriction processes is given by (E-Restriction). A new channel c is created and the encoding of P uses a renaming function f updated by associating both x and y to c.

Let us now illustrate the encoding of processes by a simple example. Consider the equality test problem. There are 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:

$$\mathsf{server} \triangleq x?(nr1).x?(nr2).x!\langle nr1 == nr2\rangle.\mathbf{0}$$
$$\mathsf{client} \triangleq y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

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:

$$[\![\mathsf{server}]\!]_{f,\{x,y\mapsto z\}} = z?(nr1, c).c?(nr2, c').(\boldsymbol{\nu} c'')c'!\langle nr1 == nr2, c''\rangle.\mathbf{0}$$
$$[\![\mathsf{client}]\!]_{f,\{x,y\mapsto z\}} = (\boldsymbol{\nu} c)z!\langle 3, c\rangle.(\boldsymbol{\nu} c')c!\langle 5, c'\rangle.c'?(eq, c'').\mathbf{0}$$

Function $f, \{x, y \mapsto z\}$ maps x and y to a new name z, and after that, before every output action, a new channel is created and sent to the partner together with the payload: first channel c, then c' and at the end c'' are created to accommodate the continuation of communication. The endpoints x and y are respectively typed with T and $\overline{T}$, which were previously introduced and encoded.

6.3 Properties of the Encoding

In this section we present some important theoretical results regarding our encoding, by following the requirements stated in [53] about an encoding being a good means for language comparison.

In order to prove these results, the encoding is extended to typing contexts and is presented in Fig. 6.3. The notion of renaming function is thus extended to typing contexts and is as expected. Notice that, the ',' operator in session typing contexts is interpreted as the '$\uplus$' operator in linear π-calculus 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 '$\uplus$' operator, the dual capabilities of linear channels can be *combined* into the connection capability.

$$\begin{array}{rcll} [\![\emptyset]\!]_f & \triangleq & \emptyset & \text{(E-Empty)} \\ [\![\Gamma, x : T]\!]_f & \triangleq & [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] & \text{(E-Gamma)} \end{array}$$

Figure 6.3: Encoding of session typing contexts

6.3.1 Auxiliary Results

In this section we present some auxiliary results needed to prove the correctness of the encoding wrt typing and reduction.

The following proposition states that the encoding of typing contexts, given in Fig. 6.3, is sound and complete wrt to predicates lin and un.

Lemma 6.3.1. Let Γ be a session typing context and q be either lin or un. Then, $q(\Gamma)$ if and only if $q([\![\Gamma]\!]_f)$, for all renaming functions f for Γ.

Proof. The result follows immediately by the encoding of typing contexts given in Fig. 6.3 and by the definitions of lin and un on typing contexts in the π-calculus with sessions and in the standard π-calculus. □

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

Lemma 6.3.2. 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 6.3.3. 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 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 6.3.4 (Split to Combination)**.** Let $\Gamma_1, \ldots, \Gamma_n$ be session typing contexts, such that $\Gamma_1 \circ \ldots \circ \Gamma_n$ is defined. Let f be a renaming function for all Γ_i, for $i \in 1 \ldots n$ such that $[\![\Gamma_1]\!]_f \uplus \ldots \uplus [\![\Gamma_n]\!]_f$ is defined. Then, $[\![\Gamma_1 \circ \ldots \circ \Gamma_n]\!]_f = [\![\Gamma_1]\!]_f \uplus \ldots \uplus [\![\Gamma_n]\!]_f$.

Proof. The result follows immediately by the encoding of typing contexts, given in Fig. 6.3, context split '$\circ$' for session typing contexts, given in Fig. 5.6 and context combination '$\uplus$' for linear typing contexts, given in Fig. 4.6. □

Lemma 6.3.5 (Combination to Split)**.** Let Γ be a session typing context and f a renaming function for Γ and $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \ldots \uplus \Gamma_n^\pi$. Then, $\Gamma = \Gamma_1 \circ \ldots \circ \Gamma_n$ and for all $i \in 1 \ldots n$, $\Gamma_i^\pi = [\![\Gamma_i]\!]_f$.

Proof. The result follows immediately by the encoding of typing contexts, given in Fig. 6.3, context split '$\circ$' for session typing contexts, given in Fig. 5.6 and context combination '$\uplus$' for linear typing contexts, given in Fig. 4.6. □

Lemma 6.3.6. Let $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P. For all functions g with $\mathsf{dom}(g) \supseteq \mathsf{dom}(f)$ such that, for all $x : S \in \Gamma$ with $\mathsf{lin}(S)$ and $g(x) = f(x)$, and for some $y : T \in \Gamma$ with $\mathsf{un}(T)$ and $g(y) \neq f(y)$ and $g(y)$ is fresh, then it is the case that $[\![\Gamma]\!]_g \vdash [\![P]\!]_g$.

Proof. The proof follows immediately from the encoding of processes, the definition of renaming functions and the typing rules for the linear π-calculus. □

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 Session 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. 4.7.

- $T = \mathsf{end}$
  By (E-End) we have $[\![\mathsf{end}]\!] = \emptyset[]$ and $\overline{T} = \mathsf{end}$. It follows by 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.

- $T = ?T.U$
  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 convolution 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 convolution 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 correctness of the encoding wrt typing 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 encoding of T in a typing context being the encoding of Γ, and vice versa.

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

Proof. The proof is done by cases on the value v:

- Case $v = x$:
  By (E-Variable) we have that $[\![x]\!]_f = f_x$ 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 6.3.3 and by (E-Gamma) means that $\Gamma = \Gamma_1, x : T$, where $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$. By (T$\pi$-Var) we have $\mathsf{un}([\![\Gamma_1]\!]_f)$. By Lemma 6.3.1 also $\mathsf{un}(\Gamma_1)$ holds. By applying rule (T-Var) we obtain the result.

- Case $v = \texttt{true}$:
  By (E-True) and (E-Bool) we have that $[\![\texttt{true}]\!]_f = \texttt{true}$ and assume $[\![\Gamma]\!]_f \vdash \texttt{true} : \texttt{Bool}$ and $\mathsf{un}([\![\Gamma]\!]_f)$. By Lemma 6.3.1 also $\mathsf{un}(\Gamma)$ holds. By applying rule (T-Val) we obtain the result.

  Case $v = \texttt{false}$ is symmetrical to the above.

□

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

Proof. The proof is done by induction on the derivation $\Gamma \vdash v : T$.

- Case (T-Var):
  $$\frac{\mathsf{un}(\Gamma)}{\Gamma, x : T \vdash x : T}$$
  By Lemma 6.3.1 we obtain $\mathsf{un}([\![\Gamma]\!]_f)$. By (E-Gamma), (E-Variable), Lemma 6.3.2 and rule (Tπ-Var) we obtain $[\![\Gamma]\!]_f, f_x : [\![T]\!] \vdash f_x : [\![T]\!]$ for any renaming function f for x.

- Case (T-VAL):

$$\frac{\mathsf{un}(\Gamma) \quad v = \mathtt{true} \;/\; \mathtt{false}}{\Gamma \vdash v : \mathtt{Bool}}$$

  By Lemma 6.3.1 we obtain $\mathsf{un}(\llbracket\Gamma\rrbracket_f)$. By applying (E-TRUE) or (E-FALSE) depending on whether v is `true` or `false`, (E-BOOL) and rule (Tπ-VAL) we obtain $\llbracket\Gamma\rrbracket_f \vdash \llbracket v\rrbracket_f : \llbracket\mathtt{Bool}\rrbracket$, for any renaming function f.

□

6.3.3 Typing Processes by Encoding

Recall that we are interested in encoding session types, namely linear pretypes. Hence, in the following we will omit q from the typing rules. The only unrestricted types we encode are `Bool` and `end`. Moreover, the update operator + used in session typing rules boils down to ',' operator, by following the definition of context update given in Fig 5.6.

The following two theorems give the correctness of the encoding wrt typing processes, namely if a session process P is well typed in a session typing context Γ, then the encoding of P is also well typed in the encoding of Γ, and vice versa.

Theorem 6.3.10 (Soundness: Process Typing)**.** If $\llbracket\Gamma\rrbracket_f \vdash \llbracket P\rrbracket_f$ for some renaming function f for P, then $\Gamma \vdash P$.

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

- Case $\mathbf{0}$:
  By (E-INACTION) we have $\llbracket\mathbf{0}\rrbracket_f = \mathbf{0}$ and assume $\llbracket\Gamma\rrbracket_f \vdash \mathbf{0}$, where $\mathsf{un}(\llbracket\Gamma\rrbracket_f)$ holds. By Lemma 6.3.1 we obtain $\mathsf{un}(\Gamma)$. By applying (T-INACT) we conclude this case.

- Case $P \mid Q$:
  By (E-COMPOSITION) we have that $\llbracket P \mid Q\rrbracket_f = \llbracket P\rrbracket_f \mid \llbracket Q\rrbracket_f$ and assume $\llbracket\Gamma\rrbracket_f \vdash \llbracket P\rrbracket_f \mid \llbracket Q\rrbracket_f$, which by rule (T$\pi$-PAR) means:

$$\frac{\Gamma_1^\pi \vdash \llbracket P\rrbracket_f \quad \Gamma_2^\pi \vdash \llbracket Q\rrbracket_f}{\Gamma_1^\pi \uplus \Gamma_2^\pi \vdash \llbracket P\rrbracket_f \mid \llbracket Q\rrbracket_f}$$

  where $\llbracket\Gamma\rrbracket_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.5 $\Gamma_1^\pi = \llbracket\Gamma_1\rrbracket_f$ and $\Gamma_2^\pi = \llbracket\Gamma_2\rrbracket_f$, 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 (E-Conditional) we have that:

$$[\![\textbf{if } v \textbf{ then } P \textbf{ else } Q]\!]_f = \textbf{if } [\![v]\!]_f \textbf{ then } [\![P]\!]_f \textbf{ else } [\![Q]\!]_f$$

  Assume $[\![\Gamma]\!]_f \vdash \textbf{if } [\![v]\!]_f \textbf{ then } [\![P]\!]_f \textbf{ else } [\![Q]\!]_f$, which by rule (T$\pi$-If) means:

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

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.5 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 6.3.8 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 (E-Restriction) we have $[\![(\nu xy)P]\!]_f = (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$ and assume $[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$. Then, either (T$\pi$-Res1) or (T$\pi$-Res2) is the last typing rule applied. We consider both cases in the following:

  – (Tπ-Res1) is applied.

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

    The premise of the rule asserts that $c : \ell_\sharp\ [W]$ and $c \in [\![P]\!]_{f,\{x,y\mapsto c\}}$, which implies $[\![\Gamma]\!]_f, c : \ell_\beta\ [W] \uplus c : \ell_{\overline{\beta}}\ [W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By Lemma 6.3.3 we obtain $[\![\Gamma]\!]_f \uplus c : \ell_\beta\ [W] \uplus c : \ell_{\overline{\beta}}\ [W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. Let $[\![T]\!] = \ell_\beta\ [W]$, then by Lemma 6.3.7 we have $[\![\overline{T}]\!] = \ell_{\overline{\beta}}\ [W]$. Then, by induction hypothesis we have $\Gamma, x : T, y : \overline{T} \vdash P$. By applying rule (T-Res) we obtain $\Gamma \vdash (\nu xy)P$, which concludes the proof.

  – (Tπ-Res2) is applied.

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

    Since $[\![\Gamma]\!]_f, c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$ holds, then $c \notin \mathsf{dom}([\![\Gamma]\!]_f)$. By the combination operation on unrestricted variables and by Lemma 6.3.3 we have $[\![\Gamma]\!]_f \uplus c : \emptyset[] \uplus c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By induction hypothesis and by (E-End) we have $\Gamma, x : \mathsf{end}, y : \mathsf{end} \vdash P$. By (T-Res) we obtain $\Gamma \vdash (\nu xy)P$, which concludes the case.

- Case $x?(y).P$:
  By (E-Input) we have $[\![x?(y).P]\!]_f = f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$ and assume that $[\![\Gamma]\!]_f \vdash f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$, which by rule (T$\pi$-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 6.3.5 we have that $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 6.3.8 we have $\Gamma_1 \vdash x : ?T.U$, where $T^\pi = [\![T]\!]$, $U^\pi = [\![U]\!]$. By induction hypothesis and Lemma 6.3.3 we have $\Gamma_2, y : T, x : U \vdash P$, where $f, \{x \mapsto c\}$ is used in the encoding of the top-right premise. By applying rule (T-Inp) we obtain the result.

- Case $x!\langle v\rangle.P$:
  By (E-Output) we have $[\![x!\langle v\rangle.P]\!]_f = (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f \vdash (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$. Since c is a restricted channel, then either rule (Tπ-Res1) or (Tπ-Res2) is applied. We consider only the former, as the case where (Tπ-Res2) is applied for $c : \emptyset[]$ is similar. By rule (Tπ-Res1) and rule (Tπ-Out) we have the following derivation:

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

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi \uplus \Gamma_3^\pi$. By Lemma 6.3.5 we have $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ and $\Gamma_3^\pi = [\![\Gamma_3]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$. Notice that the type of c is $\ell_\sharp\,[W]$, meaning that c owns both capabilities of input and output. One capability of c is sent along with value $[\![v]\!]_f$ and the other one is used in the encoding of the continuation $[\![P]\!]_{f,\{x\mapsto c\}}$. By Lemma 6.3.8 we have $\Gamma_1 \vdash x : !T.U$ where $\ell_{\mathsf{o}}[T^\pi, U^\pi] = [\![!T.U]\!]$, which by (E-Out) means that $T^\pi = [\![T]\!]$ and $U^\pi = [\![\overline{U}]\!] = \ell_\alpha[W]$, for the capability α. By Lemma 6.3.8 we have $\Gamma_2 \vdash v : T$. By induction hypothesis and by Lemma 6.3.3 we have $\Gamma_3, x : U \vdash P$, where $f, \{x \mapsto c\}$ is used in the encoding of this premise and $[\![U]\!] = \ell_{\overline{\alpha}}[W]$, which is obtained by applying Lemma 6.3.7. By rule (T-Out) we obtain the result $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!\langle v\rangle.P$

- Case $x \triangleright \{l_i : P_i\}_{i\in I}$:
  By (E-Branching) $[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f = f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$.
  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 (Tπ-Inp) and (Tπ-Case) means that the following derivation is possible:

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

where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.5 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By Lemma 6.3.8 we have $\Gamma_1 \vdash x : \&\{l_i : T_i\}_{i\in I}$ and by applying (E-Branch) we have $[\![\&\{l_i : T_i\}_{i\in I}]\!]_f = \ell_{\mathsf{i}}[\langle l_i\_T_i^\pi\rangle_{i\in I}]$, which implies that for all $i \in I$ $[\![T_i]\!] = T_i^\pi$. By induction hypothesis and by Lemma 6.3.3, for all $i \in I$ we have $\Gamma_2, x : T_i \vdash P_i$, where $f, \{x \mapsto c\}$ is used in the encoding of this premise. 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 (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 $[\![\Gamma]\!]_f \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$. Since c is a restricted channel in the encoding of $x \triangleleft l_j.P$, then either rule (Tπ-Res1) or (Tπ-Res2) must have been applied. We consider only the case for (Tπ-Res1), as the one for (Tπ-Res2) and $c : \emptyset[]$ is similar. By (Tπ-Res1), (Tπ-Out), (Tπ-Lval) and (Tπ-Var) we have the following derivation:

$$\dfrac{\dfrac{\Gamma_1^\pi \vdash f_x : \ell_{\mathsf{o}}\,[\langle l_i\_T_i^\pi\rangle_{i\in I}] \quad \Gamma_2^\pi, c : \overline{T_j^\pi} \vdash [\![P]\!]_{f,\{x\mapsto c\}} \quad \dfrac{\dfrac{}{c : T_j^\pi \vdash c : T_j^\pi}\,(\text{T}\pi\text{-Var}) \quad j \in I}{c : T_j^\pi \vdash l_j\_c : \langle l_i\_T_i^\pi\rangle_{i\in I}}\,(\text{T}\pi\text{-Lval})}{[\![\Gamma]\!]_f, c : \ell_\sharp\,[W] \vdash f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}\,(\text{T}\pi\text{-Out})}{[\![\Gamma]\!]_f \vdash (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}\,(\text{T}\pi\text{-Res1})$$

  where $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By using Lemma 6.3.5 we have that $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. Notice that the type of c is $\ell_\sharp\,[W]$, meaning that c owns both capabilities of input and output, because one capability of c is sent along with value l_j_c and the other one is used in the continuation process $[\![P]\!]_{f,\{x\mapsto c\}}$. This implies that $\ell_\sharp\,[W] = T_j^\pi \uplus \overline{T_j^\pi}$. In the case where (T$\pi$-Res2) is applied, c is of type $\emptyset[] = \emptyset[] \uplus \emptyset[]$. By using Lemma 6.3.8 we have that $\Gamma_1 \vdash x : \oplus\{l_i : T_i\}_{i\in I}$. 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 for all $i \in I$. $T_i^\pi = [\![\overline{T_i}]\!]$. By induction hypothesis and by Lemma 6.3.3, we have $\Gamma_2, x : T_j \vdash P$, where $f, \{x \mapsto c\}$ is used in the encoding of this premise. By rule (T-Sel) we obtain the result $\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P$.

□

Theorem 6.3.11 (Completeness: Process Typing)**.** If $\Gamma \vdash P$, then $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P.

Proof. The proof is done by induction on the derivation $\Gamma \vdash P$.

- Case (T-Inact):
$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}}\ (\text{T-Inact})$$
By Lemma 6.3.1 we obtain $\mathsf{un}([\![\Gamma]\!]_f)$. By applying (E-Inaction) and rule (Tπ-Inact) 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 we have $[\![\Gamma_1]\!]_{f'} \vdash [\![P]\!]_{f'}$ for some function f' and $[\![\Gamma_2]\!]_{f''} \vdash [\![Q]\!]_{f''}$ for some function f''. Since $\Gamma_1 \circ \Gamma_2$ is defined by assumption, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and let $f'_D = f' \setminus \bigcup_{d \in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d \in D}\{d \mapsto f''(d)\}$. Hence, for all $d \in D$ we are not making any assumption on $f'(d)$ and $f''(d)$. We define f as $f = \bigcup_{d \in D}\{d \mapsto d'\} \cup f'_D \cup f''_D$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. We can then rewrite the induction hypothesis as $[\![\Gamma_1]\!]_f \vdash [\![P]\!]_f$ and $[\![\Gamma_2]\!]_f \vdash [\![Q]\!]_f$, by applying Lemma 6.3.6. By applying (E-Composition), rule (Tπ-Par) and Lemma 6.3.4 we obtain $[\![\Gamma_1 \circ \Gamma_2]\!]_f \vdash [\![P \mid Q]\!]_f$.

- Case (T-Res):
$$\frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)P}\ (\text{T-Res})$$
Notice that $x, y \notin \mathsf{dom}(\Gamma)$ by typability assumptions. We distinguish the following two cases:

  - Suppose $T \neq \mathsf{end}$. By duality on session types $\overline{T} \neq \mathsf{end}$. By induction hypothesis $[\![\Gamma, x : T, y : \overline{T}]\!]_{f'} \vdash [\![P]\!]_{f'}$, for some function f', which by (E-Gamma) means $[\![\Gamma]\!]_{f'} \uplus f'_x : [\![T]\!] \uplus f'_y : [\![\overline{T}]\!] \vdash [\![P]\!]_{f'}$. Let $f = f'$ and update f with $\{x, y \mapsto c\}$ for a fresh name c that does not occur in the codomain of f. We will use $f, \{x, y \mapsto c\}$ as a renaming function. By Lemma 6.3.7, $[\![T]\!] = \tau$ and $[\![\overline{T}]\!] = \overline{\tau}$. Since $T \neq \mathsf{end}$ and $\overline{T} \neq \mathsf{end}$, we have that $[\![T]\!] = \ell_\alpha\ [W]$ and $[\![\overline{T}]\!] = \ell_{\overline{\alpha}}\ [W]$ and by the combination of linear channel types $\ell_\alpha\ [W] \uplus \ell_{\overline{\alpha}}\ [W] = \ell_\sharp\ [W]$, where W denotes the

pair of carried types, which are irrelevant for this proof. Hence, we can rewrite the induction hypothesis as $[\![\Gamma]\!]_f \uplus c : \ell_\sharp\,[W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By Lemma 6.3.2, $[\![\Gamma]\!]_f, c : \ell_\sharp\,[W] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. By (T$\pi$-Res1) we obtain $[\![\Gamma]\!]_f \vdash (\nu c)[\![P]\!]_{f,\{x,y\mapsto c\}}$, which concludes this case.

- Suppose $T = \mathsf{end}$. By duality on session types $\overline{T} = \mathsf{end}$. By induction hypothesis $[\![\Gamma, x : \mathsf{end}, y : \mathsf{end}]\!]_{f'} \vdash [\![P]\!]_{f'}$, for some function f'. By (E-Gamma) it means that $[\![\Gamma]\!]_{f'} \uplus f'_x : [\![\mathsf{end}]\!] \uplus f'_y : [\![\mathsf{end}]\!] \vdash [\![P]\!]_{f'}$. Let $f = f'$ and update f with $\{x, y \mapsto c\}$ for a fresh name c that does not occur in the codomain of f. We will use $f, \{x, y \mapsto c\}$ as a renaming function. Hence, we can rewrite the induction hypothesis as $[\![\Gamma]\!]_f \uplus c : \emptyset[] \uplus c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$, which by the combination of unrestricted types means $[\![\Gamma]\!]_f \uplus c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. Moreover, $c \notin \mathsf{dom}([\![\Gamma]\!]_f)$, since c is chosen fresh. By Lemma 6.3.2 we obtain $[\![\Gamma]\!]_f, c : \emptyset[] \vdash [\![P]\!]_{f,\{x,y\mapsto c\}}$. We conclude by applying rule (Tπ-Res2).

- Case (T-In):

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

By Lemma 6.3.9 $[\![\Gamma_1]\!]_{f'} \vdash f'_x : [\![?T.U]\!]$, for some function f' and by induction hypothesis $[\![\Gamma_2, y : T, x : U]\!]_{f''} \vdash [\![P]\!]_{f''}$, for some function f''. By applying (E-Inp) we have $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_{\mathrm{i}}[[\![T]\!], [\![U]\!]]$ and by (E-Gamma) we have $[\![\Gamma_2]\!]_{f''} \uplus f''_y : [\![T]\!] \uplus f''_x : [\![U]\!] \vdash [\![P]\!]_{f''}$. By rule (T$\pi$-Var) we can derive $y : [\![T]\!] \vdash y : [\![T]\!]$. Since f'' is a renaming function for P and $y \in \mathsf{fv}(P)$, by the top-right premise of (T-In), then $y \notin \mathsf{dom}([\![\Gamma_2]\!]_{f''})$ and $y \neq f''_x$. Then, $[\![\Gamma_2]\!]_{f''} \uplus y : [\![T]\!] \uplus f''_x : [\![U]\!]$ is defined. By Lemma 4.5.2 we obtain that $[\![\Gamma_2]\!]_{f''} \uplus y : [\![T]\!] \uplus f''_x : [\![U]\!] \vdash [\![P]\!]_{f''}[y/f''_y]$. Since $\Gamma_1 \circ \Gamma_2$ is defined, it means that for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and let $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$. Now, suppose that $f''(x) = c$. Then, we define f as $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D, \{y \mapsto y\} \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Notice that $f''_D(y)$ is defined and is f''_y from the induction hypothesis. Then, $f''_D, \{y \mapsto y\}$ updates f''_y to y by the association of $\{y \mapsto y\}$. Moreover, f is a function since its subcomponents act on disjoint domains. Then, by Lemma 6.3.6 we can rewrite the above as:

$$[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathrm{i}}[[\![T]\!], [\![U]\!]]$$

Since $x, y \notin \mathsf{dom}(\Gamma_2)$, then $[\![\Gamma_2, y : T, x : U]\!]_{f,\{x\mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_2]\!]_f \uplus y : [\![T]\!] \uplus c : [\![U]\!]$. Then, by Lemma 6.3.2:

$$[\![\Gamma_2]\!]_f, y : [\![T]\!], c : [\![U]\!] \vdash [\![P]\!]_{f,\{x\mapsto c\}}$$

By applying (E-Input), rule (Tπ-Inp) and Lemma 6.3.4 we obtain the result $[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash f_x?(y,c).[\![P]\!]_{f,\{x\mapsto c\}}$.

- Case (T-Out):

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

By Lemma 6.3.9 $[\![\Gamma_1]\!]_{f'} \vdash [\![x : !T.U]\!]_{f'}$, for some function f', which by applying (E-Out) means that $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_o[[\![T]\!], [\![\overline{U}]\!]]$. By Lemma 6.3.9 $[\![\Gamma_2]\!]_{f''} \vdash [\![v]\!]_{f''} : [\![T]\!]$ for some function f''. By induction hypothesis and by applying (E-Gamma) we have $[\![\Gamma_3]\!]_{f'''} \uplus f'''_x : [\![U]\!] \vdash [\![P]\!]_{f'''}$, for some f'''. Since $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3$ is defined, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \cap \mathsf{dom}(\Gamma_3)$ it must be the case that $\Gamma_1(x) = \Gamma_2(x) = \Gamma_3(x) = T$ and $\mathsf{un}(T)$. Now, let $D = \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \cap \mathsf{dom}(\Gamma_3)$. Let $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$, $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$ and $f'''_D = f''' \setminus \bigcup_{d\in D}\{d \mapsto f'''(d)\}$. Suppose $f'''_x = c$. Then, define f as $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_d \cup f'''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Notice that f is a function because its subcomponents act on disjoint domains. Then, by Lemma 6.3.6, the above can be rewritten as:

$$[\![\Gamma_1]\!]_f \vdash f_x : \ell_o[[\![T]\!], [\![\overline{U}]\!]] \qquad [\![\Gamma_2]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$$

Since $x \notin \mathsf{dom}(\Gamma_3)$, then $[\![\Gamma_3, x : U]\!]_{f,\{x\mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_3]\!]_f \uplus c : [\![U]\!]$. Then, the induction hypothesis becomes:

$$[\![\Gamma_3]\!]_f \uplus c : [\![U]\!] \vdash [\![P]\!]_{f,\{x\mapsto c\}}$$

Assume $U \neq \mathsf{end}$ and hence $\overline{U} \neq \mathsf{end}$. By rule (T$\pi$-Var) we can derive $c : [\![\overline{U}]\!] \vdash c : [\![\overline{U}]\!]$. By rule (T$\pi$-Out) and by using Lemma 6.3.7 and "$\uplus$" operator to obtain $c : \ell_\sharp\,[W]$, we have the following derivation:

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

Then, by Lemma 6.3.2 and by applying (Tπ-Res1) we have the following:

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

The case where $U = \overline{U} = \mathsf{end}$, which yields $c : \emptyset[]$, is symmetrical and is obtained by using (Tπ-Res2) instead of (Tπ-Res1). By Lemma 6.3.4 and (E-Outputt) we conclude this case.

- Case (T-Brch):

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

By Lemma 6.3.9 $[\![\Gamma_1]\!]_{f'} \vdash [\![x : \&\{l_i : T_i\}_{i\in I}]\!]_{f'}$, for some function f', which by applying (E-Branch) means $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_i[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$. By induction hypothesis $[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![T_i]\!] \vdash [\![P_i]\!]_{f''}$ for all $i \in I$, for some function f''. Since $\Gamma_1 \circ \Gamma_2$ is defined, it means that for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and define $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$. Now, suppose that $f''(x) = c$. Then, define $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. Then, by applying Lemma 6.3.6, the above can be rewritten as:

$$[\![\Gamma_1]\!]_f \vdash f_x : \ell_i[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \qquad [\![\Gamma_2]\!]_f \uplus c : [\![T_i]\!] \vdash [\![P_i]\!]_{f,\{x\mapsto c\}} \text{ for all } i \in I$$

Since $x \notin \mathsf{dom}(\Gamma_2)$, then $[\![\Gamma_2, x : T_i]\!]_{f,\{x\mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_2]\!]_f \uplus c : [\![T_i]\!]$, as we did in the previous cases. By rules (Tπ-Case), and (Tπ-Var) for deriving $y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I}$, and Lemma 6.3.2 we have the following derivation:

$$\dfrac{\dfrac{\text{(T}\pi\text{-Case)}\ \text{(T}\pi\text{-Var)}}{y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \vdash y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I}} \qquad [\![\Gamma_2]\!]_f, c : [\![T_i]\!] \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}\ \forall i \in I}{[\![\Gamma_2]\!]_f, y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

Then, by applying (Tπ-Inp) we have:

$$\text{(T}\pi\text{-Inp)}\ \frac{[\![\Gamma_1]\!]_f \vdash f_x : \ell_i[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \quad [\![\Gamma_2]\!]_f, y : \langle l_i\_[\![T_i]\!]\rangle_{i\in I} \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \vdash f_x?(y).\, \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

By (E-Branching) and Lemma 6.3.4 we conclude this case.

- Case (T-Sel):

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

By Lemma 6.3.9 $[\![\Gamma_1]\!]_{f'} \vdash [\![x : \oplus\{l_i : T_i\}_{i\in I}]\!]_{f'}$, for some function f', which by applying (E-Select) means that $[\![\Gamma_1]\!]_{f'} \vdash f'_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}]$. By induction hypothesis and (E-Gamma) $[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![T_j]\!] \vdash [\![P]\!]_{f''}$ for $j \in I$, for some function f''. Since $\Gamma_1 \circ \Gamma_2$ is defined, it means that for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it is the case that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Now, let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and let $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$ and suppose that $f''(x) = c$. Then, define f as $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. Then, by Lemma 6.3.6 we can rewrite the above as follows:

$$[\![\Gamma_1]\!]_f \vdash f_x : \ell_{\mathsf{o}}[\langle l_i\_[\![T_i]\!]\rangle_{i\in I}] \qquad [\![\Gamma_2]\!]_f \uplus c : [\![T_j]\!] \vdash [\![P]\!]_{f,\{x\mapsto c\}} \text{ for } j \in I$$

Since $x \notin \mathsf{dom}(\Gamma_2)$, then $[\![\Gamma_2, x : T_j]\!]_{f,\{x\mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_2]\!]_f \uplus c : [\![T_j]\!]$, as we did in the previous cases. By applying (Tπ-Var) to derive $c : [\![\overline{T_j}]\!]$, and by (T$\pi$-LVal) we have:

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

Assume $T_j \neq \mathsf{end}$ and hence $\overline{T_j} \neq \mathsf{end}$. By rule (T$\pi$-Out) and by using Lemma 6.3.7 and "$\uplus$" operator to obtain $c : \ell_\sharp\,[W]$, we have the following derivation:

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

Then, by Lemma 6.3.2 and by applying (Tπ-Res1) we have:

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

The case where $T_j = \overline{T_j} = \mathsf{end}$, which yields $c : \emptyset[]$, is symmetrical and is obtained by using (Tπ-Res2) instead of (Tπ-Res1). By (E-Selection) and Lemma 6.3.4 we conclude this case.

□

6.3.4 Operational Correspondence

In this section we prove the operational correspondence. This property states that the encoding of session-typed processes is sound and complete wrt the operational semantics of the π-calculus with and without sessions. We start by introducing the notion of *evaluation context* and give two auxiliary lemmas that are used in the proof of the operational correspondence.

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

$$\mathcal{E}[\cdot] \triangleq [\cdot] \mid (\nu xy)[\cdot]$$

Given a session process P, we say that $\mathcal{E}[\cdot]$ is a *suitable* evaluation context for process P, if whenever $\mathcal{E}[\cdot] = (\nu xy)[\cdot]$, then $x, y \in \mathsf{fv}(P)$. Hence, $[\cdot]$ is always a suitable evaluation context for every session process. In the remainder of the thesis we will consider only suitable evaluation contexts and we will refer to them simply as evaluation contexts.

Lemma 6.3.13. Let Q be a session process and let $Q[v/z]$ denote process Q where variable z is substituted by value v. Then,

$$[\![Q[v/z]]\!]_f = [\![Q]\!]_f[[\![v]\!]_f/f_z]$$

for all renaming functions f for Q and v.

Proof. It follows immediately by the encoding of processes given in Fig. 6.2 and by the standard substitution of variables by values in a process. □

Lemma 6.3.14 (Structural Congruence and Encoding)**.** Let P and P' be session processes. Then, $P \equiv P'$ if and only if $[\![P]\!]_f \equiv [\![P']\!]_f$ for all renaming functions f for P and P'.

Proof. The proof is done by cases on the structural congruence relation. □

Let $\hookrightarrow$ denote structural congruence extended with a case normalisation, namely a reduction by using (Rπ-Case).

Theorem 6.3.15 (Operational Correspondence)**.** Let P be a session process, Γ a session typing context, and f a renaming function for P such that $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$. Then, the following statements hold.

1. If $P \to P'$, then $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$.
2. If $[\![P]\!]_f \to Q$, then there are $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, and either $f' = f$ or $f' = f, \{x, y \mapsto c\}$ for x, y such that (νxy) appears in $\mathcal{E}[P]$.

Proof. Notice that, since $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$, by Theorem 6.3.10 it means that $\Gamma \vdash P$. We split the proof as follows.

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

- Case (R-Com):

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

By the encoding of session processes we have

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

Since z is bound with scope Q_2 it means that $f_z = z$. Notice that since P is a session-typed process, it means that $x \notin \mathsf{fv}(Q_2)$ and $y \notin \mathsf{fv}(Q_1)$. Then, $f, \{x, y \mapsto c, x \mapsto c'\}$ and $f, \{x, y \mapsto c, y \mapsto c'\}$ can be subsumed by $f, \{x, y \mapsto c'\}$. We can rewrite the above as:

$$(\nu c')([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/z])$$

On the other hand we have:

$$\begin{array}{rcl}
[\![P']\!]_f & = & [\![(\nu xy)(Q_1 \mid Q_2[v/z])]\!]_f \\
& = & (\nu c')([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2[v/z]]\!]_{f,\{x,y\mapsto c'\}}) \\
& = & (\nu c')([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_{f,\{x,y\mapsto c'\}}/[\![z]\!]_{f,\{x,y\mapsto c'\}}]) \\
& = & (\nu c')([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/f_z]) \\
& = & (\nu c')([\![Q_1]\!]_{f,\{x,y\mapsto c'\}} \mid [\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/z])
\end{array}$$

In order to obtain $[\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_{f,\{x,y\mapsto c'\}}/[\![z]\!]_{f,\{x,y\mapsto c'\}}]$ in line 3 we apply Lemma 6.3.13. Function f coincides with $f, \{x, y \mapsto c'\}$ when applied to value v and variable z and $f_z = z$, so we can obtain $[\![Q_2]\!]_{f,\{x,y\mapsto c'\}}[[\![v]\!]_f/z]$.

The above implies:

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

- Case (R-Sel):

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

By the encoding of session processes we have

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

Notice that since P is a well-typed session process, it means that for all $i \in I$, $x \notin \mathsf{fv}(P_i)$ and $y \notin \mathsf{fv}(Q)$. Then, both functions $f, \{x, y \mapsto c, x \mapsto c'\}$ and $f, \{x, y \mapsto c, y \mapsto c'\}$ can be subsumed by $f, \{x, y \mapsto c'\}$. We can rewrite the above as:

$$(\nu c')([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}})$$

On the other hand we have:

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

The above implies:

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

- 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

$$\begin{array}{rcl}
[\![P]\!]_f & = & [\![\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]_f \\
& = & \mathbf{if}\ \mathtt{true}\ \mathbf{then}\ [\![P_1]\!]_f\ \mathbf{else}\ [\![P_2]\!]_f \\
& \to & [\![P_1]\!]_f
\end{array}$$

- Case (R-Res):

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

By the encoding of session processes we have

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

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

- Case (R-Par):

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

By the encoding of session processes we have

$$[\![P \mid Q]\!]_f = [\![P]\!]_f \mid [\![Q]\!]_f \qquad [\![P' \mid Q]\!]_f = [\![P']\!]_f \mid [\![Q]\!]_f$$

By induction hypothesis we have that $[\![P]\!]_f \to\hookrightarrow [\![Q]\!]_f$. We conclude that $[\![P]\!]_f \mid [\![Q]\!]_f \to\hookrightarrow [\![P']\!]_f \mid [\![Q]\!]_f$ by applying (Rπ-Par) and (Rπ-Struct) and the transitivity of the reduction relation.

- Case (R-Struct):

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

Trivial case, by applying (Rπ- Struct) and Lemma 6.3.14 on the induction hypothesis.

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

- Case $P = P_1 \mid P_2$.

Since $\Gamma \vdash P_1 \mid P_2$, by inversion on (T-Par) we have that $\Gamma_1 \vdash P_1$ and $\Gamma_2 \vdash P_2$ and $\Gamma = \Gamma_1 \circ \Gamma_2$. By (E-Composition) we have $[\![P_1 \mid P_2]\!]_f = [\![P_1]\!]_f \mid [\![P_2]\!]_f$ and assume $[\![P_1]\!]_f \mid [\![P_2]\!]_f \to Q$. There are only the following cases to be considered:

  - Only $[\![P_1]\!]_f$ reduces. Let $[\![P_1]\!]_f \to R$. Then, by rule (Rπ- Par) $[\![P_1]\!]_f \mid [\![P_2]\!]_f \to R \mid [\![P_2]\!]_f$ and let $Q = R \mid [\![P_2]\!]_f$. Since P_1 is a subprocess of P, by induction hypothesis there exist $P_1', \mathcal{E}'[\cdot]$, such that $\mathcal{E}'[P_1] \to \mathcal{E}'[P_1']$ and $R \hookrightarrow [\![P_1']\!]_{f''}$, where either $f'' = f$ or $f'' =$

$f, \{z, w \mapsto d\}$, such that (νzw) appears in $\mathcal{E}'[P_1]$. Choose $\mathcal{E}[\cdot] = \mathcal{E}'[\cdot]$. Since $\mathcal{E}[\cdot]$ is a suitable context for P_1 and $\Gamma \vdash P_1 \mid P_2$ it means that for all (νzw) that appear in $\mathcal{E}[P_1]$, it is the case that $z, w \notin \mathsf{fv}(P_2)$. Hence, by structural congruence we obtain that $\mathcal{E}[P_1] \mid P_2 \equiv \mathcal{E}[P_1 \mid P_2]$ (1). By rule (R-Par) we have $\mathcal{E}[P_1] \mid P_2 \to \mathcal{E}[P_1'] \mid P_2$ (2). Again, by structural congruence we have $\mathcal{E}[P_1'] \mid P_2 \equiv \mathcal{E}[P_1' \mid P_2]$ (3). By rule (R-Struct) on (1), (2), (3) we can conclude that $\mathcal{E}[P_1 \mid P_2] \to \mathcal{E}[P_1' \mid P_2]$. Let $P' = P_1' \mid P_2$. We observe that $\mathcal{E}[P_1' \mid P_2] = \mathcal{E}[P']$. The last thing to show is that $Q \hookrightarrow [\![P']\!]_{f'}$, where either $f' = f$ or $f' = f, \{x, y \mapsto c\}$ and (νxy) appears in $\mathcal{E}[P]$. Choose $f' = f''$.

If $f' = f'' = f$, then by context closure of structural congruence and by applying rules (Rπ-Struct) and (Rπ-Par) we obtain the result $Q = R \mid [\![P_2]\!]_f \hookrightarrow [\![P_1']\!]_f \mid [\![P_2]\!]_f = [\![P']\!]_f$.

If $f' = f'' = f, \{z, w \mapsto d\}$ and since for all (νzw) that appear in $\mathcal{E}[P_1]$, it is the case that $z, w \notin \mathsf{fv}(P_2)$, we can use f' instead of f to encode P_2 and we obtain $Q = R \mid [\![P_2]\!]_f \hookrightarrow [\![P_1']\!]_{f'} \mid [\![P_2]\!]_{f'} = [\![P']\!]_{f'}$.

– Only $[\![P_2]\!]_f$ reduces. This case is symmetrical to the previous one, by simply exchanging the roles of P_1 and P_2.

– $[\![P_1]\!]_f$ and $[\![P_2]\!]_f$ communicate and both reduce. Since communication occurs between two processes in parallel, it means that rule (Rπ-Com) is applied. Let $[\![P_1]\!]_f$ perform and output action and $[\![P_2]\!]_f$ perform and input action – the symmetrical case is similar. But then, since these are encodings of session-typed processes and $\Gamma_1 \vdash P_1$ and $\Gamma_2 \vdash P_2$ and $\Gamma = \Gamma_1 \circ \Gamma_2$, there are only two possible cases:

  * Case $P_1 \mid P_2 = x!\langle v \rangle.P_1' \mid y?(z).P_2'$.
    By (E-Composition), (E-Output) and (E-Input), we have that:

$$
\begin{aligned}
[\![P_1]\!]_f \mid [\![P_2]\!]_f &= (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P_1']\!]_{f,\{x \mapsto c\}} \mid f_y?(z, c).[\![P_2']\!]_{f,\{y \mapsto c\}} \\
&\to [\![P_1']\!]_{f,\{x \mapsto c\}} \mid [\![P_2']\!]_{f,\{y \mapsto c\}}[[\![v]\!]_f / z] \\
&= [\![P_1']\!]_{f,\{x,y \mapsto c\}} \mid [\![P_2']\!]_{f,\{x,y \mapsto c\}}[[\![v]\!]_f / z] \\
&= Q
\end{aligned}
$$

    Since z is bound with scope P_2', then $f_z = z$. The assumption $[\![P]\!]_f \to Q$ implies that $f_x = f_y$. Since $\Gamma \vdash P_1 \mid P_2$, it means that $x \notin \mathsf{fv}(P_2)$ and $y \notin \mathsf{fv}(P_1)$. Hence, in line 3 above we used function $f, \{x, y \mapsto c\}$ to subsume both $f, \{x \mapsto c\}$ and $f, \{y \mapsto c\}$. We will show that there are $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P_1 \mid P_2] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, where either $f' = f$ or $f' = f, \{z, w \mapsto d\}$ for

z, w such that $(\boldsymbol{\nu} zw)$ appears in $\mathcal{E}[P]$. Choose $\mathcal{E}[\cdot] = (\boldsymbol{\nu} xy)[\cdot]$, then by rule (R-Com) on $\mathcal{E}[P_1 \mid P_2]$ we have that:

$$(\boldsymbol{\nu} xy)(x!\langle v\rangle.P'_1 \mid y?(z).P'_2) \to (\boldsymbol{\nu} xy)(P'_1 \mid P'_2[v/z])$$

Choose $P' = P'_1 \mid P'_2[v/z]$ and $f' = f, \{x, y \mapsto c\}$. By the encoding of P' we have:

$$\begin{aligned}
[\![P']\!]_{f'} &= [\![P'_1 \mid P'_2[v/z]]\!]_{f'} \\
&= [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid [\![P'_2]\!]_{f,\{x,y\mapsto c\}}[[\![v]\!]_f / f_z] \\
&= [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid [\![P'_2]\!]_{f,\{x,y\mapsto c\}}[[\![v]\!]_f / z] \\
&= Q
\end{aligned}$$

Line 2 above holds by Lemma 6.3.13. Notice that $v \neq x, y$ and $z \neq x, y$ by the well-typedness of P. Hence, we can simply use f instead of f' in the encoding of v and z and since z is bound, $f_z = z$. This concludes the case.

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

$$\begin{aligned}
[\![P_1]\!]_f \mid [\![P_2]\!]_f \;=\;& (\boldsymbol{\nu} c) f_x!\langle l_j_c\rangle.[\![P'_1]\!]_{f,\{x\mapsto c\}} \mid \\
& \qquad f_y?(z).\ \mathbf{case}\ z\ \mathbf{of}\ \{l_i_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I} \\
\to\;& [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid \mathbf{case}\ l_j_c\ \mathbf{of}\ \{l_i_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I} \\
=\;& [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid \mathbf{case}\ l_j_c\ \mathbf{of}\ \{l_i_c \triangleright [\![P''_i]\!]_{f,\{x,y\mapsto c\}}\}_{i\in I} \\
=\;& Q
\end{aligned}$$

The assumption $[\![P]\!]_f \to Q$ implies that $f_x = f_y$. Since $\Gamma \vdash P_1 \mid P_2$, it means that $x \notin \mathsf{fv}(P_2)$ and $y \notin \mathsf{fv}(P_1)$. Hence, in line 4 above we used function $f, \{x, y \mapsto c\}$ to subsume both $f, \{x \mapsto c\}$ and $f, \{y \mapsto c\}$. We need to show that there are $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P_1 \mid P_2] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, where either $f' = f$ or $f' = f, \{z, w \mapsto d\}$ for z, w such that $(\boldsymbol{\nu} zw)$ appears in $\mathcal{E}[P]$. Choose $\mathcal{E}[\cdot] = (\boldsymbol{\nu} xy)[\cdot]$. Then, by rule (R-Sel) on $\mathcal{E}[P_1 \mid P_2]$ we have that:

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

Choose $P' = P'_1 \mid P''_j$ and $f' = f, \{x, y \mapsto c\}$. By the encoding of P' we have:

$$[\![P']\!]_{f'} = [\![P'_1 \mid P''_j]\!]_{f'} = [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid [\![P''_j]\!]_{f,\{x,y\mapsto c\}}$$

It remains to show that $Q \hookrightarrow [\![P']\!]_{f,\{x,y\mapsto c\}}$. By rules (R$\pi$-Case) and (R$\pi$-Par) we obtain the result:

$$\begin{array}{rcl} Q & = & [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid \mathbf{case}\, l_j\_c\, \mathbf{of}\, \{l_i\_c \triangleright [\![P''_i]\!]_{f,\{x,y\mapsto c\}}\}_{i\in I} \\ & \to & [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid [\![P''_j]\!]_{f,\{x,y\mapsto c\}} \\ & = & [\![P']\!]_{f,\{x,y\mapsto c\}} \end{array}$$

- Case $P = (\nu xy)P_1$.

  Since $\Gamma \vdash (\nu xy)P_1$, then there is a session type T, such that by inversion on (T-Res) we have $\Gamma, x : T, y : \overline{T} \vdash P_1$. By (E-Restriction) we have $[\![(\nu xy)P_1]\!]_f = (\nu c)[\![P_1]\!]_{f,\{x,y\mapsto c\}}$ and assume $(\nu c)[\![P_1]\!]_{f,\{x,y\mapsto c\}} \to Q$. This implies that the reduction comes from $[\![P_1]\!]_{f,\{x,y\mapsto c\}}$. The reason is that in the standard π-calculus restriction does not enable any new communication in addition to the ones performed by process $[\![P_1]\!]_{f,\{x,y\mapsto c\}}$; differently from communications in the session π-calculus which occur only under restricted co-variables. Hence, $Q = (\nu c)R$. By rule (Rπ-Res) it means that $[\![P_1]\!]_{f,\{x,y\mapsto c\}} \to R$. By induction hypothesis there are $P'_1, \mathcal{E}'[\cdot]$ such that $\mathcal{E}'[P_1] \to \mathcal{E}'[P'_1]$ and $R \hookrightarrow [\![P'_1]\!]_{f''}$, where either $f'' = f, \{x, y \mapsto c\}$ or $f'' = f, \{x, y \mapsto c\}, \{z, w \mapsto d\}$ such that (νzw) appears in $\mathcal{E}'[P_1]$. We need to prove that there are $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[(\nu xy)P_1] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, where either $f' = f$ or $f' = f, \{k, l \mapsto e\}$ such that (νkl) appears in $\mathcal{E}[P_1]$. We distinguish the following cases according to the structure of $\mathcal{E}'[\cdot]$.

  - $\mathcal{E}'[\cdot] = [\cdot]$. The induction hypothesis is rewritten as $P_1 \to P'_1$. By rule (R-Res) we have $(\nu xy)P_1 \to (\nu xy)P'_1$. Choose $P' = (\nu xy)P'_1$ and $\mathcal{E}[\cdot] = [\cdot]$. We know by induction hypothesis that $R \hookrightarrow [\![P'_1]\!]_{f''}$, where either $f'' = f, \{x, y \mapsto c\}$ or $f'' = f, \{x, y \mapsto c\}, \{z, w \mapsto d\}$, such that (νzw) appears in P_1.

    If $f'' = f, \{x, y \mapsto c\}$, choose $f' = f$. Then, $R \hookrightarrow [\![P'_1]\!]_{f,\{x,y\mapsto c\}}$ and $[\![P']\!]_{f'} = (\nu c)[\![P'_1]\!]_{f,\{x,y\mapsto c\}}$. By context closure of structural congruence and (Rπ-Struct) and (Rπ-Res) we have $(\nu c)R \hookrightarrow (\nu c)[\![P'_1]\!]_{f,\{x,y\mapsto c\}}$.

    If $f'' = f, \{x, y \mapsto c\}, \{z, w \mapsto d\}$, choose $f' = f, \{z, w \mapsto d\}$. We can distinguish two possible cases: either $\{x, y\} = \{z, w\}$ and d overrides c, or $\{x, y\} \cap \{z, w\} = \emptyset$. All other cases would violate linearity and hence the well-typedness assumption.

If $\{x, y\} = \{z, w\}$ and d overrides c, then it is the case that $f'' = f, \{x, y \mapsto c\}, \{z, w \mapsto d\} = f, \{x, y \mapsto d\}$ and $f' = f, \{x, y \mapsto d\}$. Then, the induction hypothesis can be rewritten as $R \hookrightarrow [\![P'_1]\!]_{f,\{x,y\mapsto d\}}$. The encoding of P' under f' is $[\![P']\!]_{f'} = (\nu c)[\![P'_1]\!]_{f,\{x,y\mapsto d\},\{x,y\mapsto c\}}$ and (νxy) appears in $\mathcal{E}[P_1] = P_1$. Since $(\nu xy)P_1$ is well-typed and (νxy) appears in P_1, by Lemma 5.5.1 it must be that the outermost x, y are terminated channels, namely $T = \overline{T} = \mathsf{end}$. The result follows by context closure of structural congruence and by rules (Rπ-Struct) and (Rπ-Res).

If $\{x, y\} \cap \{z, w\} = \emptyset$, then it holds that $f'' = f, \{x, y \mapsto c\}, \{z, w \mapsto d\} = f, \{z, w \mapsto d\}, \{x, y \mapsto c\}$. Then, the induction hypothesis can be rewritten as $R \hookrightarrow [\![P'_1]\!]_{f,\{z,w\mapsto d\},\{x,y\mapsto c\}}$. The encoding of P' under f' is $[\![P']\!]_{f'} = (\nu c)[\![P'_1]\!]_{f,\{z,w\mapsto d\},\{x,y\mapsto c\}}$ and (νzw) appears in $\mathcal{E}[P_1]$. Hence, by context closure of structural congruence and by rules (Rπ-Struct) and (Rπ-Res) we have $(\nu c)R \hookrightarrow (\nu c)[\![P'_1]\!]_{f,\{z,w\mapsto d\},\{x,y\mapsto c\}}$.

- $\mathcal{E}'[\cdot] = (\nu xy)[\cdot]$. We have that $(\nu xy)P_1 \to (\nu xy)P'_1$. Let $P' = (\nu xy)P'_1$ and $\mathcal{E}[\cdot] = [\cdot]$. The result follows by context closure of structural congruence and by rules (Rπ-Res) and (Rπ-Struct).
- $\mathcal{E}'[\cdot] = (\nu x'y')[\cdot]$, such that $\{x', y'\} \cap \{x, y\} = \emptyset$. We have that $(\nu x'y')P_1 \to (\nu x'y')P'_1$. Choose $P' = (\nu xy)P'_1$ and $\mathcal{E}[\cdot] = \mathcal{E}'[\cdot]$. We need to show that $(\nu x'y')(\nu xy)P_1 \to (\nu x'y')(\nu xy)P'_1$. By structural congruence and by rules (R-Res) and (R-Struct) we obtain the result.

□

6.4 Corollaries from the Encoding

In this section we show how we can use our encoding and properties from the standard typed π-calculus to derive the analogous properties in the π-calculus with session types. Before proving the subject reduction and type safety theorems, we give the following auxiliary lemmas. We start with an auxiliary result, that of type preservation for the structural congruence.

Lemma 6.4.1 (Type Preservation under $\equiv$ for Sessions by Encoding)**.** Let P be a session process. If $\Gamma \vdash P$ and $P \equiv P'$, then $\Gamma \vdash P'$.

Proof. Assume $\Gamma \vdash P$ and $P \equiv P'$. By Theorem 6.3.11 we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P. By Lemma 6.3.14 $[\![P]\!]_f \equiv [\![P']\!]_f$, then by Lemma 4.5.5 we have $[\![\Gamma]\!]_f \vdash [\![P']\!]_f$. We conclude by Theorem 6.3.10. □

Now we are ready to prove the subject reduction property for the π-calculus with sessions by using our encoding and by the corresponding subject reduction for the linear π-calculus.

Theorem 6.4.2 (Subject Reduction for Sessions by Encoding)**.** Let P be a session process. If $\Gamma \vdash P$ and $P \to P'$, then $\Gamma \vdash P'$.

Proof. Assume $\Gamma \vdash P$ and $P \to P'$. By Theorem 6.3.11 we have $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$, for some renaming function f for P and by point 1. of Theorem 6.3.15 we have that $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$. Let Q be the π-calculus process such that $[\![P]\!]_f \to Q \hookrightarrow [\![P']\!]_f$. By subject reduction for the linear π-calculus, given by Theorem 4.5.6, we have $[\![\Gamma]\!]_f \vdash Q$. By type preservation for structural congruence, given by Lemma 4.5.5, and by subject reduction for the linear π-calculus, we have $[\![\Gamma]\!]_f \vdash [\![P']\!]_f$. By the assumption that $P \to P'$ and the operational semantics rules in the π-calculus with sessions which state that communication occurs only in restricted co-variables, we can conclude that $[\![\Gamma]\!]_f$ is closed. By Theorem 6.3.10 we conclude that $\Gamma \vdash P'$. □

Theorem 6.4.3 (Type Safety for Sessions by Encoding)**.** Let P be a session process. If $\vdash P$, then P is well formed.

Proof. By Theorem 6.3.11 we have $\vdash [\![P]\!]_f$ for some renaming function f for P. By type safety in the linear π-calculus, given by Theorem 4.5.8, we have that $[\![P]\!]_f$ is well formed. The result follows immediately by applying the notion of well-formedness in session π-calculus, given by Definition 5.5.5 and the encoding of processes given in Fig. 6.2. □

At this point we can derive the main result, that of type soundness, which states the absence of runtime errors of well-typed programs. It follows immediately from subject reduction given by Theorem 6.4.2 and type safety given by Theorem 6.4.3, which we proved by using the corresponding properties in the standard typed π-calculus and our encoding.

Theorem 6.4.4 (Type Soundness for Sessions by Encoding)**.** Let P be a session process. If $\vdash P$ and $P \to^* Q$, then Q is well formed.

Proof. The result follows immediately from Theorem 6.4.2 and Theorem 6.4.3. □

Part III

Advanced Features on Safety by Encoding

Introduction to Part III

In the π-calculus with session types, different typing features have been added. Subtyping relation for (recursive) session types is added in [48]. Bounded polymorphism is added in [45] as a further extension to subtyping. The authors in [89] 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 to which extent this redundancy is necessary. After having analysed 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 to the π-calculus with sessions and to the encoding. They present subtyping, polymorphism, higher-order, and recursion respectively, and study the encoding wrt these extensions. Chapter 11 presents an optimisation on the usage of linear channels. 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 has been studied in the standard typed π-calculus [98, 101] and later on in the π-calculus with session types [48]. In this section we show that subtyping in the standard π-calculus is enough to derive subtyping in session types.

7.1 Subtyping Rules

Subtyping rules for the π-calculus with sessions are given in Fig. 7.1 and the ones for the standard typed 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 co-variant on the payload type, whether the output rule states that subtyping is contra-variant on the payload 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. 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 (S-Sel) states it is contra-variant in breadth.

We now focus on subtyping for standard π-calculus types. Rules (Sπ-Refl) and (Sπ-Trans) state that subtyping is a pre-order. Rules (Sπ-ii) and (Sπ-oo) define subtyping for input and output channel types, respectively. The input action is co-variant in the carried types, whether the output action is contra-variant. Rule (Sπ-Variant) defines subtyping for variant types which is co-variant both in depth and in breadth, namely in the carried types and in the set of labelled types.

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

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

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

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

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

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

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

Figure 7.2: Subtyping rules for the standard π-calculus

7.2 Properties

In order to use the encoding of the π-calculus with session types to derive basic properties like subject reduction, type safety etc., in presence of subtyping, we need to prove the correctness of the encoding wrt subtyping.

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 wrt Subtyping)**.** If $[\![T]\!] \leq [\![T']\!]$, then $T <: T'$.

Proof. The proof is done by induction on the structure of session types T, T'.

- 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 $T = T' = \mathsf{end}$:
  By (E-End) we have $[\![T]\!] = [\![T']\!] = \emptyset[]$. By rule (S$\pi$-Refl) we have that $\emptyset[] \leq \emptyset[]$. 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 encoding of input means $\ell_{\mathsf{i}}[[\![T_1]\!], [\![U_1]\!]] \leq \ell_{\mathsf{i}}[[\![T_2]\!], [\![U_2]\!]]$. The last rule applied is (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 by encoding of output means $\ell_{\mathsf{o}}[[\![T_1]\!], [\![\overline{U_1}]\!]] \leq \ell_{\mathsf{o}}[[\![T_2]\!], [\![\overline{U_2}]\!]]$. The last rule applied is (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, 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) 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 by encoding of branch means $\ell_{\mathsf{i}}\ [\langle l_i \_ [\![T_i]\!]\rangle_{i\in I}] \leq \ell_{\mathsf{i}}\ [\langle l_j \_ [\![T'_j]\!]\rangle_{j\in J}]$. The last rule applied must have been (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 $[\![T_i]\!] \leq [\![T'_j]\!]$ for all $i \in I$ and $I \subseteq J$. By induction hypothesis we have that $T_i <: T'_j$ for all $i \in I$ and $I \subseteq J$. By applying rule (S-Brch) 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 by encoding of select means $\ell_{\mathsf{o}}\ [\langle l_i \_ [\![\overline{T_i}]\!]\rangle_{i\in I}] \leq \ell_{\mathsf{o}}\ [\langle l_j \_ [\![\overline{T'_j}]\!]\rangle_{j\in J}]$. The last rule applied must have been (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), $[\![\overline{T'_j}]\!] \leq [\![\overline{T_i}]\!]$ for all $j \in J$ and $J \subseteq I$. By Lemma 7.2.1, we obtain $[\![T_i]\!] \leq [\![T_j]\!]$ for all $j \in J$ and $J \subseteq I$. By induction hypothesis we have that $T_i <: T'_j$ for all $j \in J$ and $J \subseteq I$. By applying rule (S-Sel) on the induction hypothesis we obtain $\oplus\{l_i : T_i\}_{i\in I} <: \oplus\{l_j : T'_j\}_{j\in J}$.

□

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

Proof. The proof is done by induction on the derivation for $T <: T'$.

- Case (S-Bool):
  By (E-Bool) and by rule (Sπ- Refl) we obtain $\mathtt{Bool} \leq \mathtt{Bool}$, which concludes the case.

- Case (S-End):
  It means $\mathsf{end} <: \mathsf{end}$. By (E-End) and rule (S$\pi$-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 $[\![U]\!] \leq [\![U']\!]$. We need to prove that $[\![?T.U]\!] \leq [\![?T'.U']\!]$. By applying (E-Inp) we obtain $[\![?T.U]\!] = \ell_{\mathsf{i}}\,[[\![T]\!], [\![U]\!]]$ and $[\![?T'.U']\!] = \ell_{\mathsf{i}}\,[[\![T']\!], [\![U']\!]]$. 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 $[\![U]\!] \leq [\![U']\!]$. We need to prove that $[\![!T.U]\!] \leq [\![!T'.U']\!]$. By applying (E-Out) we obtain $[\![!T.U]\!] = \ell_{\mathsf{o}}\,[[\![T]\!], [\![\overline{U}]\!]]$ and $[\![!T'.U']\!] = \ell_{\mathsf{o}}\,[[\![T']\!], [\![\overline{U'}]\!]]$. 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 $[\![T_i]\!] \leq [\![T'_j]\!]$ for all $i \in I$. We need to prove that $[\![\&\{l_i : T_i\}_{i\in I}]\!] \leq [\![\&\{l_j : T'_j\}_{j\in J}]\!]$. By applying (E-Branch) we obtain $[\![\&\{l_i : T_i\}_{i\in I}]\!] = \ell_{\mathsf{i}}\,[\langle l_i \_ [\![T_i]\!]\rangle_{i\in I}]$ and $[\![\&\{l_j : T'_j\}_{j\in J}]\!] = \ell_{\mathsf{i}}\,[\langle l_j \_ [\![T'_j]\!]\rangle_{j\in J}]$. 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 $[\![T_i]\!] \leq [\![T'_j]\!]$ for all $j \in J$. We need to prove that $[\![\oplus\{l_i : T_i\}_{i\in I}]\!] \leq [\![\oplus\{l_j : T'_j\}_{j\in J}]\!]$. By applying (E-Select) we obtain $[\![\oplus\{l_i : T_i\}_{i\in I}]\!] = \ell_{\mathsf{o}}\,[\langle l_i \_ [\![\overline{T_i}]\!]\rangle_{i\in I}]$ and $[\![\oplus\{l_j : T'_j\}_{j\in J}]\!] = \ell_{\mathsf{o}}\,[\langle l_j \_ [\![\overline{T'_j}]\!]\rangle_{j\in J}]$. By Lemma 7.2.1 we get $[\![\overline{T'_j}]\!] \leq [\![\overline{T}_j]\!]$ for all $j \in J$. By (Sπ-Variant) and (Sπ-oo) on the induction hypothesis we obtain the result.

□

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

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

where *subtype* is instantiated with $<:$ or $\leq$ depending on the calculus where it is used. Then, we can prove the following results.

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

Proof. The proof is split as follows.

(⇒) Follows the cases in Lemma 6.3.9; we add the case for subsumption which is trivial, since this rule is added on both calculi.

(⇐) Follows the cases in Lemma 6.3.8; we add the case for subsumption which is trivial, since this rule is added on both calculi.

□

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

Proof. The proof is split as follows.

(⇒) Follows the cases in Theorem 6.3.11. Instead of Lemma 6.3.9, we apply Lemma 7.2.4.

(⇐) Follows the cases in Theorem 6.3.10. Instead of Lemma 6.3.8 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 studied subtyping on both session types and standard π-types, which is a simple form of type abstraction.

A more complex form of type abstraction is the *parametric polymorphism* that is already present and well studied in the standard 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 the π-calculus with sessions, we obtain the properties in the polymorphic sessions for free by deriving them from the theory of the polymorphic π-calculus.

In [45] the author studies *bounded polymorphism*. 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 we can obtain bounded polymorphism in the π-calculus with session types by adding bounded polymorphism to the standard π-calculus and by extending our encoding.

8.1 Parametric Polymorphism

We start with parametric polymorphism. We present the syntax of types and term, give the typing rules and the reduction rules. We extend the encoding and by proving its soundness and completeness wrt typing of values and processes, we show our encoding is robust.

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

Figure 8.1: Syntax of parametric polymorphic constructs

8.1.1 Syntax

The syntax of the polymorphic π-calculus with and without sessions is given in Fig. 8.1. 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.

We extend both syntaxes of the π-calculus with and without sessions with the type variable X and the *polymorphic type* $\langle X; T \rangle$.

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

To conclude, we add another typing context 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.

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

This reduction rule holds for both the π-calculus with and without sessions. In order to distinguish them, we use $[\pi]$ in square brackets, which means that π is optional: where π is present, then the rule refers to the standard π-calculus, otherwise it refers to the session π-calculus. 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*.

Rule (R[π]-Unpack) states that process **open** $\langle T; v \rangle$ **as** $(X; x)$ **in** P, with the guard being 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.

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

$$\frac{\Gamma_1; \Delta \vdash v : \langle X; T \rangle \qquad \Gamma_2, x : T; \Delta, X \vdash P}{\Gamma_1 \circ \Gamma_2; \Delta \vdash \textbf{open } v \textbf{ as } (X; x) \textbf{ in } P} \text{ (T-Unpack)}$$

$$\frac{\Gamma_1; \Delta \vdash v : \langle X; T \rangle \qquad \Gamma_2, x : T; \Delta, X \vdash P}{\Gamma_1 \uplus \Gamma_2; \Delta \vdash \textbf{open } v \textbf{ as } (X; x) \textbf{ in } P} \text{ (T}\pi\text{-Unpack)}$$

Figure 8.2: Typing rules for parametric polymorphic constructs

8.1.3 Typing Rules

We are ready now to give 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 typing context introduced in Section 5.4 for the π-calculus with sessions and in Section 4.4 for the standard π-calculus, and Δ collects the polymorphic type variables, needed to type polymorphic constructs.

The typing rules for parametric polymorphism are given 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 that π is optional: where π is present, then the rule refers to the standard π-calculus, otherwise it refers to the session π-calculus.

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 with T' substituting 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 $\langle X; T \rangle$ and process P is well typed in x of type T and Δ augmented with X.

8.1.4 Encoding

The encoding of polymorphic types and terms is an homomorphism and is given in Fig. 8.3. (E-PolyVar) states that the encoding of the type variable X is X itself. (E-PolyType) states that the encoding of a polymorphic session type $\langle X; T \rangle$ is a polymorphic standard π-type $\langle X; [\![T]\!] \rangle$, acting 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, as originally shown in Section 6.2. (E-PolyVal) states that the encoding of a polymorphic

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

Figure 8.3: Encoding of parametric polymorphic constructs

value $\langle T; v\rangle$ added to the session π-calculus is a polymorphic value $\langle [\![T]\!]; [\![v]\!]_f\rangle$ added to the standard π-calculus having type the encoding of T and the value v is renamed according f, resulting in $[\![v]\!]_f$. (E-Unpack) states that the encoding of the unpacking session process **open** v **as** $(X; x)$ **in** P is the unpacking process **open** $[\![v]\!]_f$ **as** $(X; f_x)$ **in** $[\![P]\!]_f$ added to the standard π-calculus where the guard is the encoded value $[\![v]\!]_f$, the polymorphic placeholder x is renamed as f_x and process P is encoded using f, i.e., $[\![P]\!]_f$.

The encoding of typing contexts is given by:

$$
\begin{array}{rcll}
[\![\emptyset]\!]_f & \triangleq & \emptyset & \text{(E-Empty)} \\
[\![\Gamma, x : T]\!]_f & \triangleq & [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] & \text{(E-Gamma)} \\
[\![\Gamma; \Delta]\!]_f & \triangleq & [\![\Gamma]\!]_f; \Delta & \text{(E-Delta)}
\end{array}
$$

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

8.1.5 Properties of the Encoding

In this section we prove the correctness of the encoding wrt typing derivations for polymorphic processes and values and the operational correspondence. We start with the following lemma which relates substitution of types and encoding.

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

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

Proof. It follows immediately from the encoding of types and the standard definition of type substitution. □

To complete Lemma 6.3.8 of soundness and Lemma 6.3.9 of completeness of the encoding wrt typing values, 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 $\Gamma \vdash v : T$ should be now written as $\Gamma; \Delta \vdash v : T$ (with $\Delta = \emptyset$ in absence of polymorphism).

Proof of Lemma 6.3.8 and Lemma 6.3.9 for Parametric Polymorphic Values:

1. If $[\![\Gamma;\Delta]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$ for some renaming function f, then $\Gamma;\Delta \vdash v : T$.

2. If $\Gamma;\Delta \vdash v : T$, then $[\![\Gamma;\Delta]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$ for some renaming function f.

Proof. We split the proof as follows.

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

   We consider only the case for polymorphic values, namely $v = \langle T'; v'\rangle$. By applying (E-PolyVal) we have $[\![\langle T'; v'\rangle]\!]_f = \langle [\![T]\!]; [\![v']\!]_f\rangle$ and assume $[\![\Gamma;\Delta]\!]_f \vdash \langle [\![T]\!]; [\![v']\!]_f\rangle : \langle X; [\![T]\!]\rangle$, which means that the last typing rule applied must have been (Tπ-PolyVal).

   $$\frac{[\![\Gamma;\Delta]\!]_f \vdash [\![v']\!]_f : [\![T]\!][[\![T']\!]/X]}{[\![\Gamma;\Delta]\!]_f \vdash \langle [\![T]\!]; [\![v']\!]_f\rangle : \langle X; [\![T]\!]\rangle}$$

   By induction hypothesis and by Lemma 8.1.1 we obtain $\Gamma \vdash v' : T[T'/X]$. We conclude by applying (T-PolyVal).

2. The proof is done by induction on the derivation for $\Gamma;\Delta \vdash v : T$.

   We consider only the case for (T-PolyVal).

   $$\frac{\Gamma;\Delta \vdash v' : T[T'/X]}{\Gamma;\Delta \vdash \langle T'; v'\rangle : \langle X; T\rangle}$$

   By induction hypothesis and by Lemma 8.1.1, there is f' such that $[\![\Gamma;\Delta]\!]_{f'} \vdash [\![v']\!]_{f'} : [\![T]\!][[\![T']\!]/X]$. By choosing $f = f'$ and by applying rules (Tπ-PolyVal), (E-PolyType) and (E-PolyVal), we obtain the result.

□

To complete Theorem 6.3.10 and Theorem 6.3.11 on the correctness of the encoding wrt typing processes, it suffices to add the case for the unpack process. As with values, adding this case to the proofs of the previous theorems requires modifications in the typing judgements: previous typing judgements of the form $\Gamma \vdash Q$ should be now written as $\Gamma;\Delta \vdash Q$, (with $\Delta = \emptyset$ in absence of polymorphism). These modifications will affect also the statement of operational correspondence given by Theorem 6.3.15.

Proof of Theorem 6.3.10 and Theorem 6.3.11 for Parametric Polymorphic Processes:

1. If $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q, then $\Gamma;\Delta \vdash Q$.
2. If $\Gamma;\Delta \vdash Q$, then $[\![\Gamma;\Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q.

Proof. We split the proof as follows.

1. The proof is done by induction on the structure of session process Q.

   We consider only the case for the unpack process. By (E-Unpack) we have that $[\![\Gamma;\Delta]\!]_f \vdash \mathbf{open}\ [\![v]\!]_f\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f$. This means that the last rule applied must be (Tπ-Unpack):

   $$\frac{[\![\Gamma]\!]_f;\Delta \vdash [\![v]\!]_f : \langle X; [\![T]\!]\rangle \qquad [\![\Gamma]\!]_f, f_x : [\![T]\!];\Delta, X \vdash [\![P]\!]_f}{[\![\Gamma]\!]_f;\Delta \vdash \mathbf{open}\ [\![v]\!]_f\ \mathbf{as}\ (X; f_x)\ \mathbf{in}\ [\![P]\!]_f}$$

   By the soundness of the encoding wrt typing parametric polymorphic values, given previously, we have $\Gamma;\Delta \vdash v : \langle X; T\rangle$. By induction hypothesis $\Gamma, x : T;\Delta, X \vdash P$. Then, by applying (T-Unpack), we conclude the case.

2. The proof is done by induction on the derivation $\Gamma;\Delta \vdash Q$.

   We consider only the case when (T-Unpack) is applied:

   $$\frac{\Gamma_1;\Delta \vdash v : \langle X; T\rangle \qquad \Gamma_2, x : T;\Delta, X \vdash P}{\Gamma_1 \circ \Gamma_2;\Delta \vdash \mathbf{open}\ v\ \mathbf{as}\ (X; x)\ \mathbf{in}\ P}$$

   By the completeness of the encoding wrt typing parametric polymorphic values $[\![\Gamma]\!]_{f'};\Delta \vdash [\![v]\!]_{f'} : \langle X; [\![T]\!]\rangle$, for some function f'. By induction hypothesis $[\![\Gamma, x : T]\!]_{f''};\Delta, X \vdash [\![P]\!]_{f''}$, for some function f''. By (E-Gamma) it means $[\![\Gamma]\!]_{f''} \uplus f''_x : [\![T]\!];\Delta, X \vdash [\![P]\!]_{f''}$ Since $\Gamma_1 \circ \Gamma_2$ is defined, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and define $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$. Let $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D$, such that for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. By Lemma 6.3.6 and since $x \notin \Gamma_2$, by Lemma 6.3.2 we have the following:

   $$[\![\Gamma]\!]_f;\Delta \vdash [\![v]\!]_f : \langle X; [\![T]\!]\rangle \qquad [\![\Gamma]\!]_f, f_x : [\![T]\!];\Delta, X \vdash [\![P]\!]_f$$

   By applying (E-Unpack) and rule (Tπ-Unpack) we obtain the result. □

To complete the operational correspondence given in Theorem 6.3.15, we add the case for polymorphic processes.

Proof of Theorem 6.3.15 for Parametric Polymorphic Processes: Let P be a session process, Γ, Δ session typing contexts, and f a renaming function for P such that $[\![\Gamma; \Delta]\!]_f \vdash [\![P]\!]_f$. Then, the following statements hold.

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

2. If $[\![P]\!]_f \to Q$, then there are $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, and either $f' = f$ or $f' = f, \{x, y \mapsto c\}$ for x, y such that $(\boldsymbol{\nu} xy)$ appears in $\mathcal{E}[P]$.

Proof. We split the proof as follows.

1. We consider only the case when (R-Unpack) is applied.

   If $\textbf{open}\ \langle T; v\rangle\ \textbf{as}\ (X; x)\ \textbf{in}\ P \to P[T/X][v/x]$, then by (E-Unpack), Lemma 8.1.1 and Lemma 6.3.13 and by reduction (Rπ- Unpack) we can obtain $[\![\textbf{open}\ \langle T; v\rangle\ \textbf{as}\ (X; x)\ \textbf{in}\ P]\!]_f \to\!\!\equiv [\![P[T/X][v/x]]\!]_f$.

2. We consider only the case for the unpack process.

   If $[\![\textbf{open}\ \langle T; v\rangle\ \textbf{as}\ (X; x)\ \textbf{in}\ P]\!]_f \to Q$, then by applying (E-Unpack), rule (R-Unpack), the definition of structural congruence and by applying Lemma 8.1.1 and Lemma 6.3.13 we obtain $\textbf{open}\ \langle T; v\rangle\ \textbf{as}\ (X; x)\ \textbf{in}\ P \to P'$ and $Q \equiv [\![P']\!]_f$. Notice that $f' = f$ and $\mathcal{E}[\cdot] = [\cdot]$.

□

8.2 Bounded Polymorphism

We now consider bounded polymorphism, which is studied in [45]. This kind of polymorphism has not been studied in the standard π-calculus; we add it and show how we can derive bounded polymorphism in session π-calculus passing through the standard one. Bounded polymorphism for session types in [45] is added only to the labels of branch and select type and term constructs. In our work, we specify only upper bounds and use only basic types in the bounds. This is a simplification wrt [45] which is sufficient to illustrate how the encoding works.

8.2.1 Syntax

In this section we present both the bounded polymorphic π- calculus with and without sessions. We give the syntax of types and terms, the typing rules and the reduction rules.

$$
\begin{array}{rcll}
T_s & ::= & \ldots & | \; B & \text{(basic type)} \\
p & ::= & \ldots & | \; \oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I} & \text{(bounded polymorphic select)} \\
 & & & | \; \&\{l_i(X_i <: B_i) : T_i\}_{i\in I} & \text{(bounded polymorphic branch)} \\
P_s & ::= & \ldots & | \; x \triangleleft l_j(B).P & \text{(bounded polymorphic selection)} \\
 & & & | \; x \triangleright \{l_i(X_i <: B_i) : P_i\}_{i\in I} & \text{(bounded polymorphic branching)}
\end{array}
$$

Figure 8.4: Syntax of bounded polymorphic session constructs

Syntax of bounded polymorphic constructs in session π-calculus We give in Fig. 8.4 only the new constructs added to the syntax of types and terms.

Type B stands for basic types e.g., integer, boolean, $X, \ldots$ Types produced by T_s, in addition to $\mathsf{lin}p \mid \mathsf{end}$, include basic types B. 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 [45], 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 can 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 report modifications only in selection and branching, 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 as expected, by following the standard definition of type duality for session types.

$$
\begin{array}{rcl}
\overline{\oplus\{l_i(X_i <: B_i) : T\}_{i\in I}} & \triangleq & \&\{l_i(X_i <: B_i) : \overline{T_i}\}_{i\in I} \\
\overline{\&\{l_i(X_i <: B_i) : T\}_{i\in I}} & \triangleq & \oplus\{l_i(X_i <: B_i) : \overline{T_i}\}_{i\in I}
\end{array}
$$

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 for session types: we add type constraints to the labels of variant types and values. We give in Fig. 8.5 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.

$$
\begin{array}{lll}
T_\pi ::= \ \ldots & \mid B & \text{(basic type)} \\
 & \mid \langle l_i(X_i \leq B_i)\_T_i \rangle_{i \in I} & \text{(bounded poly variant)} \\
P_\pi ::= \ \ldots & \mid \textbf{case}\ v\ \textbf{of}\ \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i \in I} & \text{(bounded poly case)} \\
v_\pi ::= \ \ldots & \mid l(B)\_v & \text{(bounded poly variant value)}
\end{array}
$$

Figure 8.5: 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 wrt 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

We now introduce the reduction rules for bounded polymorphic processes for both the π-calculus with and without sessions. We start with session π-calculus.

$$
\begin{array}{ll}
\text{(R-BPolySel)} & (\boldsymbol{\nu} xy)(x \triangleleft l_j(B).P \mid y \triangleright \{l_i(X_i <: B_i) : P_i\}_{i \in I}) \rightarrow \\
 & \qquad (\boldsymbol{\nu} xy)(P \mid P_j[B/X_j]) \quad j \in I \\
\text{(R}\pi\text{-BPolyCase)} & \textbf{case}\ l_j(B)\_v\ \textbf{of}\ \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i \in I} \rightarrow P_j[B/X_j][v/x_j] \quad j \in I
\end{array}
$$

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 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 in addition to the standard substitution of the

$$\frac{\Gamma_1;\Delta \vdash x : \oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I} \quad \Gamma_2, x : T_j[B/X_j];\Delta \vdash P \quad j \in I \quad B <: B_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2;\Delta \vdash x \triangleleft l_j(B).P} \text{ (T-BPolySel)}$$

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

Figure 8.6: Typing rules for bounded polymorphic session constructs

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

We now give the typing rules for both the π-calculus with and without sessions.

Typing rules for bounded polymorphic session π-calculus The typing judgements now 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 context 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 context Γ and a set of type variables Δ.

The new typing rules for the bounded polymorphic branching and selection are given in Fig. 8.6. Rule (T-BPolySel) states that the selection process, where label l_j together with the basic type B are 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 The 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 context 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 context Γ and a set of type variables Δ.

The new typing rules for the standard π- calculus are presented in Fig. 8.7. Rule (Tπ- BPolyLVal) states that the bounded 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

$$\frac{\Gamma;\Delta \vdash v : T_j[B/X_j] \quad j \in I \qquad B \leq B_i \quad \forall i \in I}{\Gamma;\Delta \vdash l_j(B)\_v : \langle l_i(X_i \leq B_i)\_T_i\rangle_{i\in I}} \text{ (T}\pi\text{-BPolyLVal)}$$

$$\frac{\Gamma_1;\Delta \vdash v : \langle l_i(X_i \leq B_i)\_T_i\rangle_{i\in I} \qquad \Gamma_2, x_i : T_i;\Delta, X_i \leq B_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \uplus \Gamma_2;\Delta \vdash \textbf{case}\ v\ \textbf{of}\ \{l_i(X_i \leq B_i)\_x_i \triangleright P\}_{i\in I}} \text{ (T}\pi\text{-BPolyCase)}$$

Figure 8.7: Typing rules for bounded polymorphic π-constructs

$$\begin{array}{rcll} [\![B]\!] & \triangleq & B & \text{(E-BPolyB)} \\ [\![\oplus\{l_i(X_i <: B_i) : T_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathsf{o}}\,[\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i\in I}] & \text{(E-BPolySel)} \\ [\![\&\{l_i(X_i <: B_i) : T_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathsf{i}}\,[\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}] & \text{(E-BPolyBrch)} \end{array}$$

Figure 8.8: Encoding of bounded polymorphic types

all $i \in I$ 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 typing context with the type assumption $x_i : T_i$ and the constraint $X_i \leq B_i$.

8.2.4 Encoding

The encoding of bounded polymorphic types is defined in Fig. 8.8. (E-BPolyB) states that the encoding is the identity function on a basic type, 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. (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 and the types in the branches of the variant type are $[\![\overline{T_i}]\!]$ for all $i \in I$. (E-BPolyBrch) states the dual of the previous one: the bounded polymorphic branch is encoded as a linear input channel type. The subtyping constraints are the same and the types in the branches of the variant type are $[\![T_i]\!]$ for all $i \in I$.

The encoding of bounded polymorphic terms is defined in Fig. 8.9. The difference wrt (E-Selection) and (E-Branching) is the annotation of labels with types. (E-BPolySelection) states that the bounded polymorphic selection is interpreted

$$\begin{array}{rcl}
\text{(E-BPolySelection)} & & \\
[\![x \triangleleft l_j(B).P]\!]_f & \triangleq & (\nu c) f_x!\langle l_j(B)_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \\
 & & \\
\text{(E-BPolyBranching)} & & \\
[\![x \triangleright \{l_i(X_i <: B_i) : P_i\}_{i\in I}]\!]_f & \triangleq & f_x?(y).\ \mathbf{case}\ y\ \mathbf{of}\ \{l_i(X_i \leq B_i)_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}
\end{array}$$

Figure 8.9: Encoding of bounded polymorphic terms

as an output with subject the renamed variable x and object a bounded polymorphic variant value, where the selected label and the basic type are the same as the original ones and the value carried by the variant value is a freshly created channel c, used in the rest of the communication. The continuation process P is encoded in f updated with x renamed as c. (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 typing contexts is defined as follows:

$$[\![\Gamma; \Delta]\!]_f \quad \triangleq \quad [\![\Gamma]\!]_f; \Delta$$

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 of bounded polymorphic constructs wrt typing and reduction. This means that by using the encoding and bounded polymorphism in the standard π-calculus, we can derive bounded polymorphism in the π-calculus with session types.

To complete Lemma 6.3.8 of soundness and Lemma 6.3.9 of completeness of the encoding wrt typing values, it suffices to add the cases for bounded polymorphic variables. However, adding this case requires modification in the typing judgements: previous typing judgements of the form $\Gamma \vdash v : T$ should be now written as $\Gamma; \Delta \vdash v : T$, (with $\Delta = \emptyset$ in absence of polymorphism).

The cases for bounded polymorphic variables for Lemma 6.3.8 and Lemma 6.3.9, follow immediately by (E-BPolySel) and (E-BPolyBrch) and by rules (T-Var) and (Tπ-Var).

To complete Theorem 6.3.10 and Theorem 6.3.11 on the correctness of the encoding wrt typing processes, it suffices to add the cases for bounded branching and selection. Adding these cases to the proofs of the previous theorems requires modification in the typing judgements: previous typing judgements of the form

$\Gamma \vdash Q$ should be now written as $\Gamma; \Delta \vdash Q$, (with $\Delta = \emptyset$ in absence of polymorphism). These modifications will also influence the operational correspondence, as we will show in the following.

Proof of Theorem 6.3.10 for Bounded Polymorphic Processes:
If $[\![\Gamma; \Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q, then $\Gamma; \Delta \vdash Q$.

Proof. The proof is done by induction on the structure of session process Q.
We consider only the new cases for bounded polymorphic processes.

- Case $Q = x \triangleleft l_j(B).P$:
By (E-BPolySelection) we have $[\![x \triangleleft l_j(B).P]\!]_f = (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$ and assume $[\![\Gamma]\!]_f; \Delta \vdash (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$. Since c is a restricted channel in the encoding of Q, then either rule (Tπ- Res1) or (Tπ- Res2) must be applied. We consider only the case for (Tπ- Res1), as the one for (Tπ- Res2) is symmetrical. Then, by (Tπ- Res1) and (Tπ- Out) we have the following derivation:

$$\dfrac{\dfrac{\begin{array}{c}(\text{T}\pi\text{-Res1})\\(\text{T}\pi\text{-Out})\\ \Gamma_1^\pi; \Delta \vdash f_x : \ell_{\mathsf{o}}\,[\langle l_i(X_i \le B_i)\_T_i^\pi\rangle_{i\in I}] \qquad \Gamma_2^\pi, c : \overline{T_j^\pi}[B/X_j]; \Delta \vdash [\![P]\!]_{f,\{x\mapsto c\}} \\ c : T_j^\pi[B/X_j]; \Delta \vdash l_j(B)\_c : \langle l_i(X_i \le B_i)\_T_i^\pi\rangle_{i\in I}\end{array}}{[\![\Gamma]\!]_f, c : \ell_\sharp\,[W][B/X_j]; \Delta \vdash f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}}{[\![\Gamma]\!]_f; \Delta \vdash (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

and $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.5 $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$. By applying (T$\pi$-Var) and (T$\pi$-BPolyLval) for some $j \in I$, we have the following derivation:

$$\dfrac{\dfrac{(\text{T}\pi\text{-Var})}{c : T_j^\pi[B/X_j]; \Delta \vdash c : T_j^\pi[B/X_j]} \quad B \le B_i \quad \forall i \in I}{c : T_j^\pi[B/X_j]; \Delta \vdash l_j(B)\_c : \langle l_i(X_i \le B_i)\_T_i^\pi\rangle_{i\in I}}\ (\text{T}\pi\text{-BPolyLVal})$$

Notice that c is of type $\ell_\sharp\ [W][B/X_j]$, which is $T_j^\pi[B/X_j] \uplus \overline{T_j^\pi}[B/X_j]$ and one capability of c is sent along $l_j(B)\_c$ whether the other one is used in the continuation $[\![P]\!]_{f,\{x\mapsto c\}}$. In the case where (T$\pi$- Res2) is applied, c is of type $\emptyset[] \uplus \emptyset[]$. By the correctness of the encoding wrt typing bounded polymorphic values, as shown earlier in Section 8.2.5, we have $\Gamma_1; \Delta \vdash x : \oplus\{l_i(X_i \le B_i) : T_i\}_{i\in I}$ which by (E-BPolySel) means

$[\![\oplus\{l_i(X_i \leq B_i) : T_i\}_{i\in I}]\!] = \ell_{\mathsf{o}}\,[\langle l_i(X_i \leq B_i)\_[\![T_i^\pi]\!]\rangle_{i\in I}]$ and $T_i^\pi = [\![\overline{T_i}]\!]$ for all $i \in I$. By induction hypothesis $\Gamma_2, x : T_j[B/X_j]; \Delta \vdash P$. By Theorem 7.2.2 we obtain $B <: B_i$ for all $i \in I$. By applying typing rule (T-BPolySel) we obtain $\Gamma_1 \circ \Gamma_2; \Delta \vdash x \triangleleft l_j(B).P$, as required.

- Case $Q = x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}$:
  By (E-BPolyBranching) we have

  $$[\![x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}]\!]_f = f_x?(y).\ \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}$$

  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) means that:

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

  and $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.5 we have that $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ such that $\Gamma = \Gamma_1 \circ \Gamma_2$.

  By (Tπ-BPolyCase) and (Tπ-Var) we have the following derivation:

  $$\dfrac{\dfrac{}{y : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}; \Delta \vdash y : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}}\ (\text{T}\pi\text{-Var}) \quad \Gamma_2^\pi, c : T_i^\pi; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f,\{x\mapsto c\}} \quad \forall i \in I}{\Gamma_2^\pi, y : \langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}; \Delta \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}\ (\text{T}\pi\text{-BPolyCase})$$

  By the correctness of the encoding wrt typing bounded polymorphic values, we have that $\Gamma_1; \Delta \vdash x : \&\{l_i(X_i \leq B_i) : T_i\}_{i\in I}$ where by (E-BPolyBrch) $[\![\&\{l_i(X_i \leq B_i) : T_i\}_{i\in I}]\!]_f = \ell_{\mathsf{i}}\,[\langle l_i(X_i \leq B_i)\_T_i^\pi\rangle_{i\in I}]$ and $[\![T_i]\!] = T_i^\pi$ for all $i \in I$. By the premise of (Tπ-Inp) $[\![\Gamma_2]\!]_f, c : [\![T_i]\!]; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}$, then by induction hypothesis we have $\Gamma_2, x : T_i; \Delta, X_i <: B_i \vdash P_i$ for all $i \in I$. By (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 required. □

Proof of Theorem 6.3.11 for Bounded Polymorphic Processes:
If $\Gamma; \Delta \vdash Q$, then $[\![\Gamma; \Delta]\!]_f \vdash [\![Q]\!]_f$ for some renaming function f for Q.

Proof. The proof is done by induction on the derivation $\Gamma; \Delta \vdash Q$. We examine only the cases where either (T-BPolySel) or (T-BPolyBrch) is applied.

- Case (T-BPolySel):

$$\frac{\Gamma_1; \Delta \vdash x : \oplus\{l_i(X_i \leq B_i) : T_i\}_{i \in I} \quad \Gamma_2, x : T_j[B/X_j]; \Delta \vdash P \quad j \in I \quad B <: B_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2; \Delta \vdash x \triangleleft l_j(B).P} \text{ (T-BPolySel)}$$

By the correctness of the encoding wrt typing bounded polymorphic values, as shown in Section 8.2.5, $[\![\Gamma_1]\!]_{f'}; \Delta \vdash f'_x : \ell_\circ\ [\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}]$ for some function f'. By induction hypothesis, (E-Gamma) and Lemma 8.1.1 we have that $[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![T_j]\!][B/X_j]; \Delta \vdash [\![P]\!]_{f''}$ for $j \in I$ for some function f''. By Theorem 7.2.3 $B \leq B_i$ for all $i \in I$. Since $\Gamma_1 \circ \Gamma_2$ is defined, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and let $f'_D = f' \setminus \bigcup_{d \in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d \in D}\{d \mapsto f''(d)\}$. Now, suppose $f''(x) = c$, then define $f = \bigcup_{d \in D}\{d \mapsto d'\} \cup f'_D \cup f''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. Hence, by Lemma 6.3.6 we can rewrite the induction hypothesis as follows:

$$[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_\circ\ [\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}]$$

and for $j \in I$

$$[\![\Gamma_2]\!]_f \uplus c : [\![T_j]\!][B/X_j]; \Delta \vdash [\![P]\!]_{f,\{x \mapsto c\}}$$

Since $x \notin \mathsf{dom}(\Gamma_2)$, then $[\![\Gamma_2, x : T_j[B/X_j]]\!]_{f,\{x \mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_2]\!]_f \uplus c : [\![T_j]\!][B/X_j]$. By applying (T$\pi$-Var) in order to derive $c : [\![\overline{T_j}]\!]$, rule (T$\pi$-BPolyLVal) and by Theorem 7.2.3 on completeness of subtyping wrt to encoding we have the following:

$$\frac{\dfrac{}{c : [\![\overline{T_j}]\!][B/X_j]; \Delta \vdash c : [\![\overline{T_j}]\!][B/X_j]} \text{ (T}\pi\text{-Var)} \quad j \in I \quad B \leq B_i \quad \forall i \in I}{c : [\![\overline{T_j}]\!][B/X_j]; \Delta \vdash l_j(B)\_c : \langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}} \text{ (T}\pi\text{-BPolyLVal)}$$

Suppose that $T_j \neq \mathsf{end}$ and hence $\overline{T_j} \neq \mathsf{end}$. By applying rule (T$\pi$-Out) we have the following derivation:

$$\frac{[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_\circ\ [\langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I}] \quad c : [\![\overline{T_j}]\!][B/X_j]; \Delta \vdash l_j(B)\_c : \langle l_i(X_i \leq B_i)\_[\![\overline{T_i}]\!]\rangle_{i \in I} \quad [\![\Gamma_2]\!]_f \uplus c : [\![T_j]\!][B/X_j]; \Delta \vdash [\![P]\!]_{f,\{x \mapsto c\}} \ j \in I}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus c : \ell_\sharp[W][B/X_j]; \Delta \vdash f_x!\langle l_j(B)\_c\rangle.[\![P]\!]_{f,\{x \mapsto c\}}}$$

In the above derivation we have that $c : [\![\overline{T_j}]\!][B/X_j]$ and $c : [\![T_j]\!][B/X_j]$ are combine and Lemma 6.3.7 we 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 Lemma 6.3.2 and (Tπ-Res1):

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

The case where $T_j = \overline{T_j} = \mathsf{end}$, which yields $c : \emptyset[]$, is symmetrical and is obtained by using (Tπ-Res2) instead of (Tπ-Res1).

- Case (T-BPolyBrch):

$$\frac{\Gamma_1; \Delta \vdash x : \&\{l_i(X_i \leq B_i) : T_i\}_{i\in I} \qquad \Gamma_2, x : T_i; \Delta, X_i <: B_i \vdash P_i \quad \forall i \in I}{\Gamma_1 \circ \Gamma_2; \Delta \vdash x \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}} \text{(T-BPolyBrch)}$$

By the correctness of the encoding wrt typing bounded polymorphic values, we have $[\![\Gamma_1]\!]_{f'}; \Delta \vdash f'_x : \ell_{\mathsf{i}}\ [\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}]$ for some function f'. By induction hypothesis, by (E-Gamma) and Theorem 7.2.3 we have that $[\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![T_i]\!]; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f''}$ for all $i \in I$ and for some function f''. Since $\Gamma_1 \circ \Gamma_2$ is defined, it means that for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$. Then, we define $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\}$. Suppose $f''(x) = c$. We let $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. We now have:

$$[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_{\mathsf{i}}\ [\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}]$$

and for all $i \in I$,

$$[\![\Gamma_2]\!]_f \uplus c : [\![T_i]\!]; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f,\{x\mapsto c\}}$$

Since $x \notin \mathsf{dom}(\Gamma_2)$, then $[\![\Gamma_2, x : T_j]\!]_{f,\{x\mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_2]\!]_f \uplus c : [\![T_j]\!]$. By (T$\pi$-Var) in order to derive $y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}$, and (T$\pi$-BPolyCase) and Lemma 6.3.2 we have the following derivation:

$$\frac{\dfrac{}{y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}; \Delta \vdash y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}} \text{(T}\pi\text{-Var)} \qquad [\![\Gamma_2]\!]_f, c : [\![T_i]\!]; \Delta, X_i \leq B_i \vdash [\![P_i]\!]_{f,\{x\mapsto c\}} \quad \forall i \in I}{[\![\Gamma_2]\!]_f, y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}; \Delta \vdash \mathbf{case}\ y\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}} \text{(T}\pi\text{-BPolyCase)}$$

Then, by applying (Tπ-Inp) we conclude as follows:

$$\frac{[\![\Gamma_1]\!]_f; \Delta \vdash f_x : \ell_i\ [\langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}] \quad [\![\Gamma_2]\!]_f, y : \langle l_i(X_i \leq B_i)\_[\![T_i]\!]\rangle_{i\in I}; \Delta \vdash \mathbf{case}\, y\, \mathbf{of}\, \{l_i\_c(X_i \leq B_i) \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}{[\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; \Delta \vdash f_x?(y).\, \mathbf{case}\, y\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P_i]\!]_{f,\{x\mapsto c\}}\}_{i\in I}}$$

□

In the following, we prove the operational correspondence in the case of bounded polymorphic processes.

Proof of Theorem 6.3.15 for Bounded Polymorphic Processes: Let P be a session process, Γ, Δ session typing contexts, and f a renaming function for P such that $[\![\Gamma; \Delta]\!]_f \vdash [\![P]\!]_f$. Then, the following statements hold.

1. If $P \to P'$, then $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$.
2. If $[\![P]\!]_f \to Q$, then there are $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, and either $f' = f$ or $f' = f, \{x, y \mapsto c\}$ for x, y such that (νxy) appears in $\mathcal{E}[P]$.

Proof. Since $[\![\Gamma; \Delta]\!]_f \vdash [\![P]\!]_f$, then by Theorem 6.3.10 for bounded polymorphic processes, given earlier in this section, it is the case that $\Gamma; \Delta \vdash P$. We consider both cases in the following.

1. We consider only the case where rule (R-BPolySel) is applied.

   $$P \triangleq (\nu xy)(x \triangleleft l_j(B).Q \mid y \triangleright \{l_i(X_i \leq B_i) : P_i\}_{i\in I}) \to (\nu xy)(Q \mid P_j[B/X_j]) \triangleq P' \ j \in I$$

   By the encoding of bounded polymorphic processes we have

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

Notice that since P is a well-typed session process, it means that for all $i \in I$, $x \notin \mathsf{fv}(P_i)$ and $y \notin \mathsf{fv}(Q)$. Then, function $f, \{x, y \mapsto c, x \mapsto c'\}$ and function $f, \{x, y \mapsto c, y \mapsto c'\}$ can both be subsumed by $f, \{x, y \mapsto c'\}$. We can rewrite the above as:

$$(\nu c')([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}})$$

On the other hand we have:

$$\begin{array}{rcl} [\![P']\!]_f & = & [\![(\nu xy)(Q \mid P_j[B/X_j])]\!]_f \\ & = & (\nu c')([\![Q]\!]_{f,\{x,y\mapsto c'\}} \mid [\![P_j]\!]_{f,\{x,y\mapsto c'\}}[B/X_j]) \end{array}$$

We use Lemma 8.1.1 to obtain $[\![P_j]\!]_{f,\{x,y\mapsto c'\}}[B/X_j]$. The above implies:

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

2. Case $P = P_1 \mid P_2 = x \triangleleft l_j(B).P'_1 \mid y \triangleright \{l_i(X_i \leq B_i) : P''_i\}_{i\in I}$.

   By (E-Composition), (E-BPolySelection) and (E-BPolyBranching), we have that:

$$\begin{array}{rcl} [\![P_1]\!]_f \mid [\![P_2]\!]_f & & \\ = & (\nu c) f_x!\langle l_j(B)\_c\rangle.[\![P'_1]\!]_{f,\{x\mapsto c\}} \mid & \\ & \quad f_y?(z).\ \mathbf{case}\ z\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I} & \\ \rightarrow & [\![P'_1]\!]_{f,\{x\mapsto c\}} \mid \mathbf{case}\ l_j(B)\_c\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright [\![P''_i]\!]_{f,\{y\mapsto c\}}\}_{i\in I}[B/X_j] & \\ = & [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid \mathbf{case}\ l_j(B)\_c\ \mathbf{of}\ \{l_i(X_i \leq B_i)\_c \triangleright [\![P''_i]\!]_{f,\{x,y\mapsto c\}}\}_{i\in I}[B/X_j] & \\ = & Q & \end{array}$$

   The assumption $[\![P_1]\!]_f \mid [\![P_2]\!]_f \rightarrow Q$ implies that $f_x = f_y$. Since by assumption $\Gamma; \Delta \vdash P_1 \mid P_2$, it means that $x \notin \mathsf{fv}(P_2)$ and $y \notin \mathsf{fv}(P_1)$. Hence, in the last line before Q above we used function $f, \{x, y \mapsto c\}$ to subsume both $f, \{x \mapsto c\}$ and $f, \{y \mapsto c\}$. We need to show that there are $P', \mathcal{E}[\cdot]$ such that $\mathcal{E}[P_1 \mid P_2] \rightarrow \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, where either $f' = f$ or $f' = f, \{z, w \mapsto d\}$ for z, w such that (νzw) appears in $\mathcal{E}[P]$. Choose $\mathcal{E}[\cdot] = (\nu xy)[\cdot]$. Then, by rule (R-BPolySel) on $\mathcal{E}[P_1 \mid P_2]$ we have that:

$$\begin{array}{rcl} \mathcal{E}[P_1 \mid P_2] & = & (\nu xy)\left(x \triangleleft l_j(B).P'_1 \mid y \triangleright \{l_i(X_i \leq B_i) : P''_i\}_{i\in I}\right) \\ & \rightarrow & (\nu xy)(P'_1 \mid P''_j[B/X_j]) \\ & = & \mathcal{E}[P'] \end{array}$$

   Choose $P' = P'_1 \mid P''_j[B/X_j]$ and $f' = f, \{x, y \mapsto c\}$. By the encoding of P' we have:

$$[\![P']\!]_{f'} = [\![P'_1 \mid P''_j[B/X_j]]\!]_{f'} = [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid [\![P''_j]\!]_{f,\{x,y\mapsto c\}}[B/X_j]$$

It remains to show that $Q \hookrightarrow [\![P']\!]_{f,\{x,y\mapsto c\}}$. By rules (R$\pi$-BPolyCase) and (R$\pi$-Par) and Lemma 8.1.1 we have:

$$\begin{aligned} Q &= [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid \mathbf{case}\, l_j(B)\_c\, \mathbf{of}\, \{l_i(X_i \leq B_i)\_c \triangleright [\![P''_i]\!]_{f,\{x,y\mapsto c\}}\}_{i\in I}[B/X_j] \\ &\to [\![P'_1]\!]_{f,\{x,y\mapsto c\}} \mid [\![P''_j]\!]_{f,\{x,y\mapsto c\}}[B/X_j] \\ &= [\![P']\!]_{f,\{x,y\mapsto c\}} \end{aligned}$$

This concludes the proof.

□

Chapter 9

Higher-Order Communication

Higher-Order π-calculus (HOπ) models mobility of processes that can be sent and received and can be run locally [101]. Higher-order communication has also been studied in the π- calculus with sessions [89]. Following the same line as in the previous chapters, we want to use standard HOπ to obtain higher-order communication in the π-calculus with sessions by exploiting the encoding.

9.1 Syntax

σ	::=	...	T	(general type)
			$\diamond$	(process type)
T	::=	...	$\texttt{Unit}$	(unit type)
			$T \to \sigma$	(functional type)
			$T \overset{1}{\to} \sigma$	(linear functional type)
P	::=	...	PQ	(application)
			v	(values)
v	::=	...	$\lambda x : T.P$	(abstraction)
			$\star$	(unit value)

Figure 9.1: Syntax of higher-order constructs

We present in Fig. 9.1 the modifications done to the syntax of types and terms for both the π-calculus with and without sessions. We will distinguish the session constructs from the standard π-calculus ones 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.

$$\text{(R}[\pi]\text{-Beta)} \qquad (\lambda x : T.P)v \to P[v/x]$$

$$\text{(R}[\pi]\text{-ApplLeft)} \qquad \frac{P \to P'}{PQ \to P'Q}$$

$$\text{(R}[\pi]\text{-ApplRight)} \qquad \frac{P \to P'}{vP \to vP'}$$

Figure 9.2: Semantics of higher-order constructs

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 a functional term that can be used without any restriction and the linear functional type $T \xrightarrow{1} \sigma$, 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 guarantee in 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*, used 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. A value v 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 $\star$ having `Unit` type.

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. We give them in Fig. 9.2. We will distinguish the reduction rules for the π-calculus with sessions from the ones for the standard π-calculus by the presence of $[\pi]$ in the rule name.

Rule (R$[\pi]$-Beta) states that the application of an abstraction $\lambda x : T.P$ to a value v reduces to P where v substitutes x. Rules on context closure, given by (R$[\pi]$-ApplLeft) and (R$[\pi]$-ApplRight), state that the application process reduces if one of its subprocesses 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 $\Phi; \Gamma; \mathcal{S} \vdash P : \diamond$.

9.3.1 HOπ Session Typing Rules

The session typing contexts are defined as follows:

$$
\begin{array}{lll}
\Phi ::= & \emptyset \mid \Phi, x : \texttt{Bool} \mid \Phi, x : \texttt{Unit} & \\
& \Phi, x : T \to \sigma \mid \Phi, x : T \overset{1}{\to} \sigma & \text{(general typing context)} \\
\Gamma ::= & \emptyset \mid \Gamma, x : \mathsf{lin} p \mid \Gamma, x : \texttt{end} & \text{(session typing context)} \\
\mathcal{S} ::= & \emptyset \mid \mathcal{S} \cup \{x\} & \text{(linear functional variables)}
\end{array}
$$

where 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 $\mathsf{dom}(\Phi) \cap \mathsf{dom}(\Gamma) = \emptyset$. The predicates lin and un are defined in Section 5.3. 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 Fig. 9.3. Rule (T-HoSess) states that a variable x has session type T, if this is assumed in Γ. Rule (T-HoVar) states that a variable has type T different from a session type and from a linear functional type, if this is assumed in Φ. 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-HoFun) states that a variable is of a linear functional type, if this is assumed in Φ. Rule (T-HoUnit) is similar to (T-HoBool). There are two typing rules for abstractions, depending on the type of the binder x 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 and x has a value type. In case x is a linear functional variable, then it appears in $\mathcal{S}$. Rule (T-HoAbs2) is similar to the previous one, but in this case x has a session type. Rule (T-HoSub) is a subsumption typing rule. It states that a functional type can be lifted to a linear functional type. Rule (T-HoApp) states that the application of process P to Q has type σ if P is of a linear functional type $T \overset{1}{\to} \sigma$ and Q is of type T. In case the type of Q is a standard functional type, then Q does not have any session channel, enforced by condition $\mathsf{un}(\Gamma_2)$, or any linear functional variables, enforced by condition $\mathcal{S}_2 = \emptyset$, otherwise this would violate linearity.

$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma, x:T;\emptyset \vdash x:T}\ (\text{T-HoSess}) \qquad \frac{T \neq T' \overset{1}{\to} \sigma \qquad \mathsf{un}(\Gamma)}{\Phi, x:T;\Gamma;\emptyset \vdash x:T}\ (\text{T-HoVar})$$

$$\frac{\mathsf{un}(\Gamma) \quad v = \mathtt{true} \mid \mathtt{false}}{\Phi;\Gamma;\emptyset \vdash v:\mathtt{Bool}}\ (\text{T-HoBool}) \qquad \frac{\mathsf{un}(\Gamma)}{\Phi, x:T \overset{1}{\to} \sigma;\Gamma;\{x\} \vdash x:T \overset{1}{\to} \sigma}\ (\text{T-HoFun})$$

$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset \vdash \star:\mathtt{Unit}}\ (\text{T-HoUnit}) \qquad \frac{\Phi, x:T;\Gamma;\mathcal{S} \vdash P:\sigma \quad \text{if } T = T' \overset{1}{\to} \sigma \text{ then } x \in \mathcal{S}}{\Phi;\Gamma;\mathcal{S} - \{x\} \vdash \lambda x:T.P : T \to \sigma}\ (\text{T-HoAbs1})$$

$$\frac{\Phi;\Gamma, x:T;\mathcal{S} \vdash P:\sigma}{\Phi;\Gamma;\mathcal{S} \vdash \lambda x:T.P : T \to \sigma}\ (\text{T-HoAbs2}) \qquad \frac{\Phi;\Gamma;\mathcal{S} \vdash P : T \to \sigma}{\Phi;\Gamma;\mathcal{S} \vdash P : T \overset{1}{\to} \sigma}\ (\text{T-HoSub})$$

$$\frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash P : T \overset{1}{\to} \sigma \qquad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q : T \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash PQ : \sigma}\ (\text{T-HoApp})$$

Figure 9.3: Typing rules for the HOπ with sessions: values

The typing rules for processes are given in Fig. 9.4. Rule (T-Inact) states that the terminated process is well typed in any typing context without assumptions on session types or linear functional types. Rule (T-HoPar) is straightforward, it uses context split and union of sets of linear functional variables. Rules (T-HoRes) and (T-HoIf) are straightforward. There are two typing rules for the input process, depending on the type of the placeholder of the input prefix. Rule (T-HoInp1) is similar to (T-Inp) where the type of the placeholder y is a session type. Rule (T-HoInp2) states the well-typedness of the input process where y is of a value type. In case y is a linear functional variable, then it occurs in the set $\mathcal{S}$. 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 when a session channel is sent (and in that case it can be read as (T-Out)), or when a 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 typing contexts, which are split in three parts.

$$\text{(T-HoInact)}\quad \frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset \vdash \mathbf{0}} \qquad \text{(T-HoPar)}\quad \frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash P \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash P \mid Q}$$

$$\text{(T-HoRes)}\quad \frac{\Phi;\Gamma, x : T, y : \overline{T};\mathcal{S} \vdash P}{\Phi;\Gamma;\mathcal{S} \vdash (\nu xy)P} \qquad \text{(T-HoInp1)}\quad \frac{\Phi;\Gamma_1;\emptyset \vdash x : ?T.U \quad \Phi;\Gamma_2, x : U, y : T;\mathcal{S} \vdash P}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} \vdash x?(y).P}$$

$$\text{(T-HoIf)}\quad \frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash v : \mathtt{Bool} \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash P \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q}$$

$$\text{(T-HoInp2)}\quad \frac{\Phi;\Gamma_1;\emptyset \vdash x : ?T.U \quad \Phi, y : T;\Gamma_2, x : U;\mathcal{S} \vdash P \quad \text{if } T = T' \xrightarrow{1} \sigma \text{ then } y \in \mathcal{S}}{\Phi;\Gamma_1 \circ \Gamma_2;\mathcal{S} - \{y\} \vdash x?(y).P}$$

$$\text{(T-HoOut)}\quad \frac{\Phi;\Gamma_1;\emptyset \vdash x : !T.U \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash v : T \quad \Phi;\Gamma_3, x : U;\mathcal{S}_3 \vdash P \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi;\Gamma_1 \circ \Gamma_2 \circ \Gamma_3;\mathcal{S}_2 \cup \mathcal{S}_3 \vdash x!\langle v \rangle.P}$$

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

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

Figure 9.4: Typing rules for the HOπ with sessions: processes

9.3.2 HOπ Typing Rules

The typing contexts for the standard HOπ are defined as follows:

$$
\begin{array}{lll}
\Phi ::= & \emptyset \mid \Phi, x : \texttt{Bool} \mid \Phi, x : \texttt{Unit} & \\
& \Phi, x : \langle l_i\_T_i \rangle_{i \in I} \mid \Phi, x : T \to \sigma & \\
& \Phi, x : T \overset{1}{\to} \sigma & \text{(general typing context)} \\
\Gamma ::= & \emptyset \mid \Gamma, x : \tau & \text{(channel typing context)} \\
\mathcal{S} ::= & \emptyset \mid \mathcal{S} \cup \{x\} & \text{(linear functional variables)}
\end{array}
$$

where Φ associates value types, except channel 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. 4.6. As for sessions, 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 $\emptyset[]$.

The typing rules for 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. We comment only on the typing rules that are new or different wrt the ones previously presented. Rule (Tπ- HoLVal) is the same as (Tπ- LVal), the only difference is the split of the typing contexts in three parts. There are two typing rules for restriction, as in the case of first-order standard π- calculus. Rule (Tπ- HoCase) is similar to (Tπ- Case). In addition, it uses a set of linear functional variables that comes from the union of linear variables in the guard v and in P_i for all $i \in I$. In the same way as for the branching process, the set of linear variables for every P_i is the same set $\mathcal{S}_2$, considering that only one of such processes will be executed.

9.4 Encoding

We start with the encoding of typing contexts, defined in the following:

$$
\begin{array}{rcll}
[\![\emptyset]\!]_f & \triangleq & \emptyset & \text{(E-Empty)} \\
[\![\Phi; \Gamma; \mathcal{S}]\!]_f & \triangleq & [\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f & \text{(E-HOContext)} \\
[\![\Gamma, x : T]\!]_f & \triangleq & [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] & \text{(E-Gamma)} \\
[\![\Phi, x : T]\!] & \triangleq & [\![\Phi]\!]_f, f_x : [\![T]\!] & \text{(E-Phi)}
\end{array}
$$

The encoding of Γ is the same as in Fig. 6.3. The encoding on the typing contexts Φ and $\mathcal{S}$ is an homomorphism.

$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma, x : T;\emptyset \vdash x : T}\ (\text{T}\pi\text{-HoSess})$$

$$\frac{T \neq T' \xrightarrow{1} \sigma \qquad \mathsf{un}(\Gamma)}{\Phi, x : T;\Gamma;\emptyset \vdash x : T}\ (\text{T}\pi\text{-HoVar})$$

$$\frac{\mathsf{un}(\Gamma) \quad v = \texttt{true} \,/\, \texttt{false}}{\Phi;\Gamma;\emptyset \vdash v : \texttt{Bool}}\ (\text{T}\pi\text{-HoBool})$$

$$\frac{\mathsf{un}(\Gamma)}{\Phi, x : T \xrightarrow{1} \sigma;\Gamma;\{x\} \vdash x : T \xrightarrow{1} \sigma}\ (\text{T}\pi\text{-HoFun})$$

$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset \vdash \star : \texttt{Unit}}\ (\text{T}\pi\text{-HoUnit})$$

$$\frac{\Phi, x : T;\Gamma;\mathcal{S} \vdash P : \sigma \quad \text{if } T = T' \xrightarrow{1} \sigma \text{ then } x \in \mathcal{S}}{\Phi;\Gamma;\mathcal{S} - \{x\} \vdash \lambda x : T.P : T \to \sigma}\ (\text{T}\pi\text{-HoAbs1})$$

$$\frac{\Phi;\Gamma, x : T;\mathcal{S} \vdash P : \sigma}{\Phi;\Gamma;\mathcal{S} \vdash \lambda x : T.P : T \to \sigma}\ (\text{T}\pi\text{-HoAbs2})$$

$$\frac{\Phi;\Gamma;\mathcal{S} \vdash P : T \to \sigma}{\Phi;\Gamma;\mathcal{S} \vdash P : T \xrightarrow{1} \sigma}\ (\text{T}\pi\text{-HoSub})$$

$$\frac{\Phi;\Gamma;\mathcal{S} \vdash v : T_j \quad j \in I}{\Phi;\Gamma;\mathcal{S} \vdash l_j\_v : \langle l_i\_T_i \rangle_{i \in I}}\ (\text{T}\pi\text{-HoLVal})$$

$$\frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash P : T \xrightarrow{1} \sigma \qquad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q : T \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash PQ : \sigma}\ (\text{T}\pi\text{-HoApp})$$

Figure 9.5: Typing rules for the standard HOπ: values

(Tπ-HoInact)
$$\frac{\mathsf{un}(\Gamma)}{\Phi;\Gamma;\emptyset \vdash \mathbf{0}}$$

(Tπ-HoPar)
$$\frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash P \quad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash P \mid Q}$$

(Tπ-HoRes1)
$$\frac{\Phi;\Gamma, x : \ell_\alpha[\widetilde{T}];\mathcal{S} \vdash P}{\Phi;\Gamma;\mathcal{S} \vdash (\nu x)P}$$

(Tπ-HoRes2)
$$\frac{\Phi;\Gamma;\mathcal{S} \vdash P}{\Phi;\Gamma;\mathcal{S} \vdash (\nu x)P}$$

(Tπ-HoIf)
$$\frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash v : \mathtt{Bool} \qquad \Phi;\Gamma_2;\mathcal{S}_2 \vdash P \qquad \Phi;\Gamma_2;\mathcal{S}_2 \vdash Q}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash \textbf{if } v \textbf{ then } P \textbf{ else } Q}$$

(Tπ-HoInp)
$$\frac{\begin{array}{c}\Phi;\Gamma_1;\emptyset \vdash x : \ell_{\mathsf{i}}[\widetilde{T}] \qquad (\Phi;\Gamma_2), \tilde{y} : \widetilde{T};\mathcal{S} \vdash P \\ \text{if } \exists\, \tilde{\tilde{y}} : \widetilde{\widetilde{T}} \subseteq \tilde{y} : \widetilde{T} \text{ s.t. } T_i = T' \xrightarrow{1} \sigma,\ T_i \in \widetilde{\widetilde{T}} \text{ then } \tilde{\tilde{y}} \in \mathcal{S}\end{array}}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S} - \{\tilde{\tilde{y}}\} \vdash x?(\tilde{y}).P}$$

(Tπ-HoOut)
$$\frac{\begin{array}{c}\Phi;\Gamma_1;\emptyset \vdash x : \ell_{\mathsf{o}}[\widetilde{T}] \qquad \Phi;\widetilde{\Gamma_2};\widetilde{\mathcal{S}_2} \vdash \tilde{v} : \widetilde{T} \\ \Phi;\Gamma_3;\mathcal{S}_3 \vdash P \qquad \text{if } T_i = T' \to \sigma', \text{ then } \mathsf{un}(\Gamma_{2i}) \text{ and } \mathcal{S}_{2i} = \emptyset\end{array}}{\Phi;\Gamma_1 \uplus \widetilde{\Gamma_2} \uplus \Gamma_3;\widetilde{\mathcal{S}_2} \cup \mathcal{S}_3 \vdash x!\langle\tilde{v}\rangle.P}$$

(Tπ-HoCase)
$$\frac{\Phi;\Gamma_1;\mathcal{S}_1 \vdash v : \langle l_i\_T_i\rangle_{i\in I} \qquad \Phi;\Gamma_2, x_i : T_i;\mathcal{S}_2 \vdash P_i \quad \forall i \in I}{\Phi;\Gamma_1 \uplus \Gamma_2;\mathcal{S}_1 \cup \mathcal{S}_2 \vdash \textbf{case } v \textbf{ of } \{l_i\_x_i \triangleright P_i\}_{i\in I}}$$

Figure 9.6: Typing rules for the standard HOπ: processes

$$
\begin{array}{rcll}
[\![\diamond]\!] & \triangleq & \diamond & \text{(E-ProcType)} \\
[\![T \overset{1}{\to} \sigma]\!] & \triangleq & [\![T]\!] \overset{1}{\to} [\![\sigma]\!] & \text{(E-LinFunType)} \\
[\![T \to \sigma]\!] & \triangleq & [\![T]\!] \to [\![\sigma]\!] & \text{(E-FunType)} \\
[\![\star]\!]_f & \triangleq & \star & \text{(E-Star)} \\
[\![\lambda x : T.P]\!]_f & \triangleq & \lambda x : [\![T]\!].[\![P]\!]_f & \text{(E-Abstraction)} \\
[\![PQ]\!]_f & \triangleq & [\![P]\!]_f [\![Q]\!]_f & \text{(E-Application)}
\end{array}
$$

Figure 9.7: Encoding of HOπ types and terms

The encoding of HOπ session types and terms is an homomorphism and is given in Fig. 9.7. The process type, functional types, $\star$, abstraction and application in the HOπ calculus with sessions are encoded respectively as the process type, functional types, $\star$, abstraction and application in the standard HOπ calculus.

9.5 Properties of the Encoding

In this section we present the correctness of the encoding of HOπ constructs wrt typing derivations for values and processes. Since processes include values, we present the result in the same theorem. We also give the operational correspondence for HOπ constructs.

9.5.1 Typing HOπ Processes by Encoding

We start this section by introducing the following auxiliary lemmas.

Lemma 9.5.1. Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be sets of linear functional variables such that their union is defined. Let f be a renaming function for all $\mathcal{S}_i$ for $i \in 1 \ldots n$ such that $[\![\mathcal{S}_1]\!]_f \cup \ldots \cup [\![\mathcal{S}_n]\!]_f$ is defined. Then, $[\![\mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n]\!]_f = [\![\mathcal{S}_1]\!]_f \cup \ldots \cup [\![\mathcal{S}_n]\!]_f$.

Proof. The proof follows immediately by applying any renaming function on the disjoint union of sets of linear session functional variables. □

Lemma 9.5.2. The following hold.

- Let $\mathcal{S}$ be a set of linear functional variables and f a renaming function for $\mathcal{S}$ and $[\![\mathcal{S}]\!]_f = \mathcal{S}_1^\pi \cup \ldots \cup \mathcal{S}_n^\pi$. Then, $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ and for all $i \in 1 \ldots n$, $\mathcal{S}_i^\pi = [\![\mathcal{S}_i]\!]_f$.
- Let $\mathcal{S}^\pi = [\![\mathcal{S}_1]\!]_f \cup \ldots \cup [\![\mathcal{S}_n]\!]_f$ and f a renaming function for all $\mathcal{S}_i$ for $i \in 1 \ldots n$. Then, $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ and $\mathcal{S}^\pi = [\![\mathcal{S}]\!]_f$.

Proof. The proof follows immediately from the definition of the encoding of $\mathcal{S}$ and the disjoint union of subsets of $\mathcal{S}$. □

Lemma 9.5.3 (Substitution Lemma for Linear HOπ Calculus)**.** Let P be a HOπ process. The following hold.

- If $\Phi; \Gamma, x : T; \mathcal{S} \vdash P$ or
- If $\Phi, x : T; \Gamma; \mathcal{S} \vdash P$ or
- If $\Phi, x : T; \Gamma; \mathcal{S}, \{x\} \vdash P$ and

$\Phi'; \Gamma'; \mathcal{S}' \vdash v : T$ and Φ, Φ', and $\Gamma \uplus \Gamma'$ and $\mathcal{S}, \mathcal{S}'$ are defined, then it holds $\Phi, \Phi'; \Gamma \uplus \Gamma'; \mathcal{S}, \mathcal{S}' \vdash P[v/x]$.

Proof. The result is immediate as it is a generalisation of Lemma 4.5.2. □

We are ready now to present the main contribution of this chapter, namely the correctness of the encoding of higher-order processes wrt typing.

Theorem 9.5.4 (Soundness: Typing HOπ Processes)**.** If $[\![\Phi; \Gamma; \mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$ for some renaming function f for P, then $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$.

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

The cases for values different from λ-abstractions are trivial, as the encoding is an homomorphism and the typing rules for both the HOπ calculus with and without sessions follow the same line. We present only the case for a value being a λ-abstraction.

- Case $\lambda x : T.P$:
  By applying (E-Abstraction) and (E-FunType) we have

  $$[\![\lambda x : T.P]\!]_f = \lambda x : [\![T]\!].[\![P]\!]_f$$

  and $[\![T \to \sigma]\!] = [\![T]\!] \to [\![\sigma]\!]$. Since x is bound with scope P, then $f_x = x$. Assume

  $$[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]$$

  This implies that either rule (Tπ-HoAbs1) or rule (Tπ-HoAbs2) is applied. We consider both cases in the following:

  - Rule (Tπ-HoAbs1) is applied:

    $$\frac{\begin{array}{c}[\![\Phi]\!]_f, x : [\![T]\!]; [\![\Gamma]\!]_f; \mathcal{S}_1^\pi \vdash [\![P]\!]_f : [\![\sigma]\!] \\ \text{if } [\![T]\!] = T_1^\pi \overset{1}{\to} \sigma_1^\pi \text{ then } x \in \mathcal{S}_1^\pi\end{array}}{[\![\Phi]\!]_f; [\![\Gamma]\!]_f; \mathcal{S}_1^\pi - \{x\} \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]}$$

where $[\![\mathcal{S}]\!]_f = \mathcal{S}_1^\pi - \{x\}$, which implies $\mathcal{S}_1^\pi = [\![\mathcal{S}]\!]_f \cup \{x\}$. By Lemma 9.5.2 $[\![\mathcal{S}_1]\!]_f = \mathcal{S}_1^\pi$ and thus $\mathcal{S} = \mathcal{S}_1 - x$. By induction hypothesis we have $\Phi, x : T; \Gamma; \mathcal{S}_1 \vdash P : \sigma$. We conclude by (T-HoAbs1).

– Rule (Tπ-HoAbs2) is applied:

$$\frac{[\![\Phi]\!]_f; [\![\Gamma]\!]_f, x : [\![T]\!]; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]}{[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash \lambda x : [\![T]\!].[\![P]\!]_f : [\![T]\!] \to [\![\sigma]\!]}$$

By induction hypothesis $\Phi; \Gamma, x : T; \mathcal{S} \vdash P : \sigma$. Then, we obtain the result by applying rule (T-HoAbs1).

- Case PQ:
By (E-Application) we have $[\![PQ]\!]_f = [\![P]\!]_f[\![Q]\!]_f$ and assume

$$[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f[\![Q]\!]_f : [\![\sigma]\!]$$

Then, rule (Tπ-HoApp) is applied:

$$\frac{\begin{array}{c}[\![\Phi]\!]_f; \Gamma_1^\pi; \mathcal{S}_1^\pi \vdash [\![P]\!]_f : T^\pi \xrightarrow{1} [\![\sigma]\!] \\ [\![\Phi]\!]_f; \Gamma_2^\pi; \mathcal{S}_2^\pi \vdash [\![Q]\!]_f : T^\pi \qquad \text{if } T^\pi = T_1^\pi \to \sigma_1^\pi \text{ then } \mathsf{un}(\Gamma_2^\pi) \text{ and } \mathcal{S}_2^\pi = \emptyset\end{array}}{[\![\Phi]\!]_f; \Gamma_1^\pi \uplus \Gamma_2^\pi; \mathcal{S}_1^\pi \cup \mathcal{S}_2^\pi \vdash [\![P]\!]_f[\![Q]\!]_f : [\![\sigma]\!]}$$

We have that $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$ and $[\![\mathcal{S}]\!]_f = \mathcal{S}_1^\pi \cup \mathcal{S}_2^\pi$. By Lemma 6.3.5 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 9.5.2 we have that $[\![\mathcal{S}_1]\!]_f = \mathcal{S}_1^\pi$ and $[\![\mathcal{S}_2]\!]_f = \mathcal{S}_2^\pi$ such that $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$. By induction hypothesis $\Phi; \Gamma_1; \mathcal{S}_1 \vdash P : T \xrightarrow{1} \sigma$ where $T^\pi = [\![T]\!]$, and $\Phi; \Gamma_2; \mathcal{S}_2 \vdash Q : T$. Then, the result follows immediately by applying rule (T-HoApp) on the induction hypothesis.

- Case $x?(y).P$:
By assumption and by (E-Input) we have that

$$[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash f_x?(y, c).[\![P]\!]_{f,\{x \mapsto c\}}$$

Then, rule (Tπ-HoInp) is applied:

$$\frac{\begin{array}{c}[\![\Phi]\!]_f; \Gamma_1^\pi; \emptyset \vdash f_x : \ell_{\mathsf{i}}[T^\pi, U^\pi] \qquad ([\![\Phi]\!]_f; \Gamma_2^\pi), y : T^\pi, c : U^\pi; \mathcal{S}_1^\pi \vdash [\![P]\!]_{f,\{x \mapsto c\}} \\ \text{if } T^\pi = T_1^\pi \xrightarrow{1} \sigma^\pi, \text{ then } y \in \mathcal{S}_1^\pi\end{array}}{[\![\Phi]\!]_f; \Gamma_1^\pi \uplus \Gamma_2^\pi; \mathcal{S}_1^\pi - \{y\} \vdash f_x?(y, c).[\![P]\!]_{f,\{x \mapsto c\}}}$$

We have that $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi$. By Lemma 6.3.5 we obtain $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$ and $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2$. Moreover, $[\![\mathcal{S}]\!]_f = \mathcal{S}_1^\pi - \{y\}$,

namely $\mathcal{S}_1^\pi = [\![\mathcal{S}]\!]_f \cup \{y\}$ and notice that since y is bound, then $f_y = y$. By Lemma 9.5.2 $[\![\mathcal{S}_1]\!]_f = \mathcal{S}_1^\pi$ and $\mathcal{S} = \mathcal{S}_1 - y$. By induction hypothesis we have $\Phi; \Gamma_1; \emptyset \vdash x : ?T.U$ and, depending on whether y is a channel variable or not, we have one of the following:

$$\Phi; \Gamma_2, x : U, y : T; \mathcal{S} \vdash P \qquad \text{or} \qquad \Phi, y : T; \Gamma_2, x : U; \mathcal{S} \vdash P$$

where $T^\pi = [\![T]\!]$, $U^\pi = [\![U]\!]$. Then, we apply either rule (T-HoInp1) or rule (T-HoInp2) to obtain the result.

- Case $x!\langle v\rangle.P$:
  By assumption and by (E-Output) we have

  $$[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}$$

  By rule (Tπ-HoSess) we have

  $$\frac{}{[\![\Phi]\!]_f; c : \ell_\alpha[W]; \emptyset \vdash c : \ell_\alpha[W]}\ (\text{T}\pi\text{-HoSess})$$

  By rules (Tπ-HoRes1) and (Tπ-HoOut), we have the following derivation:

  $$\dfrac{\dfrac{\begin{array}{c}(\text{T}\pi\text{-HoRes1})\\(\text{T}\pi\text{-HoOut})\end{array}\quad \begin{array}{c}[\![\Phi]\!]_f; \Gamma_1^\pi; \emptyset \vdash f_x : \ell_{\mathsf{o}}[T^\pi, U^\pi] \qquad [\![\Phi]\!]_f; \Gamma_2^\pi; \mathcal{S}_2^\pi \vdash [\![v]\!]_f : T^\pi\\ [\![\Phi]\!]_f; \Gamma_3^\pi, c : \ell_{\overline{\alpha}}[W]; \mathcal{S}_3^\pi \vdash [\![P]\!]_{f,\{x\mapsto c\}} \qquad [\![\Phi]\!]_f; c : \ell_\alpha[W]; \emptyset \vdash c : \ell_\alpha[W]\\ \text{if } T^\pi = T_1^\pi \to \sigma_1^\pi, \text{ then } \mathsf{un}(\Gamma_2^\pi) \text{ and } \mathcal{S}_2^\pi = \emptyset\end{array}}{[\![\Phi]\!]_f; \Gamma_1^\pi \uplus \Gamma_2^\pi \uplus \Gamma_3^\pi, c : \ell_\sharp[T^\pi, U^\pi]; \mathcal{S}_2^\pi \cup \mathcal{S}_3^\pi \vdash f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}}{[\![\Phi]\!]_f; \Gamma_1^\pi \uplus \Gamma_2^\pi \uplus \Gamma_3^\pi; \mathcal{S}_2^\pi \cup \mathcal{S}_3^\pi \vdash (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

  We have $[\![\Gamma]\!]_f = \Gamma_1^\pi \uplus \Gamma_2^\pi \uplus \Gamma_3^\pi$. By Lemma 6.3.5 we have that $\Gamma_1^\pi = [\![\Gamma_1]\!]_f$, $\Gamma_2^\pi = [\![\Gamma_2]\!]_f$ and $\Gamma_3^\pi = [\![\Gamma_3]\!]_f$, such that $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$. We also have $[\![\mathcal{S}]\!]_f = \mathcal{S}_2^\pi \cup \mathcal{S}_3^\pi$. By Lemma 9.5.2 we have that $[\![\mathcal{S}_2]\!]_f = \mathcal{S}_2^\pi$ and $[\![\mathcal{S}_3]\!]_f = \mathcal{S}_3^\pi$ such that $\mathcal{S} = \mathcal{S}_2 \cup \mathcal{S}_3$. By induction hypothesis we have $\Phi; \Gamma_1; \emptyset \vdash x : !T.U$ where $\ell_{\mathsf{o}}[T^\pi, U^\pi] = [\![!T.U]\!]$, 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 (T-HoOut) on the induction hypothesis we obtain the result $\Phi; \Gamma_1 \circ \Gamma_2 \circ \Gamma_3; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash x!\langle v\rangle.P$. Notice that in the above we have used rule (Tπ-HoRes1) and hence we have that $[\![U]\!] = \ell_{\overline{\alpha}}[W]$ and $[\![\overline{U}]\!] = \ell_\alpha[W]$. The case for $[\![U]\!] = [\![\overline{U}]\!] = \emptyset[]$ and rule (Tπ-HoRes2) is symmetrical.

□

Theorem 9.5.5 (Completeness: Typing HOπ Processes)**.** If $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$, then $[\![\Phi; \Gamma; \mathcal{S}]\!]_f \vdash [\![P]\!]_f : [\![\sigma]\!]$ for some renaming function f for P.

Proof. The proof is done by induction on the derivation $\Phi; \Gamma; \mathcal{S} \vdash P : \sigma$.

- Case (T-HoFun):

$$\frac{\mathsf{un}(\Gamma)}{\Phi, x : T \overset{1}{\to} \sigma; \Gamma; \{x\} \vdash x : T \overset{1}{\to} \sigma}$$

  We need to prove that $[\![\Phi]\!]_f, f_x : [\![T \overset{1}{\to} \sigma]\!]; [\![\Gamma]\!]_f; \{f_x\} \vdash f_x : [\![T \overset{1}{\to} \sigma]\!]$. By Lemma 6.3.1 we obtain $\mathsf{un}([\![\Gamma]\!]_f)$. By (E-LinFunType) and by applying rule (Tπ-HoFun) we conclude the case.

- Case (T-HoAbs1):

$$\frac{\begin{array}{c}\Phi, x : T; \Gamma; \mathcal{S} \vdash P : \sigma \\ \text{if } T = T' \overset{1}{\to} \sigma \text{ then } x \in \mathcal{S}\end{array}}{\Phi; \Gamma; \mathcal{S} - \{x\} \vdash \lambda x : T.P : T \to \sigma}$$

  By induction hypothesis $[\![\Phi]\!]_{f'}, f'_x : [\![T]\!]; [\![\Gamma]\!]_{f'}; [\![\mathcal{S}]\!]_{f'} \vdash [\![P]\!]_{f'} : [\![\sigma]\!]$ for some function f' and if $[\![T]\!] = [\![T' \overset{1}{\to} \sigma]\!]$, then $f'_x \in [\![\mathcal{S}]\!]_{f'}$. Since f' is a renaming function for P and $x \in \mathsf{fv}(P)$, then $x \notin \mathsf{dom}([\![\Phi]\!]_{f'})$ and $x \notin \mathsf{dom}([\![\Gamma]\!]_{f'})$. We distinguish two cases, according to the shape of type T. If $T \neq T' \overset{1}{\to} \sigma$, then also $[\![T]\!] \neq [\![T' \overset{1}{\to} \sigma]\!]$. By typing rule (T$\pi$-HoVar) we have $x : [\![T]\!]; \emptyset; \emptyset \vdash x : [\![T]\!]$. Otherwise, if $T = T' \overset{1}{\to} \sigma$, then also $[\![T]\!] = [\![T' \overset{1}{\to} \sigma]\!]$. By rule (T$\pi$-HoFun) we have $x : [\![T]\!]; \emptyset; \{x\} \vdash x : [\![T]\!]$. Then, $[\![\Phi]\!]_{f'}, x : [\![T]\!]; [\![\Gamma]\!]_{f'}; [\![\mathcal{S}]\!]_{f'}[x/f'_x]$ is defined. Finally, by applying Lemma 9.5.3 $[\![\Phi]\!]_{f'}, x : [\![T]\!]; [\![\Gamma]\!]_{f'}; [\![\mathcal{S}]\!]_{f'}[x/f'_x] \vdash [\![P]\!]_{f'}[x/f'_x] : [\![\sigma]\!]$. Let $f = f', \{x \mapsto x\}$. It holds that if $[\![T]\!] = [\![T' \overset{1}{\to} \sigma]\!]$ then $x \in [\![\mathcal{S}]\!]_f$. Then, we write the induction hypothesis as $[\![\Phi]\!]_f, x : [\![T]\!]; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f$. By applying (E-Abstraction) and (E-FunType) and by rule (Tπ-HoAbs1) and Lemma 9.5.1 on the induction hypothesis, we obtain the result.

- Case (T-HoAbs2):

$$\frac{\Phi; \Gamma, x : T; \mathcal{S} \vdash P : \sigma}{\Phi; \Gamma; \mathcal{S} \vdash \lambda x : T.P : T \to \sigma}$$

  We need to prove that $[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![\lambda x : T.P]\!]_f : [\![T \to \sigma]\!]$. By induction hypothesis $[\![\Phi]\!]_{f'}; [\![\Gamma]\!]_{f'}, f'_x : [\![T]\!]; [\![\mathcal{S}]\!]_{f'} \vdash [\![P]\!]_{f'} : [\![\sigma]\!]$, for some

function f'. By rule (Tπ-HoSess) we can derive $\emptyset; x : [\![T]\!]; \emptyset \vdash x : [\![T]\!]$. As in (T-HoAbs1) it means that $[\![\Phi]\!]_{f'}; [\![\Gamma]\!]_{f'}, x : [\![T]\!]; [\![\mathcal{S}]\!]_{f'}$ is defined and by Lemma 9.5.3 $[\![\Phi]\!]_{f'}; [\![\Gamma]\!]_{f'}, x : [\![T]\!]; [\![\mathcal{S}]\!]_{f'} \vdash [\![P]\!]_{f'}[x/f'_x] : [\![\sigma]\!]$. Let $f = f', \{x \mapsto x\}$. By applying (E-Abstraction), (E-FunType) and typing rule (Tπ-HoAbs2) we obtain the result.

- Case (T-HoApp):

$$\frac{\Phi; \Gamma_1; \mathcal{S}_1 \vdash P : T \xrightarrow{1} \sigma \qquad \Phi; \Gamma_2; \mathcal{S}_2 \vdash Q : T \quad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S}_1 \cup \mathcal{S}_2 \vdash PQ : \sigma}$$

  By induction hypothesis $[\![\Phi]\!]_{f'}; [\![\Gamma_1]\!]_{f'}; [\![\mathcal{S}_1]\!]_{f'} \vdash [\![P]\!]_{f'} : [\![T]\!] \xrightarrow{1} [\![\sigma]\!]$ for some function f' and $[\![\Phi]\!]_{f''}; [\![\Gamma_2]\!]_{f''}; [\![\mathcal{S}_2]\!]_{f''} \vdash [\![Q]\!]_{f''} : [\![T]\!]$ for some function f''. Since $\Gamma_1 \circ \Gamma_2$ is defined, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and let $f'_D = f' \setminus \bigcup_{d \in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d \in D}\{d \mapsto f''(d)\} \setminus \bigcup_{q \in \Phi}\{q \mapsto f''(q)\}$. Hence, for all $d \in D$ we are not making any assumption on $f'(d)$ and $f''(d)$. Let $f = \bigcup_{d \in D}\{d \mapsto d'\} \cup f'_D \cup f''_D$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Moreover, f is a function since its subcomponents act on disjoint domains. By applying Lemma 6.3.6, the induction hypothesis can be rewritten as follows:

$$[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f; [\![\mathcal{S}_1]\!]_f \vdash [\![P]\!]_f : [\![T]\!] \xrightarrow{1} [\![\sigma]\!]$$

  and

$$[\![\Phi]\!]_f; [\![\Gamma_2]\!]_f; [\![\mathcal{S}_2]\!]_f \vdash [\![Q]\!]_f : [\![T]\!]$$

  By (E-Application), (Tπ-HoApp), by Lemma 6.3.4 and Lemma 9.5.1 we obtain the result: $[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; [\![\mathcal{S}_1]\!]_f \cup [\![\mathcal{S}_2]\!]_f \vdash [\![P]\!]_f [\![Q]\!]_f : [\![\sigma]\!]$.

- Case (T-HoInp1):

$$\frac{\Phi; \Gamma_1; \emptyset \vdash x : ?T.U \qquad \Phi; \Gamma_2, x : U, y : T; \mathcal{S} \vdash P}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} \vdash x?(y).P}$$

  By induction hypothesis and by (E-Inp) $[\![\Phi]\!]_{f'}; [\![\Gamma_1]\!]_{f'}; \emptyset \vdash f'_x : \ell_{\mathrm{i}}[[\![T]\!], [\![U]\!]]$, for some function f'. By induction hypothesis and by (E-Gamma) we have $[\![\Phi]\!]_{f''}; [\![\Gamma_2]\!]_{f''} \uplus f''_x : [\![U]\!] \uplus f''_y : [\![T]\!]; [\![\mathcal{S}]\!]_{f''} \vdash [\![P]\!]_{f''}$ for some function f''. By rule (Tπ-HoSess) we can derive $\emptyset; y : [\![T]\!]; \emptyset \vdash y : [\![T]\!]$. Since f'' is a renaming function for P and $y \in \mathsf{fv}(P)$, by the top-right premise of typing

rule (T-HoInp1), then $y \notin \mathsf{dom}(\llbracket\Phi\rrbracket_{f''})$, $y \notin \mathsf{dom}(\llbracket\Gamma_2\rrbracket_{f''})$ and $y \neq f''_x$. Then, $\llbracket\Phi\rrbracket_{f''}; \llbracket\Gamma_2\rrbracket_{f''} \uplus f''_x : \llbracket U\rrbracket \uplus y : \llbracket T\rrbracket; \llbracket\mathcal{S}\rrbracket_{f''}$ is defined. By Lemma 9.5.3 we obtain that $\llbracket\Phi\rrbracket_{f''}; \llbracket\Gamma_2\rrbracket_{f''} \uplus f''_x : \llbracket U\rrbracket \uplus y : \llbracket T\rrbracket; \llbracket\mathcal{S}\rrbracket_{f''} \vdash \llbracket P\rrbracket_{f''}[y/f''_y]$. Since $\Gamma_1 \circ \Gamma_2$ is defined, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and define $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\} \setminus \bigcup_{q\in\Phi}\{q \mapsto f''(q)\}$, meaning that only f'_D acts on variables in Φ. Suppose $f''(x) = c$. We let $f = \bigcup_{d\in D}\{d \mapsto d'\} \cup f'_D \cup f''_D, \{y \mapsto y\} \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Notice that $f''_D(y)$ is defined and is f''_y from the induction hypothesis. Then, $f''_D, \{y \mapsto y\}$ updates f''_y to y by mapping $\{y \mapsto y\}$. Moreover, f is a function since its subcomponents act on disjoint domains. By applying Lemma 6.3.6, we can rewrite the induction hypothesis as:

$$\llbracket\Phi\rrbracket_f; \llbracket\Gamma_1\rrbracket_f; \emptyset \vdash f_x : \ell_{\mathsf{i}}[\llbracket T\rrbracket, \llbracket U\rrbracket]$$

and

$$\llbracket\Phi\rrbracket_f; \llbracket\Gamma_2\rrbracket_f \uplus c : \llbracket U\rrbracket \uplus y : \llbracket T\rrbracket; \llbracket\mathcal{S}\rrbracket_f \vdash \llbracket P\rrbracket_{f,\{x\mapsto c\}}$$

Since $x, y \notin \mathsf{dom}(\Gamma_2)$, then $\llbracket\Gamma_2, x : U, y : T\rrbracket_{f,\{x\mapsto c\}}$ can be optimised as $\llbracket\Gamma_2\rrbracket_f \uplus c : \llbracket U\rrbracket \uplus y : \llbracket T\rrbracket$. Also in $\llbracket\Phi\rrbracket_f$ and $\llbracket\mathcal{S}\rrbracket_f$ we simply use f. Then, by Lemma 6.3.2 we have $\llbracket\Phi\rrbracket_f; \llbracket\Gamma_2\rrbracket_f, c : \llbracket U\rrbracket, y : \llbracket T\rrbracket; \llbracket\mathcal{S}\rrbracket_f \vdash \llbracket P\rrbracket_{f,\{x\mapsto c\}}$. By (E-Input), rule (T$\pi$- HoInp) and by Lemma 6.3.4 we obtain the result $\llbracket\Phi\rrbracket_f; \llbracket\Gamma_1\rrbracket_f \uplus \llbracket\Gamma_2\rrbracket_f; \llbracket\mathcal{S}\rrbracket_f \vdash f_x?(y, c).\llbracket P\rrbracket_{f,\{x\mapsto c\}}$.

- Case (T-HoInp2):

$$\frac{\Phi; \Gamma_1; \emptyset \vdash x : ?T.U \qquad \Phi, y : T; \Gamma_2, x : U; \mathcal{S} \vdash P \qquad \text{if } T = T' \overset{1}{\to} \sigma \text{ then } y \in \mathcal{S}}{\Phi; \Gamma_1 \circ \Gamma_2; \mathcal{S} - \{y\} \vdash x?(y).P}$$

By induction hypothesis and by (E-Inp) $\llbracket\Phi\rrbracket_{f'}; \llbracket\Gamma_1\rrbracket_{f'}; \emptyset \vdash f'_x : \ell_{\mathsf{i}}[\llbracket T\rrbracket, \llbracket U\rrbracket]$, for some function f'. By induction hypothesis and by (E-Gamma) $\llbracket\Phi\rrbracket_{f''}, f''_y : \llbracket T\rrbracket; \llbracket\Gamma_2\rrbracket_{f''} \uplus f''_x : \llbracket U\rrbracket; \llbracket\mathcal{S}\rrbracket_{f''} \vdash \llbracket P\rrbracket_{f''}$, for some function f''. Since f'' is a renaming function for P and $y \in \mathsf{fv}(P)$, by the second premise of (T-HoInp2), then $y \notin \mathsf{dom}(\llbracket\Phi\rrbracket_{f''})$, $y \notin \mathsf{dom}(\llbracket\Gamma_2\rrbracket_{f''})$ and $y \neq f''_x$. Then, by following the same reasoning as in (T-HoAbs1) and by Lemma 9.5.3 we have $\llbracket\Phi\rrbracket_{f''}, y : \llbracket T\rrbracket; \llbracket\Gamma_2\rrbracket_{f''} \uplus f''_x : \llbracket U\rrbracket; \llbracket\mathcal{S}\rrbracket_{f''}[y/f''_y] \vdash \llbracket P\rrbracket_{f''}[y/f''_y]$. Since $\Gamma_1 \circ \Gamma_2$ is defined, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$ it holds that $\Gamma_1(x) = \Gamma_2(x) = T$ and $\mathsf{un}(T)$. Let $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = D$ and define $f'_D = f' \setminus \bigcup_{d\in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d\in D}\{d \mapsto f''(d)\} \setminus \bigcup_{q\in\Phi}\{q \mapsto f''(q)\}$, meaning that only f'_D acts on variables in Φ. Suppose $f''(x) = c$. We let

$f = \bigcup_{d \in D}\{d \mapsto d'\} \cup f'_D \cup f''_D, \{y \mapsto y\} \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Notice that $f''_D(y)$ is defined and is f''_y from the induction hypothesis. Then, $f''_D, \{y \mapsto y\}$ updates f''_y to y by mapping $\{y \mapsto y\}$. Moreover, f is a function since its subcomponents act on disjoint domains. By applying Lemma 6.3.6, we can rewrite the induction hypothesis as:

$$[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f; \emptyset \vdash f_x : \ell_{\mathsf{i}}[[\![T]\!], [\![U]\!]]$$

and

$$[\![\Phi]\!]_f, y : [\![T]\!]; [\![\Gamma_2]\!]_f \uplus c : [\![U]\!]; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_{f,\{x \mapsto c\}}$$

Since $x \notin \mathsf{dom}(\Gamma_2)$, then $[\![\Gamma_2, x : U]\!]_{f,\{x \mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_2]\!]_f \uplus c : [\![U]\!]$. We also use f in $[\![\Phi]\!]_f$ and $[\![\mathcal{S}]\!]_f$. Moreover, the condition "if $[\![T]\!] = [\![T']\!] \xrightarrow{1} [\![\sigma]\!]$ then $y \in \mathcal{S}$" holds. By (E-Input), typing rule (Tπ-HoInp) and by Lemma 6.3.2 and Lemma 6.3.4 we obtain the result $[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f; [\![\mathcal{S}]\!]_f - \{y\} \vdash f_x?(y, c).[\![P]\!]_{f,\{x \mapsto c\}}$.

- Case (T-HoOut):

$$\frac{\Phi; \Gamma_1; \emptyset \vdash x : !T.U \qquad \Phi; \Gamma_2; \mathcal{S}_2 \vdash v : T \qquad \Phi; \Gamma_3, x : U; \mathcal{S}_3 \vdash P \qquad \text{if } T = T' \to \sigma' \text{ then } \mathsf{un}(\Gamma_2) \text{ and } \mathcal{S}_2 = \emptyset}{\Phi; \Gamma_1 \circ \Gamma_2 \circ \Gamma_3; \mathcal{S}_2 \cup \mathcal{S}_3 \vdash x!\langle v \rangle.P}$$

By induction hypothesis and (E-Out) $[\![\Phi]\!]_{f'}; [\![\Gamma_1]\!]_{f'}; \emptyset \vdash f'_x : \ell_{\mathsf{o}}[[\![T]\!], [\![\overline{U}]\!]]$, for some function f' and $[\![\Phi]\!]_{f''}; [\![\Gamma_2]\!]_{f''}; [\![\mathcal{S}_2]\!]_{f''} \vdash [\![v]\!]_{f''} : [\![T]\!]$ for some function f'' and by (E-Gamma) $[\![\Phi]\!]_{f'''}; [\![\Gamma_3]\!]_{f'''} \uplus f'''_x : [\![U]\!]; [\![\mathcal{S}_3]\!]_{f'''} \vdash [\![P]\!]_{f'''}$ for some function f'''. Since $\Gamma_1 \circ \Gamma_2 \circ \Gamma_3$ is defined by assumption, then for all $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \cap \mathsf{dom}(\Gamma_3)$, we have $\Gamma_1(x) = \Gamma_2(x) = \Gamma_3(x) = T$ and $\mathsf{un}(T)$. Now, let $D = \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \cap \mathsf{dom}(\Gamma_3)$. We define $f'_D = f' \setminus \bigcup_{d \in D}\{d \mapsto f'(d)\}$ and $f''_D = f'' \setminus \bigcup_{d \in D}\{d \mapsto f''(d)\} \setminus \bigcup_{q \in \Phi}\{q \mapsto f''(q)\}$ and $f'''_D = f''' \setminus \bigcup_{d \in D}\{d \mapsto f'''(d) \setminus \bigcup_{q \in \Phi}\{q \mapsto f''(q)\}$. Suppose $f'''_x = c$. Then, define $f = \bigcup_{d \in D}\{d \mapsto d'\} \cup f'_D \cup f''_d \cup f'''_D \setminus \{x \mapsto c\}$, where for all $d \in D$ we create a fresh name d' and associate $d \mapsto d'$. Notice that f is a function because its subcomponents act on disjoint domains. Then, by Lemma 6.3.6, the induction hypothesis can be rewritten as follows:

$$[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f; \emptyset \vdash f_x : \ell_{\mathsf{o}}[[\![T]\!], [\![\overline{U}]\!]] \qquad [\![\Phi]\!]_f; [\![\Gamma_2]\!]_f; [\![\mathcal{S}_2]\!]_f \vdash [\![v]\!]_f : [\![T]\!]$$

and

$$[\![\Phi]\!]_f; [\![\Gamma_3]\!]_f \uplus c : [\![U]\!]; [\![\mathcal{S}_3]\!]_f \vdash [\![P]\!]_{f,\{x \mapsto c\}}$$

Since $x \notin \mathsf{dom}(\Gamma_3)$, then $[\![\Gamma_3, x : U]\!]_{f,\{x \mapsto c\}}$ can be optimised and distributed as $[\![\Gamma_3]\!]_f \uplus c : [\![U]\!]$. Assume $U \neq \mathsf{end}$ and hence $\overline{U} \neq \mathsf{end}$. By (T$\pi$-HoSess)

we can derive $c : [\![\overline{U}]\!] \vdash c : [\![\overline{U}]\!]$. By rule (T$\pi$- HoOut) and by using Lemma 6.3.7 and "$\uplus$" operator to obtain $c : \ell_\sharp\,[W]$, we have the following:

$$\frac{\begin{array}{c}[\![\Phi]\!]_f; c : [\![\overline{U}]\!]; \emptyset \vdash c : [\![\overline{U}]\!] \\ [\![\Phi]\!]_f; [\![\Gamma_1]\!]_f; \emptyset \vdash f_x : \ell_{\mathsf{o}}[[\![T]\!], [\![\overline{U}]\!]] \qquad [\![\Phi]\!]_f; [\![\Gamma_2]\!]_f; [\![\mathcal{S}_2]\!]_f \vdash [\![v]\!]_f : [\![T]\!] \\ [\![\Phi]\!]_f; [\![\Gamma_3]\!]_f \uplus c : [\![U]\!]; [\![\mathcal{S}_3]\!]_f \vdash [\![P]\!]_{f,\{x\mapsto c\}}\end{array}}{[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f \uplus c : \ell_\sharp[W]; [\![\mathcal{S}_2]\!]_f \cup [\![\mathcal{S}_3]\!]_f \vdash f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

Then, by Lemma 6.3.2 and by applying (Tπ-Res1) we have the following:

$$\frac{[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f, c : \ell_\sharp[W]; [\![\mathcal{S}_2]\!]_f \cup [\![\mathcal{S}_3]\!]_f \vdash f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}{[\![\Phi]\!]_f; [\![\Gamma_1]\!]_f \uplus [\![\Gamma_2]\!]_f \uplus [\![\Gamma_3]\!]_f; [\![\mathcal{S}_2]\!]_f \cup [\![\mathcal{S}_3]\!]_f \vdash (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}}$$

The case where $U = \overline{U} = \mathsf{end}$, which yields $c : \emptyset[]$, is symmetrical and is obtained by using (Tπ- Res2) instead of (Tπ- Res1). By (E-Output) and Lemma 6.3.4 and Lemma 9.5.1 we conclude the proof.

□

9.5.2 Operational Correspondence for HOπ

In the following, we prove the operational correspondence in the case of higher-order constructs.

Proof of Theorem 6.3.15 for Higher-Order Terms: Let P be a session process, $\Phi, \Gamma, \mathcal{S}$ session typing contexts, and f a renaming function for P such that $[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f$. Then, the following statements hold.

1. If $P \to P'$, then $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$.
2. If $[\![P]\!]_f \to Q$, then there are $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, and either $f' = f$ or $f' = f, \{x, y \mapsto c\}$ for x, y such that (νxy) appears in $\mathcal{E}[P]$.

Proof. Since $[\![\Phi]\!]_f; [\![\Gamma]\!]_f; [\![\mathcal{S}]\!]_f \vdash [\![P]\!]_f$, then by Theorem 9.5.4 it is the case that $\Phi; \Gamma; \mathcal{S} \vdash P$. We consider both cases in the following.

1. The proof is done by induction on the derivation $P \to P'$.
   - Case (R-Beta):

$$P \triangleq (\lambda x : T.Q)v \to Q[v/x] \triangleq P'$$

By the encoding of abstraction in HOπ with session types we have:

$$\begin{aligned} [\![P]\!]_f &= [\![(\lambda x : T.Q)v]\!]_f \\ &= (\lambda x : [\![T]\!].[\![Q]\!]_f)[\![v]\!]_f \\ &\to [\![Q]\!]_f[[\![v]\!]_f/x] \end{aligned}$$

Notice that x is bound with scope Q, hence $f_x = x$. On the other hand, by the encoding of P' and by using Lemma 6.3.13 we have:

$$[\![P']\!]_f = [\![Q[v/x]]\!]_f = [\![Q]\!]_f[[\![v]\!]_f/f_x] = [\![Q]\!]_f[[\![v]\!]_f/x]$$

This implies that $[\![P]\!]_f \to\equiv [\![P']\!]_f$.

- Case (R-ApplLeft):

$$\frac{P \to P'}{PQ \to P'Q}$$

By induction hypothesis $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$. We conclude by context closure of structural congruence and by applying rules (Rπ-ApplLeft) and (Rπ-Struct).

- Case (R-ApplRight):

$$\frac{P \to P'}{vP \to vP'}$$

This case is symmetrical to the previous one. By induction hypothesis $[\![P]\!]_f \to\hookrightarrow [\![P']\!]_f$. We conclude by context closure of structural congruence and by applying rules (Rπ-ApplRight) and (Rπ-Struct).

2. The proof is done by induction on the structure of the higher-order session process P. There is only one case to be considered in addition to the cases of Theorem 6.3.15, namely $P = P_1P_2$. By (E-Application) and by assumption we have $[\![P_1]\!]_f[\![P_2]\!]_f \to Q$. We need to show that there exist $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, and either $f' = f$ or $f' = f, \{x, y \mapsto c\}$ for x, y such that $(\boldsymbol{\nu} xy)$ appears in $\mathcal{E}[P]$. There are only the following cases to be considered:

   - $P = P_1P_2 = (\lambda x : T.Q')v$ and the abstraction $\lambda x : T.Q'$ is applied on v. By assumption we have that $[\![\lambda x : T.Q']\!]_f[\![v]\!]_f \to Q$, for some Q. By (E-Abstraction) $[\![\lambda x : T.Q']\!]_f = \lambda x : [\![T]\!].[\![Q']\!]_f$ and since x is bound with scope Q', then $f_x = x$. Then, by rule (Rπ-Beta) we have $(\lambda x : [\![T]\!].[\![Q']\!]_f)[\![v]\!]_f \to [\![Q']\!]_f[[\![v]\!]_f/x]$. Let $Q = [\![Q']\!]_f[[\![v]\!]_f/x]$. We need to show that there exist $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and

$Q \hookrightarrow [\![P']\!]_{f'}$, where either $f' = f$ or $f' = f, \{z, w \mapsto c\}$ such that (νzw) appears in $\mathcal{E}[P]$. Let $\mathcal{E}[\cdot] = [\cdot]$, $P' = Q'[v/x]$ and $f' = f$. Then, it is the case that $P \to Q'[v/x]$ and $[\![P']\!]_f = [\![Q'[v/x]]\!]_f$, which by Lemma 6.3.13 means that $[\![Q'[v/x]]\!]_f = [\![Q']\!]_f[[\![v]\!]_f/f_x]$. Again, since $f_x = x$, then $Q \equiv [\![Q']\!]_f[[\![v]\!]_f/x]$.

- Only $[\![P_1]\!]_f$ reduces.

  Let $[\![P_1]\!]_f \to R$. By rule (Rπ-AppLLeft) $[\![P_1]\!]_f[\![P_2]\!]_f \to R[\![P_2]\!]_f$. Let $Q = R[\![P_2]\!]_f$. By induction hypothesis, since P_1 is a subprocess of P and $[\![P_1]\!]_f \to R$, there exist $P_1', \mathcal{E}'[\cdot]$, such that $\mathcal{E}'[P_1] \to \mathcal{E}'[P_1']$ and $R \hookrightarrow [\![P_1']\!]_{f''}$, where either $f'' = f$ or $f'' = f, \{z, w \mapsto c\}$, such that (νzw) appears in $\mathcal{E}'[P_1]$. Let $\mathcal{E}[\cdot] = \mathcal{E}'[\cdot]$. Since $\mathcal{E}[\cdot]$ is a suitable context for P_1 and $\Phi; \Gamma; \mathcal{S} \vdash P_1P_2$ it means that for all (νzw) that appear in $\mathcal{E}[P_1]$, it is the case that $z, w \notin \mathsf{fv}(P_2)$. Hence, by structural congruence we obtain that $\mathcal{E}[P_1]P_2 \equiv \mathcal{E}[P_1P_2]$ (1). By rule (R-AppLLeft) we have $\mathcal{E}[P_1]P_2 \to \mathcal{E}[P_1']P_2$ (2). Again, by structural congruence we have $\mathcal{E}[P_1'] \mid P_2 \equiv \mathcal{E}[P_1' \mid P_2]$ (3). By rule (R-Struct) on (1), (2), (3) we can conclude that $\mathcal{E}[P_1P_2] \to \mathcal{E}[P_1'P_2]$. Let $P' = P_1'P_2$ and $f' = f$. Then, $R[\![P_2]\!]_f \hookrightarrow [\![P_1']\!]_f[\![P_2]\!]_f = [\![P']\!]_f$.

- Only $[\![P_2]\!]_f$ reduces.

  This case is symmetrical to the previous one where the roles of P_1 and P_2 are exchanged and rules (Rπ- AppLRight) and (R-AppLRight) are used instead of (Rπ-AppLLeft) and (R-AppLLeft), respectively.

□

Chapter 10

Recursion

So far we have worked with processes that have a finite behaviour. In this chapter, we introduce *recursion*, which is widely known and used not only in process calculi, but also in other programming paradigms. Replication, on the other hand, is a simple form of recursion. It states what is exactly needed, for example in representing data and functions [101]. 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. In [95] the authors show an encoding that relates the two constructs.

10.1 Syntax

In this section we present the syntax of types and terms for both the π-calculus with and without sessions.

Recursion in the 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 types and terms

Recursion in the 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 capability α)
	α	(unrestricted qualifier used in capability α)
$\tau ::=$	$m_\alpha[\widetilde{T}]$	(channel type used in capability m_α)
	$\emptyset[\widetilde{T}]$	(channel with no capability)
$T ::=$	τ	(channel type)
	$\mathbf{t} \mid \bar{\mathbf{t}}$	(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-calculus types and terms

Recall that α ranges over 'i' input, 'o' output, or '$\sharp$' connection capabilities. A channel can be of a linear type $\ell_\alpha[\widetilde{T}]$, or of an unrestricted type $\alpha[\widetilde{T}]$, or without any capability $\emptyset[\widetilde{T}]$. The syntax of types includes type variables $\mathbf{t}, \bar{\mathbf{t}}$ and recursive types, in addition to the types given in Fig. 4.5. μ and **rec** are binders of the type and process variables, respectively. 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 π-calculus with and without sessions, satisfy the constraint that the type variable $\mathbf{t}$ of the $\mu\mathbf{t}.T$ expression is *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*, i.e., do not contain subexpressions like $\mu\mathbf{t}_1 \ldots \mu\mathbf{t}_n.T$.

10.2 Semantics

The reduction rule for the recursive process is the same in both the π-calculus with and without sessions and is given in the following.

$$\text{(R}[\pi]\text{-Rec)} \qquad \frac{P[\mathbf{rec}X.P/X] \to P'}{\mathbf{rec}X.P \to P'}$$

Rule (Rπ-Rec) states that a recursive process $\mathbf{rec}X.P$ reduces to a process P' if process P where X is substituted by the recursion process $\mathbf{rec}X.P$, reduces to

the same 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

On types for the π-calculus with sessions Type duality for session types extends the inductive type duality for finite types, earlier defined, to accommodate recursive types.

$$\begin{array}{rcl} \overline{\mathbf{t}} & \triangleq & \mathbf{t} \\ \overline{\mu\mathbf{t}.T} & \triangleq & \mu\mathbf{t}.\overline{T} \end{array}$$

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 $\overline{\cdot}$, since it does not commute with unfolding. In order to overcome this problem, we follow the standard way adopted in the literature, considering the above type to be ill-formed, and thus ruling it out. We let the exploration of more accurate duality relations as future work.

On types for the standard π-calculus Type duality for standard π-types is defined as follows:

$$\begin{array}{rcl} \overline{m_{\mathsf{i}}\,[\widetilde{T}]} & \triangleq & m_{\mathsf{o}}\,[\widetilde{T}] \\ \overline{m_{\mathsf{o}}\,[\widetilde{T}]} & \triangleq & m_{\mathsf{i}}\,[\widetilde{T}] \\ \overline{\emptyset[\widetilde{T}]} & \triangleq & \emptyset[\widetilde{T}] \\ \overline{\mathbf{t}} & \triangleq & \mathbf{t} \\ \overline{\mu\mathbf{t}.T} & \triangleq & \mu\mathbf{t}.\overline{T}[\overline{\mathbf{t}}/\mathbf{t}] \end{array}$$

It also holds that $\overline{\overline{\mathbf{t}}} = \mathbf{t}$. A type variable $\mathbf{t}$ and its dual $\overline{\mathbf{t}}$ are treated differently as long as *substitution* is concerned, which is defined in the following.

Definition 10.3.1. The *substitution* of a standard π-calculus type T for a type variable $\mathbf{t}$ is defined as follows:

$$\begin{array}{rcl} \mathbf{t}[T/\mathbf{t}] & \triangleq & T \\ \overline{\mathbf{t}}[T/\mathbf{t}] & \triangleq & \overline{T} \end{array}$$

The combination of types is defined as follows:

$$\begin{array}{rcll} m_{\mathsf{o}}\,[\widetilde{T}] \uplus m_{\mathsf{i}}\,[\widetilde{T}] & \triangleq & m_{\sharp}\,[\widetilde{T}] & \\ T \uplus T & \triangleq & T & \text{if } \mathsf{un}(T) \\ T \uplus S & \triangleq & \mathtt{undef} & \text{otherwise} \end{array}$$

In particular, the second definition implies $\emptyset[\widetilde{T}] \uplus \emptyset[\widetilde{T}] = \emptyset[\widetilde{T}]$.

$$\frac{\Theta(X) = \Gamma}{\Theta; \Gamma \vdash X}\ (\text{T}[\pi]\text{-RecVar}) \qquad \frac{\Theta, X : \Gamma; \Gamma \vdash P}{\Theta; \Gamma \vdash \mathbf{rec}X.P}\ (\text{T}[\pi]\text{-RecProc})$$

$$\frac{\Theta, \Gamma \vdash v : T \quad T \sim_{\mathsf{type}} S}{\Theta, \Gamma \vdash v : S}\ (\text{T}[\pi]\text{-EqVal})$$

Figure 10.3: Typing rules for recursive constructs

Typing contexts The typing context Γ is defined for both the π-calculi with and without sessions 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 the π-calculus with and without sessions The typing rules for the recursive process added to the π-calculus with and without sessions is given in 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 and the typing judgements are augmented with Θ.

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.

Types Encoding:

$$
\begin{array}{rcll}
[\![\mathsf{end}]\!] & \triangleq & \emptyset[] & \text{(E-qEnd)} \\
[\![q!T.U]\!] & \triangleq & m_{\mathsf{o}}[[\![T]\!], \overline{[\![U]\!]}] & \text{(E-qOut)} \\
[\![q?T.U]\!] & \triangleq & m_{\mathsf{i}}[[\![T]\!], [\![U]\!]] & \text{(E-qInp)} \\
[\![q \oplus \{l_i : T_i\}_{i\in I}]\!] & \triangleq & m_{\mathsf{o}}[\langle l_i \_ \overline{[\![T_i]\!]}\rangle_{i\in I}] & \text{(E-qSelect)} \\
[\![q\&\{l_i : T_i\}_{i\in I}]\!] & \triangleq & m_{\mathsf{i}}[\langle l_i \_ [\![T_i]\!]\rangle_{i\in I}] & \text{(E-qBranch)} \\
[\![\mathbf{t}]\!] & \triangleq & \mathbf{t} & \text{(E-TVar)} \\
[\![\mu\mathbf{t}.T]\!] & \triangleq & \mu\mathbf{t}.[\![T]\!] & \text{(E-TRec)}
\end{array}
$$

Terms Encoding:

$$
\begin{array}{rcll}
[\![X]\!]_f & \triangleq & X & \text{(E-PVar)} \\
[\![\mathbf{rec}X.P]\!]_f & \triangleq & \mathbf{rec}X.[\![P]\!]_f & \text{(E-PRec)}
\end{array}
$$

Typing Context Encoding:

$$
\begin{array}{rcll}
[\![\emptyset]\!]_f & \triangleq & \emptyset & \text{(E-Empty)} \\
[\![\Gamma, x : T]\!]_f & \triangleq & [\![\Gamma]\!]_f \uplus f_x : [\![T]\!] & \text{(E-Gamma)} \\
[\![\Theta, X : \Gamma]\!]_f & \triangleq & [\![\Theta]\!]_f, X : [\![\Gamma]\!]_f & \text{(E-Theta)} \\
[\![\Theta; \Gamma]\!]_f & \triangleq & [\![\Theta]\!]_f; [\![\Gamma]\!]_f & \text{(E-CtxRec)}
\end{array}
$$

Figure 10.4: Encoding of recursive types, terms and typing contexts

The encoding of types is a conservative extension of the encoding presented in Section 6.1. Here, we give the encoding of both the linear and the unrestricted pretypes as well as the recursive types that we introduced at the beginning of this chapter. The encoding of 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 follows 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 follows 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 (E-TVar) and (E-TRec), respectively for the recursive type variable $\mathbf{t}$ and for the recursive type $\mu\mathbf{t}.T$. It is important to notice that, the duality function in the

encoding of session types in rules (E-qOut) and (E-qSelect), is now applied on the encoded session type rather than on the session type itself. This is to accommodate the encoding of dual type variables. Moreover, it is easy to see that for all finite session types T it is the case that $[\![\overline{T}]\!] = \overline{[\![T]\!]}$.

The encoding of processes is the one presented in Section 6.2, with the addition of two new definitions for recursion, being (E-Pvar) and (E-Prec): the encoding of a process variable and a recursive process is an homomorphism. The encoding of typing contexts is as expected.

Finally, in order to better understand the non-standard substitution in Definition 10.3.1, we give the following example.

Example 10.4.1. Let $S \triangleq \mu\mathbf{t}.!\mathtt{Bool}.\mathbf{t}$ be a session type. Then, by duality on recursive session types, given in Section 10.3, the dual of S is $\overline{S} = \mu\mathbf{t}.!\mathtt{Bool}.\overline{\mathbf{t}} = \mu\mathbf{t}.!\overline{\mathtt{Bool}}.\overline{\mathbf{t}} = \mu\mathbf{t}.?\mathtt{Bool}.\mathbf{t}$. By the encoding of recursive session types, we have

$$T \triangleq [\![\mu\mathbf{t}.!\mathtt{Bool}.\mathbf{t}]\!] = \mu\mathbf{t}.\ell_{\mathsf{o}}\,[[\![\mathtt{Bool}]\!], \overline{[\![\mathbf{t}]\!]}] = \mu\mathbf{t}.\ell_{\mathsf{o}}\,[\mathtt{Bool}, \bar{\mathbf{t}}]$$

and

$$\overline{T} = [\![\mu\mathbf{t}.?\mathtt{Bool}.\mathbf{t}]\!] = \mu\mathbf{t}.\ell_{\mathsf{i}}\,[[\![\mathtt{Bool}]\!], [\![\mathbf{t}]\!]] = \mu\mathbf{t}.\ell_{\mathsf{i}}\,[\mathtt{Bool}, \mathbf{t}]$$

If we unfold the above types, we have

$$\begin{aligned} T &= \mu\mathbf{t}.\ell_{\mathsf{o}}\,[\mathtt{Bool}, \bar{\mathbf{t}}] \\ &\sim_{\mathsf{type}} \ell_{\mathsf{o}}\,[\mathtt{Bool}, \bar{\mathbf{t}}[T/\mathbf{t}]] \\ &= \ell_{\mathsf{o}}\,[\mathtt{Bool}, \overline{T}] \\ &= \ell_{\mathsf{o}}\,[\mathtt{Bool}, \mu\mathbf{t}.\ell_{\mathsf{i}}\,[\mathtt{Bool}, \mathbf{t}]] \end{aligned}$$

and

$$\begin{aligned} \overline{T} &= \mu\mathbf{t}.\ell_{\mathsf{i}}\,[\mathtt{Bool}, \mathbf{t}] \\ &\sim_{\mathsf{type}} \ell_{\mathsf{i}}\,[\mathtt{Bool}, \mathbf{t}[\overline{T}/\mathbf{t}]] \\ &= \ell_{\mathsf{i}}\,[\mathtt{Bool}, \overline{T}] \\ &= \ell_{\mathsf{i}}\,[\mathtt{Bool}, \mu\mathbf{t}.\ell_{\mathsf{i}}\,[\mathtt{Bool}, \mathbf{t}]] \end{aligned}$$

10.5 Properties of the Encoding

In this section we present the main properties related to the encoding of recursive types and terms. Notice in the following that the typing judgements are different wrt the ones in the original theorem, in that they are augmented with Θ to accommodate assumptions on recursive variables.

To complete Theorem 6.3.10 and Theorem 6.3.11 on the correctness of the encoding wrt typing processes, it suffices to add the case for recursive processes. In order to accommodate recursion, the new typing context Θ has to be considered. Previous typing judgements of the form $\Gamma \vdash P$ should be now written as $\Theta; \Gamma \vdash P$, (with $\Theta = \emptyset$ in absence of recursion). These modifications will affect also the statement of operational correspondence given by Theorem 6.3.15.

Proof of Theorem 6.3.10 and Theorem 6.3.11 for Recursive Processes:

1. If $[\![\Theta; \Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P, then $\Theta; \Gamma \vdash P$.

2. If $\Theta; \Gamma \vdash P$, then $[\![\Theta; \Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function f for P.

Proof. We split the proof as follows:

1. The proof is done by induction on the structure of the process.

   We consider only the case for the recursive process $\mathbf{rec}X.P$. By (E-PRec) we have $[\![\mathbf{rec}X.P]\!]_f = \mathbf{rec}X.[\![P]\!]_f$ 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). By induction hypothesis $[\![\Theta]\!]_{f'}, X : [\![\Gamma]\!]_{f'}; [\![\Gamma]\!]_{f'}, \vdash [\![P]\!]_{f'}$, for some function f'. We conclude by letting $f = f'$ and by applying (T-RecProc).

2. The proof is done by induction on the derivation $\Theta; \Gamma \vdash P$.

   We consider only the case for (T-RecProc). By induction hypothesis we have that $[\![\Theta]\!]_{f'}, X : [\![\Gamma]\!]_{f'}; [\![\Gamma]\!]_{f'} \vdash [\![P]\!]_{f'}$, for some function f'. By letting $f = f'$ and by rule (Tπ-RecProc) we obtain $[\![\Theta]\!]_f; [\![\Gamma]\!]_f \vdash [\![\mathbf{rec}X.P]\!]_f$.

□

In the following we show the operational correspondence in the case of recursive processes. We first start with an auxiliary definition.

Lemma 10.5.1. Let Q be a session process and let $Q[\mathbf{rec}X.Q/X]$ denote process Q where process variable X is substituted by $\mu X.Q$. Then,

$$[\![Q[\mathbf{rec}X.Q/X]]\!]_f = [\![Q]\!]_f[\mathbf{rec}X.[\![Q]\!]_f/X]$$

for all renaming functions f for Q and $\mathbf{rec}X.Q$.

Proof. Immediate by the definition of encoding and substitution. □

Proof of Theorem 6.3.15 for Recursive Processes: Let P be a session process, Θ, Γ session typing contexts, and f a renaming function for P such that $[\![\Theta]\!]_f; [\![\Gamma]\!]_f \vdash [\![P]\!]_f$. Then, the following statements hold.

1. If $P \to P'$, then $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$.
2. If $[\![P]\!]_f \to Q$, then there are $P', \mathcal{E}[\cdot]$, such that $\mathcal{E}[P] \to \mathcal{E}[P']$ and $Q \hookrightarrow [\![P']\!]_{f'}$, and either $f' = f$ or $f' = f, \{x, y \mapsto c\}$ for x, y such that $(\boldsymbol{\nu} xy)$ appears in $\mathcal{E}[P]$.

Proof. Since by assumption $[\![\Theta]\!]_f; [\![\Gamma]\!]_f \vdash [\![P]\!]_f$, then by Theorem 6.3.10 for recursive processes given above, we have that $\Theta; \Gamma \vdash P$.

1. The proof is done by induction on the derivation $P \to P'$. The only case to be considered is when (R-Rec) is applied.

$$\frac{P[\mathbf{rec}X.P/X] \to P'}{P \to P'}$$

   By induction hypothesis we have that $[\![P[\mathbf{rec}X.P/X]]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$. By applying (R$\pi$-Rec) and (R$\pi$-Struct) we conclude that $[\![P]\!]_f \to\!\!\hookrightarrow [\![P']\!]_f$.

2. The proof is done by induction on the derivation for $[\![P]\!]_f \to Q$. The case to be considered is when (Rπ-Rec) is applied. By the premise of (Rπ-Rec) and Lemma 10.5.1 we have that $[\![P]\!]_f[\mathbf{rec}X.[\![P]\!]_f/X] \to Q$. We conclude by induction hypothesis, by (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 encoded as a linear channel type, which in turn carries a linear channel. In order to satisfy linearity, a fresh channel is created at every step of communication and is sent along together with the original payload. This fresh channel is then used to continue the rest of the communication. The continuation-passing of channels simulates the structure of session types. 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

$$\begin{aligned}[\![x!\langle v\rangle.P]\!]_f &\triangleq (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \\ [\![x \triangleleft l_j.P]\!]_f &\triangleq (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}}\end{aligned}$$

One can argue that there is an overhead in creating at every output a new channel for the continuation of the communication. In the following, we show that the transmission of new channels is not necessary. What are going to modify the encoding in order to mimic a session type even more faithfully. In this optimised approach we reuse the same channel. But then, since channel variables have linear types, doing so would violate linearity. In order to overcome this problem, we modify the typing rules 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:

$$[\![x!\langle v\rangle.P]\!]_f \triangleq (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \tag{11.1}$$

The optimised encoding is as follows:

$$[\![x!\langle v\rangle.P]\!] \triangleq x!\langle [\![v]\!]_f, x\rangle.[\![P]\!] \tag{11.2}$$

In order to overcome the linearity violation, we modify the type system by introducing the following typing rule for the output:

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

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:

$$[\![x \triangleleft l_j.P]\!]_f \triangleq (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f,\{x\mapsto c\}} \tag{11.3}$$

The optimised encoding is as follows:

$$[\![x \triangleleft l_j.P]\!] \triangleq x!\langle l_j\_x\rangle.[\![P]\!] \tag{11.4}$$

By using (Tπ-LVal)

$$\frac{\Gamma, x : \ell_\alpha\,[\widetilde{S}] \vdash x : \ell_\alpha\,[\widetilde{S}] = T_j \quad j \in I}{\Gamma, x : \ell_\alpha\,[\widetilde{S}] \vdash l_j\_x : \langle l_i\_T_i\rangle_{i\in I}}\ (\mathrm{T}\pi\text{-LVal})$$

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 $\Delta \rhd P \simeq^{\mathsf{c}} Q$, 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 present in the following the result on typed strong barbed congruence of the encoding of the output and the selection processes.

Output Let

$$\Gamma \triangleq x : \ell_{\mathsf{o}}\,[T, \ell_\alpha\,[\widetilde{S}]], v : T, \Gamma'$$

$$P \triangleq (\nu c)x!\langle v, c\rangle.[\![R]\!]_{f,\{x \mapsto c\}}$$

$$Q \triangleq x!\langle v, x\rangle.[\![R]\!]$$

$\Gamma' \vdash [\![R]\!]_{f,\{x\mapsto c\}}$ and $\Gamma', x : \ell_{\overline{\alpha}}\,[\widetilde{S}] \vdash [\![R]\!]$.
Then

$$\Gamma \rhd P \simeq^{\mathsf{c}} Q \tag{11.5}$$

Selection Let

$$\Gamma \triangleq x : \ell_{\mathsf{o}}\,[\langle l_i \_ T_i \rangle_{i\in I}], \Gamma'$$

$$P \triangleq (\nu c)x!\langle l_j \_ c\rangle.[\![R]\!]_{f,\{x\mapsto c\}}$$

$$Q \triangleq x!\langle l_j \_ x\rangle.[\![R]\!]$$

$\Gamma' \vdash [\![R]\!]_{f,\{x\mapsto c\}}$ and $\Gamma', x : \overline{T_j} \vdash [\![R]\!]$.
Then

$$\Gamma \rhd P \simeq^{\mathsf{c}} Q \tag{11.6}$$

Above, P is the encoding of output (respectively selection) by following the rules in Fig. 6.2 and Q is the encoding of output (respectively selection) by following the rules in Section 11.1. By using the typing context Γ for output (respectively selection), (11.5) and (11.6) follow by Theorem 4.5.6 and the Lemma 11.2.6.

Conclusions, Related and Future Work for Part II and III

In Part II and III of this thesis 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 [72, 101] assure that a channel is used exactly once for communication.

Variant types [99, 101] are labelled disjoint union of types, where the order of components does not matter and labels are all distinct. 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 optimisation 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 enforced 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 wrt extensions like subtyping, polymorphism, higher-order communication and recursion. This allows us to derive properties for these new features by exploiting the encoding and the 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.

Related and Future Work The idea of encoding session types into π-calculus linear types is not new. Kobayashi [71] was the first to propose such an encoding, but he did not prove any properties and did not investigate its robustness; moreover, as certain key features of session types do not clearly show up in the encoding, like duality, the faithfulness of the encoding was unclear. Later on, Dardha et al. [32] studied such encoding by showing its soundness and completeness wrt typing and reduction. Advanced features, such as subtyping, polymorphism and higher-order communication are introduced to prove the robustness of the encoding. In [30], the author investigates recursion. The interesting part of [30] is the use of the *complement* function as opposed to the inductive duality function $\bar{\cdot}$, since the latter does not commute with the unfolding of recursive session types, as stated in Chapter 10 and [8, 9].

Demangeon and Honda [35] 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 wrt 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.

Variant types are essential type constructs 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]. This paper presents a type system for the π-calculus that corresponds to the standard sequent calculus proof system for dual intuitionistic linear logic (DILL). It gives 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 [60] 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 [46] 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 future work on the encoding we want to investigate the multiparty session types [59]. In a nutshell, multiparty session types differ from dyadic session types in the interleaving of channels used among different participants. 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 [59].

Part IV

Progress of Communication

Introduction to Part IV

Progress is a fundamental characteristic of safe programs. 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 computational models like the λ-calculus [6] and it is typically analysed in closed terms using type systems. We have only recently begun to research its meaning in computational models for concurrency and distributed systems.

The most basic property related to progress in concurrency is that of *deadlock freedom*: "a process is deadlock-free if it can always reduce until it eventually terminates" [69–71]. Said differently, a communication will eventually succeed unless the whole process diverges. Observe that a deadlock-free process can diverge, and more interestingly, some subprocesses can get stuck. For instance, consider the following process:

$$P \;=\; (\nu x)(x?(y).\mathbf{0} \mid \Omega)$$

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 deadlock freedom, *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 [68, 69, 71]. Said differently, a communication will eventually succeed even if the whole process diverges. Different techniques have been proposed for guaranteeing deadlock freedom and lock freedom, mostly based on type systems for the π-calculus [66, 68–71, 74].

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 [37, 88, 91]. A recent line of work [10, 20, 27, 39] 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 a suitable context. This notion has been analysed when considering

only the behaviour of each single channel in isolation [32, 109] and of the whole system [10, 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 reduces and 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 pose the research question:

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. This part of the thesis is based on [19]. In the following we list the major contributions and give the roadmap to 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 session type system given in [109]. Our first result is that for well-typed closed processes, 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). We then focus on open precesses and we prove that it is possible to relate progress to lock freedom even for 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 for well-typed processes in the π-calculus with sessions, *progress is a compositional form 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.

A static analysis for progress in the π- calculus with sessions Based on the fact 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 a static analysis for lock freedom in the standard π- calculus. We present how Kobayashi's type system for lock freedom [68] can be reused to check whether a process has progress. Using Kobayashi's type system for progress analysis yields a new technique, which is more accurate than previous techniques in the literature.

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 on the syntax of types and typing rules for guaranteeing the lock freedom property. Chapter 13 gives

a background on the π-calculus with sessions which reports a few modifications wrt 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 sessions, by relating it to the notion of lock freedom for sessions. In addition it gives a static way for checking progress 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 [68]. The syntax of terms and the operational semantics are the same as in Chapter 4. For simplicity, we recall them in this chapter. We then introduce the *usage types* and give the type system with usage types, which guarantees lock freedom.

12.1 Syntax

The syntax of terms for the standard π-calculus is the same as in Section 4.1 with the addition of recursive term constructs given in Section 10.1. We present the complete syntax of terms 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 $\tilde{v}$ is transmitted and a tuple of placeholders $\tilde{y}$ is used, respectively; conditional **if** v **then** P **else** Q; other standard constructs like parallel composition $P \mid Q$, inaction $\mathbf{0}$ and restriction $(\nu x)P$; the **case** process and term constructs for recursion, which 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, which is simply a labelled value l_v.

12.2 Semantics

We give some of the reduction rules for the standard π- calculus in Fig. 12.2. These rules are presented in Section 4.2 and in Section 10.2. We do not give the reduction rules for context closure under composition, restriction and structural congruence, which are standard and are given in Section 4.2.

$P, Q ::=$	$x!\langle\tilde{v}\rangle.P$	(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: repeated

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

$$\text{(R}\pi\text{-Case)} \quad \mathbf{case}\ l_j\_v\ \mathbf{of}\ \{l_i\_x_i \triangleright P_i\}_{i\in I} \to P_j[v/x_j] \quad j \in I$$

$$\text{(R}\pi\text{-Rec)} \quad \frac{P[\mathbf{rec}X.P/X] \to P'}{\mathbf{rec}X.P \to P'}$$

Figure 12.2: Semantics of the standard π-calculus: repeated

$U ::=$	$\mathsf{i}_c^o.U$	(used in input)	$\emptyset$	(not usable)
	$\mathsf{o}_c^o.U$	(used in output)	$(U_1 \mid U_2)$	(used in parallel)
	$\mathbf{t}$	(usage variable)	$\mu\mathbf{t}.U$	(recursive usage)
$T ::=$	$[\widetilde{T}]mU$	(channel types)	$\langle l_i \_ T_i \rangle_{i \in I}$	(variant type)
	`Bool`	(boolean type)		

Figure 12.3: Syntax of usage types

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) states that 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 $j \in I$. Rule (Rπ-Rec) states that a recursive process **rec**$X.P$ reduces to a process P' if process P where X is substituted by the recursion process **rec**$X.P$, reduces to the same P'.

12.3 π-Types for Lock Freedom

The syntax of usage types is given in Fig. 12.3 and is inspired by Kobayashi's works on lock freedom [68, 70, 71]. Let o, c range over natural numbers, α range over actions, being *only* 'i' input or 'o' output. Let U range over usages and T over types. Let mU be either ℓU, for a linear usage U, or simply U for an unrestricted usage U. Usages U are used to build channel types. A usage can be an empty usage $\emptyset$, which denotes a channel that cannot be used at all for communication; we will often omit it when it is not necessary. Usage $\mathsf{i}_c^o.U$ describes a channel used once for input and then used according to U. Symmetrically, usage $\mathsf{o}_c^o.U$ describes a channel used once for output and then used according to U. We will comment on o and c numbers in the following. Usage $U_1 \mid U_2$ describes a channel used according to U_1 and U_2 possibly 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 mU to transmit a sequence of values of types $\widetilde{T}$. Notice that, the usages describe a channel used in structured way, differently from the linear types presented in Fig. 4.5 and similar to session types. However, the main difference wrt session types is that the carried type associated to a usage is always the same $\widetilde{T}$. A type can also be variant type $\langle l_i \_ T_i \rangle_{i \in I}$ or a ground type like `Bool`, or other type constructors, as stated in Section 4.3.

The annotations o and c in the actions are called *obligation level* and *capability level* of that action, respectively. We will commonly refer to them as *tags* or

attributes. They are thought of and defined as abstract representations of *time tags* or *reduction steps*. 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 it take for the action to find its co-action. By citing Kobayashi's works [68, 70, 71], their relation may be described as:

- An obligation of level n must be fulfilled by using only capabilities of level *less than n*. Said differently, an action of obligation n must be prefixed by actions of capabilities less than n.
- For an action with capability of level n, there must exist a co-action with obligation of level *less than or equal to n*.

It is important to notice that in the original works [68, 70, 71], tags may also range over '∞', which 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 '∞' and require that tags range over natural numbers. We illustrate the usage of tags with two simple examples, given in the following. The first example shows how tags work on a deadlocked process and the second example shows how tags work on a deadlock-free but 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 x has usage $\mathsf{i}_{c_1}^{o_1} \mid \mathsf{o}_{c_2}^{o_2}$ and y has usage $\mathsf{i}_{c_3}^{o_3} \mid \mathsf{o}_{c_4}^{o_4}$. 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-hand side 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: $(\boldsymbol{\nu} x)(x?(w) \mid (\boldsymbol{\nu} y)(y!\langle x\rangle \mid y?(z).\mathbf{rec}X.(y!\langle z\rangle \mid y?(z).X)))$ This process is never stuck, because of infinite sendings, however the first input on x will be never executed, thus making the process livelocked. Suppose y sends x having usage $\mathsf{o}_{c_1}^{o_1}$. A message sent on y is received by its counterpart in $o_3 (> 0)$ steps. The subprocess $y?(z).\mathbf{rec}X.(y!\langle z\rangle \mid y?(z).X)$ receives z of usage $\mathsf{o}_{c_1}^{o_1}$ 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. Then $o_1 + o_3 \leq o_1$, which is a contradiction.

12.4 π-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 [68, 71] with **case** process and variant values. We start by giving some auxiliary definitions and operations on types and typing contexts taken from [68].

Definition 12.4.1 (Normal Form)**.** A process is in *normal form* if it a restriction of parallel composition, namely $(\nu\widetilde{x})(R_1 \mid \ldots \mid R_n)$ and all variables in $\widetilde{x}$ are different from each other and from the free ones in the process.

Definition 12.4.2 (Reduction Sequence)**.** A set of processes $\{P_i\}_{i\in I}$ for $I \subseteq Nat$ is called a *reduction sequence*, if $P_{i-1} \rightarrow P_i$ for all $i \in I \setminus \{0\}$.
A reduction sequence is *normal* if i) for all $i \in I$, 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 12.4.3 (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 $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_j} \equiv (\nu\widetilde{x_j})(x!\langle v\rangle.Q \mid x?(z).Q_j \mid R_j)$, for all 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 $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_j} \equiv (\nu\widetilde{x_j})(x?(z).Q \mid x!\langle v\rangle.Q_j \mid R_j)$, for all 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 12.4.4 (Lock Freedom for Standard π-Calculus)**.** A process P_0 in normal form is *lock-free* under fair scheduling, if for any fair reduction sequence $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots$ the following hold.

1. if $P_i \equiv (\nu\widetilde{x})(x!\langle v\rangle.Q \mid R)$ some $i \geq 0$, then there exists $n \geq i$ such that $P_n \equiv (\nu\widetilde{x'})(x!\langle v\rangle.Q \mid x?(z).R_1 \mid R_2)$ and $P_{n+1} \equiv (\nu\widetilde{x'})(Q \mid R_1[v/z] \mid R_2)$.

2. if $P_i \equiv (\nu\widetilde{x})(x?(z).Q \mid R)$ for some $i \geq 0$, then there exists $n \geq i$ such that $P_n \equiv (\nu\widetilde{x'})(x?(z).Q \mid x!\langle v\rangle.R_1 \mid R_2)$ and $P_{n+1} \equiv (\nu\widetilde{x'})(Q[v/z] \mid R_1 \mid R_2)$.

Remark 12.4.5. Note that in the original work [68], 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 eventually communicate.

The unary operation $\uparrow^t$ applied to a usage U lifts its obligation level *up to t*, and is inductively defined as follows:

$$\begin{array}{rcl} \uparrow^t \emptyset & \triangleq & \emptyset \\ \uparrow^t \alpha_c^o.U & \triangleq & \alpha_c^{max(o,t)}.U \\ \uparrow^t (U_1 \mid U_2) & \triangleq & (\uparrow^t U_1 \mid \uparrow^t U_2) \\ \uparrow^t \mathbf{t} & \triangleq & \mathbf{t} \\ \uparrow^t \mu\mathbf{t}.U & \triangleq & \mu\mathbf{t}.\uparrow^t U \end{array}$$

The $\uparrow^t$ operator is extended to types and typing contexts in the expected way and is given in the following. It is undefined otherwise.

$$\begin{array}{rcl} \uparrow^t [\widetilde{T}]mU & \triangleq & [\widetilde{T}]m \uparrow^t U \\ (\uparrow^t \Gamma)(x) & \triangleq & \uparrow^t (\Gamma(x)) \end{array}$$

The composition operation on types, denoted $\mid$, is based on the composition of usages and is defined as follows:

$$\begin{array}{rcll} U_1[\widetilde{T}] \mid U_2[\widetilde{T}] & \triangleq & (U_1 \mid U_2)[\widetilde{T}] & \\ T \mid T & \triangleq & T & \text{if } \mathsf{un}(T) \\ T \uplus S & \triangleq & \mathtt{undef} & \text{otherwise} \end{array}$$

Its generalisation to typing contexts, denoted $(\Gamma_1 \mid \Gamma_2)(x)$, is as expected and is defined in the following:

$$x : T \in \Gamma_1 \mid \Gamma_2 \text{ iff } \begin{cases} x : T_1 \in \Gamma_1 \text{ and } x : T_2 \in \Gamma_2 \\ \qquad\qquad \text{and } T = T_1 \mid T_2 \\ x : T \in \Gamma_1 \text{ and } x \notin \mathsf{dom}(\Gamma_2) \\ x : T \in \Gamma_2 \text{ and } x \notin \mathsf{dom}(\Gamma_1) \end{cases}$$

Notice that the parallel operator $\mid$ is defined similarly to the combination operator $\uplus$ given in Section 4.4. As a result 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 † is defined on typing contexts. $\Delta = x : [T]\ \alpha_c^o \dagger \Gamma$ is such that the following holds:

$$\mathsf{dom}(\Delta) = \{x\} \cup \mathsf{dom}(\Gamma)$$
$$\Delta(x) = \begin{cases} [\widetilde{T}]\alpha_c^o.U & \text{if } \Gamma(x) = [\widetilde{T}]\ U \\ [\widetilde{T}]\alpha_c^o & \text{if } x \notin dom(\Gamma) \end{cases}$$
$$\Delta(y) = \uparrow^{c+1} \Gamma(y) \quad \text{if } y \neq x$$

The final required notion is that of a *reliable usage*. Intuitively, 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 having obligation level at most c. The following definitions are taken from [68, 70].

Definition 12.4.6. Let U be a usage. The input and output *obligation levels* (resp. *capability levels*) of U, written $\mathsf{ob}_{\mathsf{i}}(U)$ and $\mathsf{ob}_{\mathsf{o}}(U)$ (resp. $\mathsf{cap}_{\mathsf{i}}(U)$ and $\mathsf{cap}_{\mathsf{o}}(U)$), are defined as follows:

$$\mathsf{ob}_\alpha(\alpha_c^o.U) = o \qquad \mathsf{ob}_\alpha(U_1 \mid U_2) = min(\mathsf{ob}_\alpha(U_1), \mathsf{ob}_\alpha(U_2))$$
$$\mathsf{cap}_\alpha(\alpha_c^o.U) = c \qquad \mathsf{cap}_\alpha(U_1 \mid U_2) = min(\mathsf{cap}_\alpha(U_1), \mathsf{cap}_\alpha(U_2))$$

The definition of reliable usages depends on a reduction relation on usages, noted $U \to U'$. Intuitively, $U \to U'$ means that if a channel of usage U is used for communication, then after the communication occurs, the channel should be used according to usage U'. Thus, e.g., $\mathsf{i}_c^o.U_1 \mid \mathsf{i}_{c'}^{o'}.U_2$ reduces to $U_1 \mid U_2$.

Definition 12.4.7 (Reliability)**.** We write $\mathsf{con}_\alpha(U)$ when $\mathsf{ob}_{\overline{\alpha}}(U) \leq \mathsf{cap}_\alpha(U)$. We write $\mathsf{con}(U)$ when $\mathsf{con}_{\mathsf{i}}(U)$ and $\mathsf{con}_{\mathsf{o}}(U)$ hold. Usage U is *reliable*, noted $rel(U)$, if $\mathsf{con}(U')$ holds for all U' such that $U \to^* U'$.

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 $\vdash$ in order to distinguish the type system for lock freedom from the type system for the linear π-calculus given in Section 4.4.

The typing rules for lock-freedom are given in Fig. 12.4. (LF-Var), (LF-Val) and (LF-LVal) are the same as the corresponding ones given in Section 4.4, where linear types are used. Rules (LF-Inact), (LF-If),(LF-Par) and (LF-Case) are the same as the corresponding ones in Section 4.4, but 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. The obligations of the other channels in Γ are raised by using the operator †, because the actions inside process P are prefixed by the input action and will thus become available one step later. Rule (LF-Out) states that the output process

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

$$\frac{\Theta; \Gamma \vdash_{\mathrm{LF}} v : T}{\Theta; \Gamma \vdash_{\mathrm{LF}} l\_v : \langle l\_T \rangle}\ (\text{LF-LVal}) \qquad \frac{\mathsf{un}(\Gamma)}{\Theta; \Gamma \vdash_{\mathrm{LF}} \mathbf{0}}\ (\text{LF-Inact})$$

$$\frac{\Theta; \Gamma_1 \vdash_{\mathrm{LF}} v : \mathtt{Bool} \quad \Theta; \Gamma_2 \vdash_{\mathrm{LF}} P \quad \Theta; \Gamma_2 \vdash_{\mathrm{LF}} Q}{\Theta; \Gamma_1 \mid \Gamma_2 \vdash_{\mathrm{LF}} \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q}\ (\text{LF-If})$$

$$\frac{\Theta; \Gamma, \widetilde{y} : \widetilde{T} \vdash_{\mathrm{LF}} P}{\Theta; x : [\widetilde{T}]\, m\, \mathsf{i}_c^0 \ \dagger\ \Gamma \vdash_{\mathrm{LF}} x?(\tilde{y}).P}\ (\text{LF-In}) \qquad \frac{\Theta; \Gamma_1 \vdash_{\mathrm{LF}} \tilde{v} :\uparrow \widetilde{T} \quad \Theta; \Gamma_2 \vdash_{\mathrm{LF}} P}{\Theta; x : [\widetilde{T}]\, m\, \mathsf{o}_c^0 \dagger (\Gamma_1 \mid \Gamma_2) \vdash_{\mathrm{LF}} x!\langle \tilde{v} \rangle.P}\ (\text{LF-Out})$$

$$\frac{\Theta; \Gamma_1 \vdash_{\mathrm{LF}} P \quad \Theta; \Gamma_2 \vdash_{\mathrm{LF}} Q}{\Theta; \Gamma_1 \mid \Gamma_2 \vdash_{\mathrm{LF}} P \mid Q}\ (\text{LF-Par}) \qquad \frac{\Theta; \Gamma, x : [\widetilde{T}]\, m_U \vdash_{\mathrm{LF}} P \quad rel(U)}{\Theta; \Gamma \vdash_{\mathrm{LF}} (\nu x)P}\ (\text{LF-Res})$$

$$\frac{\Theta; \Gamma_1 \vdash_{\mathrm{LF}} v : \langle l\_T \rangle_{i \in I} \quad \Theta; \Gamma_2, x_i : T_i \vdash_{\mathrm{LF}} P_i \quad \forall i \in I}{\Theta; \Gamma_1 \mid \Gamma_2 \vdash_{\mathrm{LF}} \mathbf{case}\ v\ \mathbf{of}\ \{l_i\_x_i \triangleright P_i\}_{i \in I}}\ (\text{LF-Case})$$

$$\frac{\Theta(X) = \Gamma}{\Theta; \Gamma \vdash X}\ (\text{LF-RecVar}) \qquad \frac{\Theta, X : \Gamma; \Gamma \vdash P}{\Theta; \Gamma \vdash \mathbf{rec} X.P}\ (\text{LF-RecProc})$$

$$\frac{\Theta; \Gamma \vdash v : T \quad T \sim_{\mathsf{type}} S}{\Theta; \Gamma \vdash v : S}\ (\text{LF-EqVal})$$

Figure 12.4: Typing rules for the π-calculus with usage types

$x!\langle\tilde{v}\rangle.P$ is well typed and ready for execution if x is a channel used in output and has obligation level 0. Moreover, the obligation level 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 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 [69, 71], the typing rule for the output process is the only one that differs in 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 next theorems imply that well-typed processes by the type system in Fig. 12.4 are lock-free.

Theorem 12.4.8 (Subject Reduction for Usage Types)**.** If $\Gamma \vdash_{\mathrm{LF}} P$ and $P \to Q$, then $\Gamma' \vdash_{\mathrm{LF}} Q$ for some Γ' such that $\Gamma \to \Gamma'$.

Theorem 12.4.9 (Lock Freedom)**.** If $\emptyset \vdash_{\mathrm{LF}} P$, then $P \to Q$ for some Q.

Corollary 12.4.10. If $\emptyset \vdash_{\mathrm{LF}} P$, then P is lock-free.

Chapter 13

Background on Session Types for Progress

In this chapter we recall the π-calculus with session types given in Chapter 5. We male some modifications to the syntax of types and terms in order to accommodate the progress property.

13.1 Syntax

The syntax of terms of the π- calculus with sessions is presented in 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 $\mathbf{0}$. Process $(\boldsymbol{\nu} xy)P$ is the restriction of co-variables and X and $\mathbf{rec}X.P$ model recursion. Branching is the standard one $x \triangleright \{l_i : P_i\}_{i\in I}$ as in Section 5.1. 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, as we will show in the next sections.

13.2 Semantics

The reduction rules are the same as the ones given in Section 5.2. We give some of the most important ones in Fig. 13.2. Rules (R-Com), (R-Sel) and (R-Rec), were explained in details in the previous chapters. Rule (R-SelNorm) is a *selection normalisation*, stating that the generalised selection process $x \triangleleft \{l_i : P_i\}_{i\in I}$ reduces to $x \triangleleft l_j.P$ being j one of the indexes 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.

$$
\begin{array}{rll}
P, Q ::= & x!\langle v\rangle.P & \text{(output)} \\
& x?(y).P & \text{(input)} \\
& x \triangleleft \{l_i : P_i\}_{i\in I} & \text{(selection)} \\
& x \triangleright \{l_i : P_i\}_{i\in I} & \text{(branching)} \\
& \textbf{if } v \textbf{ then } P \textbf{ else } Q & \text{(conditional)} \\
& P \mid Q & \text{(composition)} \\
& \mathbf{0} & \text{(inaction)} \\
& (\boldsymbol{\nu} xy)P & \text{(session restriction)} \\
& X & \text{(process variable)} \\
& \textbf{rec}X.P & \text{(recursive process)} \\
v ::= & x & \text{(variable)} \\
& \texttt{true} \mid \texttt{false} & \text{(boolean values)}
\end{array}
$$

Figure 13.1: Syntax of the π-calculus with sessions: updated

$$
\begin{array}{rc}
\text{(R-Com)} & (\boldsymbol{\nu} xy)(x!\langle v\rangle.P \mid y?(z).Q \mid R) \to (\boldsymbol{\nu} xy)(P \mid Q[v/z] \mid R) \\
\text{(R-Sel)} & (\boldsymbol{\nu} xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I} \mid R) \to (\boldsymbol{\nu} xy)(P \mid P_j \mid R) \quad j \in I \\
\text{(R-SelNorm)} & x \triangleleft \{l_i : P_i\}_{i\in I} \to x \triangleleft l_j.P_j \quad j \in I \\
\text{(R-Rec)} & \dfrac{P[\textbf{rec}X.P/X] \to P'}{\textbf{rec}X.P \to P'}
\end{array}
$$

Figure 13.2: Semantics of the π-calculus with sessions: updated

$$
\begin{array}{rll}
q ::= & \mathsf{lin} \mid \mathsf{un} & \text{(qualifiers)} \\
p ::= & !T.U & \text{(send)} \\
 & ?T.U & \text{(receive)} \\
 & \oplus\{l_i : T_i\}_{i \in I} & \text{(select)} \\
 & \&\{l_i : T_i\}_{i \in I} & \text{(branch)} \\
T ::= & q\ p & \text{(qualified pretype)} \\
 & \mathtt{end} & \text{(termination)} \\
 & \mathtt{Bool} & \text{(boolean type)} \\
 & \mathbf{t} & \text{(type variable)} \\
 & \mu\mathbf{t}.T & \text{(recursive type)}
\end{array}
$$

Figure 13.3: Syntax of session types: updated

13.3 Session Types

The syntax of session types is given in Fig. 13.3 and is an extension of the syntax of session types given in Section 5.3, since it includes $\mathbf{t}$ and $\mu\mathbf{t}.T$. Recursive types are needed not only to model infinite behaviour of processes, but also to be able to use unrestricted types, as we explained in Section 5.4.

Qualifiers are lin (for linear) or un (for unrestricted) and have the usual meaning. A type can be qp, the qualified pretype; `end`, the type of the terminated channel where no communication can take place further; `Bool`, the type of boolean values; or recursive types and recursive variables. A pretype can be $!T.U$ or $?T.U$, which respectively, is the type 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.

13.4 Session Typing Rules

The typing judgements now have the form $\Theta; \Gamma \vdash v : T$, for values and $\Theta; \Gamma \vdash P$, for processes, such that:

$$\Gamma ::= \emptyset \mid \Gamma, x : T \qquad\qquad \Theta ::= \emptyset \mid \Theta, X : \Gamma$$

The typing context Γ is the same as in Section 5.4 and the typing context Θ is used to accommodate recursive processes.

The typing rules are given in Fig. 13.4. The differences wrt the typing rules in Section 5.4 is the presence of Θ and rule (T-Sel) which 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. For completeness, we present all the updated typing rules.

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

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

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

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

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

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

$$\frac{\Theta; \Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I} \quad \Theta; \Gamma_2 + x : T_i \vdash P_i \quad \forall i \in I}{\Theta; \Gamma_1 \circ \Gamma_2 \vdash x \triangleleft \{l_i : P_i\}_{i \in I}}\ (\text{T-SEL})$$

$$\frac{\Theta(X) = \Gamma}{\Theta; \Gamma \vdash X}\ (\text{T-RECVAR}) \qquad \frac{\Theta, X : \Gamma; \Gamma \vdash P}{\Theta; \Gamma \vdash \textbf{rec} X.P}\ (\text{T-RECPROC})$$

$$\frac{\Theta, \Gamma \vdash v : T \quad T \sim_{\mathsf{type}} S}{\Theta, \Gamma \vdash v : S}\ (\text{T-EQVAL})$$

Figure 13.4: Typing rules for the π-calculus with sessions: updated

Chapter 14

Progress as Compositional Lock Freedom

In this chapter we present our main results about progress and lock freedom in the π-calculus with session types. We start by giving the definition of lock freedom for session communication, which is an adaptation of the corresponding definition in the standard π-calculus and we give a relation between lock freedom and the notion of progress, the latter being already defined for sessions [10, 20, 27].

14.1 Lock Freedom for Sessions

In order to formally define lock freedom for session communication, we need the definitions of normal form and reduction sequence. These definitions are the same as the ones for the standard π-calculus which are given in Section 12.4.

We now give the definition of fair reduction sequence, which is an adaptation of Definition 12.4.3 given Section 12.4.

Definition 14.1.1 (Fair Reduction Sequence for Sessions)**.** 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 $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_j} \equiv (\nu \widetilde{x_j y_j})(x!\langle v\rangle.Q \mid y?(z).Q_j \mid R_j)$, for all n_j, then there exists $n \geq n_0$, such that $P_n \equiv (\nu \widetilde{xy})(x!\langle v\rangle.Q \mid y?(z).Q' \mid R')$ and $(\nu \widetilde{xy})(Q \mid Q'[v/z] \mid R') \equiv P_{n+1}$.

2. If there exists an infinite increasing sequence $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_j} \equiv (\nu \widetilde{x_j y_j})(x?(z).Q \mid y!\langle v\rangle.Q_j \mid R_j)$, for all n_j, then there exists $n \geq n_0$, such that $P_n \equiv (\nu \widetilde{xy})(x?(z).Q \mid y!\langle v\rangle.Q' \mid R')$ and $(\nu \widetilde{xy})(Q[v/z] \mid Q' \mid R') \equiv P_{n+1}$.

3. If there exists an infinite increasing sequence $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_j} \equiv (\nu \widetilde{x_j y_j})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft \{l_i : Q_i\}_{i\in I} \mid R_j)$, for all n_j, then there exists $n \geq n_0$, such that $P_n \equiv (\nu \widetilde{xy})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft \{l_i : Q'_i\}_{i\in I} \mid R')$ and $(\nu \widetilde{xy})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft l_k.Q'_k \mid R') \equiv P_{n+1}$ for some $k \in I$ and $(\nu \widetilde{xy})(P_k \mid Q'_k \mid R') \equiv P_{n+2}$.

4. If there exists an infinite increasing sequence $n_0 < n_1 < \ldots$ of natural numbers such that $P_{n_j} \equiv (\nu \widetilde{x_j y_j})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R_j)$, for all n_j, then there exists $n \geq n_0$, such that $P_n \equiv (\nu \widetilde{xy})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid y \triangleright \{l_i : Q'_i\}_{i\in I} \mid R')$ and $(\nu \widetilde{xy})(x \triangleleft l_k.P_k \mid y \triangleright \{l_i : Q'_i\}_{i\in I} \mid R') \equiv P_{n+1}$ for some $k \in I$ and $(\nu \widetilde{xy})(P_k \mid Q'_k \mid R') \equiv P_{n+2}$.

We are now 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.

In order to define lock freedom, we assume, as in the original work, a *strongly fair scheduling* [28, 43], which intuitively means that every process enabled to participate in a communication infinitely many times, will eventually do so.

Definition 14.1.2 (Lock Freedom for Sessions)**.** A process P_0 is *lock-free* under fair scheduling, if for any fair reduction sequence $P_0 \to P_1 \to P_2 \to \ldots$ we have the following:

1. if $P_i \equiv (\nu \widetilde{xy})(x!\langle v\rangle.Q \mid R)$ (for $i \geq 0$), implies that there exists $n \geq i$ such that $P_n \equiv (\nu \widetilde{x'y'})(x!\langle v\rangle.Q \mid y?(z).R_1 \mid R_2)$ and $P_{n+1} \equiv (\nu \widetilde{x'y'})(Q \mid R_1[v/z] \mid R_2)$;

2. if $P_i \equiv (\nu \widetilde{xy})(x?(z).Q \mid R)$ for some $i \geq 0$, there exists $n \geq i$ such that $P_n \equiv (\nu \widetilde{x'y'})(x?(z).Q \mid y!\langle v\rangle.R_1 \mid R_2)$ and $P_{n+1} \equiv (\nu \widetilde{x'y'})(Q[v/z] \mid R_1 \mid R_2)$;

3. if $P_i \equiv (\nu \widetilde{xy})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid R)$ for some $i \geq 0$, there exists $n \geq i$ such that $P_n \equiv (\nu \widetilde{x'y'})(x \triangleleft \{l_i : P_i\}_{i\in I} \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R')$ and $P_{n+2} \equiv (\nu \widetilde{x'y'})(P_j \mid Q_j \mid R')$ for $j \in I$;

4. if $P_i \equiv (\nu \widetilde{xy})(x \triangleright \{l_i : P_i\}_{i\in I} \mid R)$ for some $i \geq 0$, there exists $n \geq i$ such that $P_n \equiv (\nu \widetilde{x'y'})(x \triangleright \{l_i : P_i\}_{i\in I} \mid y \triangleleft \{l_i : Q_i\}_{i\in I} \mid R')$ and $P_{n+2} \equiv (\nu \widetilde{x'y'})(P_j \mid Q_j \mid R')$ for $j \in I$.

Notice that in the definition of lock freedom for sessions the reduction sequence goes from P_n to P_{n+2}. This means that P_{n+1} is obtained by a selection normalisation.

14.2 Progress for Sessions

The progress property has been studied for the session π- calculus by adopting cumbersome definitions and type systems. Progress is checked for closed and 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. Intuitively, a *characteristic process* is the simplest process that can inhabit a type.

Definition 14.2.1 (Characteristic Process)**.** Given a session type T, its *characteristic process* $[\![T]\!]_f^x$ is inductively defined on the structure of T as follows:

$$
\begin{array}{rl}
(\textsc{inVal}) & [\![q?\texttt{Bool}.U]\!]_f^x = x?(y).[\![U]\!]_f^x \\
(\textsc{outVal}) & [\![q!\texttt{Bool}.U]\!]_f^x = x!\langle\texttt{true}\rangle.[\![U]\!]_f^x \\
(\textsc{inSess}) & [\![q'?(qp).U]\!]_f^x = x?(y).([\![U]\!]_f^x \mid [\![qp]\!]_f^y) \\
(\textsc{outSess}) & [\![q'!(qp).U]\!]_f^x = (\nu zw)(x!\langle z\rangle.([\![U]\!]_f^x \mid [\![\overline{qp}]\!]_f^w)) \\
(\textsc{inSum}) & [\![q\&\{l_i : (q_i p_i)_i\}_{i\in I}]\!]_f^x = x \triangleright \{l_i : [\![q_i p_i]\!]_f^x\}_{i\in I} \\
(\textsc{outSum}) & [\![q \oplus \{l_i : (q_i p_i)_i\}_{i\in I}]\!]_f^x = x \triangleleft \{l_i : [\![q_i p_i]\!]_f^x\}_{i\in I} \\
(\textsc{end}) & [\![\texttt{end}]\!]_f^x = \mathbf{0} \\
(\textsc{recVar}) & [\![\mathbf{t}]\!]_f^x = f(\mathbf{t}) \\
(\textsc{rec}) & [\![\mu\mathbf{t}.T]\!]_f^x = \mathbf{rec}X.[\![T]\!]_{f,\{\mathbf{t}\mapsto X\}}^x
\end{array}
$$

Our definition of characteristic process is an extension of the original definition given in [20], with recursive types and variables. Moreover, (outSum) is more general and accurate than in [20], since it uses the new selection process.

We now introduce the notion of *evaluation context*, which is an extension of Definition 6.3.12.

Definition 14.2.2 (Evaluation Context)**.** An *evaluation context* is a process with a hole $[\cdot]$ and is produced by the following grammar:

$$\mathcal{E}[\cdot] ::= \quad [\cdot] \mid P \quad | \quad (\nu xy)\mathcal{E}[\cdot] \quad | \quad \mathcal{E}[\cdot] \mid \mathcal{E}[\cdot] \quad | \quad \mathbf{rec}X.\mathcal{E}[\cdot]$$

We now define the notion of *catalyser*, inspired by [27] and we illustrate it with a simple example.

Definition 14.2.3 (Catalyser)**.** A *catalyser* is a context produced by the following grammar:

$$C[\cdot] ::= \ [\cdot] \quad | \quad (\nu xy)\, C[\cdot] \quad | \quad C[\cdot] \mid [\![T]\!]_f^x$$

Example 14.2.4. 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 $T_2 = \oplus\{l_1 : \texttt{end}, l_2 : !\texttt{Bool}.\texttt{end}\}$:

$$\begin{aligned} C[\cdot] &= (\nu wx)(\nu uy)([\cdot] \mid P_1 \mid P_2) \\ P_1 &= x?(z).(z!\langle \texttt{true} \rangle.\mathbf{0} \mid \mathbf{0}) \\ P_2 &= y \triangleleft l_2.y!\langle \texttt{true} \rangle.\mathbf{0} \end{aligned}$$

To conclude, we define the duality or compatibility $\bowtie$ 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 14.2.5 ($\bowtie$)**.** The duality $\bowtie_{\{x,y\}}$ between input and output processes is defined as follows:

$$\begin{aligned} &x!\langle v \rangle.P \bowtie_{\{x,y\}} y?(z).Q \\ &x \triangleleft \{l_i : P_i\}_{i \in I} \bowtie_{\{x,y\}} y \triangleright \{l_i : Q_i\}_{i \in I} \end{aligned}$$

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 14.2.6 (Progress)**.** A process P has *progress* if for all $C[\cdot]$ such that $C[P]$ is well typed, $C[P] \rightarrow^* \mathcal{E}[R]$ (where R is an input or an output) implies that there exist $C'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' such that $C'[\mathcal{E}[R]] \rightarrow^* \mathcal{E}'[R][R']$ 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 the π-calculus with sessions in order to understand their relation. We split this section in 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 ⇒ Progress)**.** Let P be a well-typed closed session process. Then, P is 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, $C[P] \rightarrow^* \mathcal{E}[R]$ where R is an input or an output, and for all $C'[\cdot]$, $\mathcal{E}'[\cdot][\cdot]$ and R' it holds that $C'[\mathcal{E}[R]] \not\rightarrow^* \mathcal{E}'[R][R']$ such that $R \bowtie_{\{x,y\}} R'$ where x and y are such that (νxy) is a restriction in $C'[\mathcal{E}[R]]$. Instead, 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 $C'[\cdot] = [\cdot]$. 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 in $(\nu x'y')$. 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 $(\nu x'y')$ 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 session process. Then, P has progress implies P is lock-free.

Proof. From the definition of progress, 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 (νxy) 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 (νxy). 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 session 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. As expected, differently from the case of 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 14.2.6 but it is not lock-free because it does not respect Definition 14.1.2, 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 a session process such that $\Gamma \vdash P$. Then, the closure of P, denoted as $\mathsf{close}(P)$, is the process $C[P]$ where

$$C[\cdot] = (\boldsymbol{\nu} \widetilde{xy})(\,[\cdot] \mid \prod_{x_i : T_i \in \Gamma} [\![T_i]\!]_f^{y_i})$$

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 $\Gamma = x : \texttt{!Bool.?Bool.end}$. Then, the closure of P is defined as:

$$\mathsf{close}(P) = (\boldsymbol{\nu} xy)([P] \mid y?(z).y!\langle \texttt{true} \rangle.\mathbf{0})$$

The closure procedure $\mathsf{close}(P)$ can also be applied to processes that are already closed, as shown in the following.

Example 14.3.6. Consider the closed process:

$$P \;=\; (\nu xy)(x!\langle \texttt{true}\rangle.\mathbf{0} \mid y?(z).\mathbf{0})$$

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 immediately by the definition of characteristic process and (repeated applications 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 P be a session process and Γ a session typing context such that $\Gamma \vdash P$. Then, P has progress if and only if $\mathsf{close}(P) \to^* \mathcal{E}[R]$ implies there are $\mathcal{E}'[\cdot][\cdot]$ 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 (νxy) is a restriction in $\mathcal{E}[R]$.

Proof. We split the proof into the following two cases.
($\Longrightarrow$) Follows immediately by the definitions of progress and $\mathsf{close}(P)$.
($\Longleftarrow$) 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:

$$\begin{aligned} C[\cdot] &\equiv (\nu \widetilde{xy})([\cdot] \mid Q_1 \mid Q_2) \\ Q_1 &= \textstyle\prod_{x_j : T_j \notin \Gamma} [\![T_j]\!]_f^{y_j} \\ Q_2 &= \textstyle\prod_{x_i : T_i \in \Gamma'} [\![T_i]\!]_f^{y_i} \qquad \text{where } \Gamma' \subseteq \Gamma \end{aligned}$$

Moreover, from the definition of $\mathsf{close}(P)$ we know that:

$$\begin{aligned} \mathsf{close}(P) &= (\nu \widetilde{xy}')(P \mid Q_2 \mid Q_3) \\ Q_3 &= \textstyle\prod_{x_i : T_i \in \Gamma \setminus \Gamma'} [\![T_i]\!]_f^{y_i} \end{aligned}$$

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 $C[P] \to^* \mathcal{E}''[R]$: (i) $P \to^* P'$; (ii) $(\nu \widetilde{xy})Q_1 \to^* (\nu \widetilde{xy})Q_1'$; and finally (iii) $(\nu \widetilde{xy})(P \mid Q_2) \to^* (\nu \widetilde{xy})(P' \mid Q_2')$.

(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 trivially.

(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 $C[P] \rightarrow^* C'[P']$ and $C'[P'] \equiv (\nu \widetilde{xy})(P' \mid Q_1 \mid Q_2')$, since both the catalyser and the process 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 Lemma 14.3.8, we establish that checking the progress property for a session process P is equivalent to checking the progress property for its closure.

Theorem 14.3.9 (Closure Progress ⇔ Progress)**.** Let P be a session process and Γ a session typing context such that $\Gamma \vdash P$. Then, $\mathsf{close}(P)$ has progress if and only if P has progress.

Proof. We split the proof into the following two cases.
(⟸) 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 14.3.4 and Lemma 14.3.8.
(⟹) Follows immediately by Lemma 14.3.8. □

We are finally able to link progress and lock freedom and give our main contribution in the following theorem.

Theorem 14.3.10. (Progress ⇔ Closed Lock Freedom) Let P be a session process and Γ a session typing context such that $\Gamma \vdash P$. 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 some important theoretical results that permit us to use the type system for lock freedom in the standard π-calculus to check progress in the π-calculus with sessions.

We first recall the encoding of session types presented in Section 10.4. In order to encode qualifiers lin and un, and also adopt usages, we perform the following translation from m_α given in Section 10.4, to mU given in Section 12.3.

$$[\![\ell_{\mathsf{i}}]\!] = \ell \mathsf{i}^o_c.\emptyset \qquad [\![\ell_{\mathsf{o}}]\!] = \ell \mathsf{o}^o_c.\emptyset$$
$$[\![\mathsf{i}]\!] = \mathsf{i}^o_c.\emptyset \qquad [\![\mathsf{o}]\!] = \mathsf{o}^o_c.\emptyset$$

We are ready now to give the main result of this part of the thesis.

Theorem 14.4.1 (Progress in Sessions by Encoding)**.** Let P be a session process and Γ a session typing context such that $\Gamma \vdash P$. If $\emptyset \vdash_{\mathtt{LF}} [\![\mathsf{close}(P)]\!]_f$ for some renaming function f for P, then process P has progress.

Proof. Let $\Gamma \vdash P$ and $\emptyset \vdash_{\mathtt{LF}} [\![\mathsf{close}(P)]\!]_f$ for some renaming function f for P. By Corollary 12.4.10 it means that $[\![\mathsf{close}(P)]\!]_f$ is lock-free. By lock freedom for the standard π-calculus, given by Definition 12.4.4, operational correspondence, given by Theorem 6.3.15 and by Definition 14.1.2 on lock freedom for the session π-calculus, it is the case that also $\mathsf{close}(P)$ is lock-free. By Theorem 14.3.10 we have that process P has progress. □

Kobayashi's type system comes with a reference implementation, the tool TyPiCal [106], which tests deadlock freedom and lock freedom for processes in the standard π-calculus. In the light of Theorem 14.4.1, we could use TyPiCal to test the progress property for a session process by checking the lock freedom property for its encoding. Thus, we present in Fig. 14.1 a (pseudo-)algorithm for checking the progress property for a session process.

```
procedure Progress(Γ, P)
  Check Γ ⊢ P
  Build close(P) from Γ
  Encode [[close(P)]]_f = P'
  return TyPiCal(P')
end procedure
```

Figure 14.1: Checking progress with TyPiCal

Conclusions, Related and Future Work for Part IV

In Part IV of the thesis we presented the notion of lock freedom for the π-calculus with sessions and studied the relationship between the notions of progress and lock freedom. We proved that they are strongly connected and progress can be thought of as a compositional form of lock freedom. In particular, for closed terms, i.e., terms with no free variables, lock freedom and progress coincide. For open terms, i.e., terms containing free variables, lock freedom and progress do not coincide. However, we defined a procedure to *close* a process P, by using catalysers and characteristic processes and obtain $\mathsf{close}(P)$. 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 [68, 69, 71], for analysing the progress property. We show in the following that, reusing Kobayashi's technique captures new interesting processes that have progress but could not be typed by previous type systems for progress studied for the π-calculus with sessions.

Comparison with Related Work Progress for session π-calculus has been studied by several works [10, 19, 20, 26, 27, 39, 93]. In [93] the author defines a session type system for the progress property by using Kobayashi's obligation and capability levels. In [10, 26, 39] progress is studied for multiparty session types. Padovani studies deadlock and lock freedom in the linear π-calculus, and by using our encoding of session types into linear π-calculus types, he transfers these properties from the linear π-calculus to the session π-calculus [94]. A very recent work [34] studies the formal relationship between the class of deadlock free session processes induced by the correspondence of session types with linear logic [16] and the class of deadlock free session processes induced by the encoding and Kobayashi's type system for deadlock freedom [70].

In the following we recall some examples taken from [10, 20, 27, 39, 93], show how the encoding works and compare them with our analysis. 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 14.4.2. Consider the session process

$$P \triangleq (\nu ab)(\nu cd)(a?(z).d!\langle z\rangle \mid c?(w).b!\langle w\rangle)$$

which is deadlocked, and therefore does not have progress. This process is not typable in the type systems for progress presented in [10,27,93]. By the encoding we obtain the following process:

$$[\![P]\!]_f = (\nu x)(\nu y)(x?(z).y!\langle z\rangle \mid y?(w).x!\langle w\rangle)$$

where in the encoding of session process P, there are the following associations $a, b \mapsto x$ and $c, d \mapsto y$. As expected, our technique discards $[\![P]\!]_f$ above since the process is untypable in Kobayashi's type system. In particular, this process results untypable since the *rel* predicate does not hold.

Example 14.4.3. Consider the session process

$$Q \triangleq (\nu ab)(\nu cd)\left(\begin{array}{l} a?(x).\ c!\langle x\rangle.\ c?(y).\ a!\langle y\rangle \\ \mid \\ b!\langle \texttt{true}\rangle.\ d?(z).\ d!\langle \texttt{false}\rangle.\ b?(z) \end{array}\right)$$

This process satisfies the progress property, but it is rejected by the type systems in [10, 20]. This is because in the two processes in parallel there is a circular dependency between channels. However, this circularity does not lead to deadlock. Let us now consider its encoding in the π-calculus, given by the following process:

$$[\![Q]\!]_f = (\nu k)(\nu l)\left(\begin{array}{l} k?(x, c_1).\ (\nu c_2)\Big(\ l!\langle x, c_2\rangle.\ c_2?(y).\ c_1!\langle y\rangle\ \Big) \\ \mid \\ (\nu c_1)\Big(k!\langle \texttt{true}, c_1\rangle.\ l?(z, c_2).\ c_2!\langle \texttt{false}\rangle.\ c_1?(z)\Big) \end{array}\right)$$

This process is correctly recognised as having progress by using 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, 59]. 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 yet whether Kobayashi's usage types are expressive enough to handle such situations, 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 A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[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*, volume 7273 of *LNCS*, pages 35–51. Springer, 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] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA*, volume 3527 of *LNCS*, pages 1–17. Springer, 2005.

[8] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. In *TGC*, volume 8902 of *LNCS*, pages 51–66. Springer, 2014.

[9] Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types (extended abstract). In *CONCUR*, volume 8704 of *LNCS*, pages 387–401. Springer, 2014.

[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*, volume 5201 of *LNCS*, pages 418–433. Springer, 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*, volume 1119 of *LNCS*, pages 655–670. Springer, 1996.

[13] Frank S. De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, volume 4421 of *LNCS*, pages 316–330. Springer, 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*, volume 7792 of *LNCS*, pages 330–349. Springer, 2013.

[16] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[17] Luís Caires and Hugo Torres Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.

[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. In *COORDINATION*, volume 8459 of *LNCS*, pages 49–64. Springer, 2014.

[20] Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *ICE*, volume 38 of *EPTCS*, pages 13–27, 2010.

[21] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[22] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Types for the ambient calculus. *Information and Computation*, 177(2):160 – 194, 2002.

[23] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[24] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. *SIGPLAN Not.*, 39(1):123–134, 2004.

[25] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64. ACM, 1998.

[26] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *COORDINATION*, volume 7890 of *LNCS*, pages 45–59. Springer, 2013.

[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 S. Blair, Paul Grace, François Taïani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1), 2008.

[30] Ornela Dardha. Recursive session types revisited. In *BEAT*, volume 162 of *EPTCS*, pages 27–34, 2014.

[31] Ornela Dardha, Elena Giachino, and Michael Lienhardt. A type system for components. In *SEFM*, volume 8137 of *LNCS*, pages 167–181. Springer, 2013.

[32] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, pages 139–150, New York, NY, USA, 2012. ACM.

[33] Ornela Dardha, Daniele Gorla, and Daniele Varacca. Semantic subtyping for objects and classes. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 66–82. Springer, 2013.

[34] Ornela Dardha and Jorge A. Pérez. Comparing deadlock-free session typed processes. In *EXPRESS/SOS*, volume 190 of *EPTCS*, pages 1–15, 2015.

[35] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *CONCUR*, volume 6901 of *LNCS*, pages 280–296. Springer, 2011.

[36] Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Inf. Comput.*, 204(7):1045–1082, 2006.

[37] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011.

[38] Mariangiola Dezani-Ciancaglini and Ugo de' Liguoro. Sessions and session types: an overview. In *WS-FM*, volume 6194 of *LNCS*. Springer, 2010.

[39] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008.

[40] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *FMCO*, volume 4709 of *LNCS*, pages 207–245. Springer, 2007.

[41] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[42] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alexander Ahern, and Sophia Drossopoulou. A distributed object-oriented language with session types. In *TGC*, volume 3705 of *LNCS*, pages 299–318. Springer, 2005.

[43] E. Allen Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[44] Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP*, volume 3086 of *LNCS*, pages 364–388. Springer, 2004.

[45] Simon J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

[46] Simon J. Gay, Nils Gesbert, and António Ravara. Session types as generic process types. In *EXPRESS/SOS*, volume 160 of *EPTCS*, pages 94–110, 2014.

[47] Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.

[48] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[49] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[50] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *IFM*, volume 7940 of *LNCS*, pages 394–411. Springer, 2013.

[51] 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.

[52] Joseph Gil and Itay Maman. Whiteoak: introducing structural typing into java. *SIGPLAN Not.*, 43(10):73–90, 2008.

[53] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[54] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.

[55] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[56] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The geneva convention – on the treatment of object aliasing. OOPS Messenger, 1992.

[57] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[58] Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[59] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, volume 43(1), pages 273–284. ACM, 2008.

[60] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theo. Comput. Sci.*, 311(1-3):121–163, 2004.

[61] 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.

[62] Focus Inria. Overall objectives. `http://raweb.inria.fr/rapportsactivite/RA2012/focus/uid3.html`.

[63] Einar Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs: A core language for abstract behavioral specification. In *FMCO*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

[64] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.

[65] 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.

[66] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.

[67] Naoki Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In *IFIP TCS*, volume 1872 of *LNCS*, pages 365–389. Springer, 2000.

[68] Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.

[69] Naoki Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, pages 439–453, 2002.

[70] Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[71] Naoki Kobayashi. Type systems for concurrent programs. Extended version of [69], Tohoku University, 2007.

[72] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371. ACM, 1996.

[73] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR*, volume 1877 of *LNCS*, pages 489–503. Springer, 2000.

[74] Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5), 2010.

[75] 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.

[76] 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.

[77] Michael Lienhardt, Mario Bravetti, and Davide Sangiorgi. An object group-based component model. In *ISoLA (1)*, volume 7609 of *LNCS*, pages 64–78. Springer, 2012.

[78] Michael 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 *FMCO*, volume 6957 of *LNCS*, pages 165–183. Springer, 2012.

[79] Michael Lienhardt, Alan Schmitt, and Jean-Bernard Stefani. Oz/k: a kernel language for component-based open programming. In *GPCE*, pages 43–52. ACM, 2007.

[80] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, New York, NY, USA, 1988. ACM.

[81] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP*, volume 5142 of *LNCS*, pages 260–284. Springer, 2008.

[82] Sun Microsystems. JSR 220: Enterprise javabeans, version 3.0 – ejb core contracts and requirements, 2006.

[83] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, may 1999.

[84] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[85] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[86] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[87] Fabrizio Montesi and Davide Sangiorgi. A model of evolvable components. In *TGC*, volume 6084 of *LNCS*, pages 153–171. Springer, 2010.

[88] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.

[89] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In *TLCA*, volume 4583 of *LNCS*, pages 321–335. Springer, 2007.

[90] Wired News. History's worst software bugs, 2005. `http://www.wired.com/software/coolapps/news/2005/11/69355`.

[91] OASIS. Web Services Business Process Execution Language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[92] Klaus Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008.

[93] Luca Padovani. From lock freedom to progress using session types. In *PLACES*, volume 137 of *EPTCS*, pages 3–19, 2013.

[94] Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *CSL-LICS*, pages 72:1–72:10. ACM, 2014.

[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 and observational equivalences for session-based concurrency. *Inf. Comput.*, 239:254–302, 2014.

[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*, pages 376–385. IEEE Computer Society, 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 *ECOOP*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.

[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*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[105] Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In *FoSSaCS*, volume 7213 of *LNCS*, pages 346–360. Springer, 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 *CONCUR*, volume 3170 of *LNCS*, pages 497–511. Springer, 2004.

[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*, volume 4960, pages 269–283. Springer, 2008.

[113] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286. ACM, 2012.

[114] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for java. In *OOPSLA*, pages 439–453. ACM, 2005.

[115] Yannick Welsch and Jan Schäfer. Location types for safe distributed object-oriented programming. In *TOOLS*, volume 6705 of *LNCS*, pages 194–210. Springer, 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.