Imperial College of Science, Technology and Medicine
Department of Computing

Session Types in Concurrent Calculi: Higher-Order Processes and Objects

Dimitris Mostrous

SUBMITTED IN PART FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTING OF IMPERIAL COLLEGE

November 2009

Abstract

This dissertation investigates different formalisms, in the form of programming language calculi, that are aimed at providing a theoretical foundation for structured concurrent programming based on session types. The structure of a session type is essentially a process-algebraic style description of the behaviour of a single program identifier serving as a communication medium (and usually referred to as a channel): the types incorporate typed inputs, outputs, and choices which can be composed to form larger protocol descriptions. The effectiveness of session typing can be attributed to the linear treatment of channels and session types, and to the use of tractable methods such as syntactic duality to decide if the types of two connected channels are compatible. Linearity is ensured when accumulating the uses of a channel into a composite type that describes also the order of those actions. Duality provides a tractable and intuitive method for deciding when two connected channels can interact and exchange values in a statically determined type-safe way. We present our contributions to the theory of sessions, distilled into two families of programming calculi, the first based on higher-order processes and the second based on objects. Our work unifies, improves and extends, in manifold ways, the session primitives and typing systems for the Lambda-calculus, the Pi-calculus, the Object-calculus, and their combinations in multi-paradigm languages. Of particular interest are: the treatment of infinite interactions expressed with recursive sessions; the capacity to encapsulate channels in higher-order structures which can be exchanged and kept suspended, i.e., the use of code as data; the integration of protocol structure directly into the description of objects, providing a powerful and uniformly extensible set of implementation abstractions; finally, the introduction of asynchronous subtyping, which enables controlled reordering of actions on either side of a session. Our work on higher-order processes and on object calculi for session-based concurrent programming provides a theoretical foundation for programming language design integrating functional, process, and object-oriented features.

Acknowledgements

First and foremost, I would like to thank my supervisor Nobuko Yoshida, who has been without doubt the person with the greatest influence in my life during the years of my doctorate studies. She advised me almost every day, tirelessly, dedicating more time and effort than anyone could ask from a supervisor. Nobuko, thank you for your support these past few years, and for your always moderate advice, on both professional and personal issues.

I would also like to thank my second supervisor, Sophia Drossopoulou (Imperial College London). I learned from Sophia both as a student and as a collaborator and co-author, and she was also always there for me in difficult times.

During my time at Imperial, I had the opportunity to work with and interact with some great people from other institutions. To begin, I would like to express my gratefulness that I worked with, and learned from, Mariangiola Dezani-Ciancaglini (Università di Torino). Also, I would like to thank Kohei Honda (Queen Mary, University of London) for our collaborations and discussions, and also for his frequent advice and encouragement. My warmest thanks go to Vasco Vasconcelos (Universidade de Lisboa), for his support and advice, and for his detailed feedback on my thesis. Finally, I am indebted to my examiners for their positive comments and for their proposed corrections which improved the presentation of my work, and to all the reviewers of the papers which I co-authored.

During my doctorate studies I also received useful advice, not on the technical material itself but on various other topics, from Susan Eisenbach (Director of Studies at the Department of Computing, Imperial College London).

I started to obtain my foundations in theoretical computer science during my Masters at Imperial, and I would like to extend my appreciation to all those that taught me during this time. Also I would like to thank the Department of Computing for the DTA award which supported my fees throughout my study, and for providing an excellent environment for students.

Last but not least, I would like to express my gratefulness and love to my family and friends for their support and friendship during these years. You know who you are!

The PhD is not just a piece of scientific work. It represents the training and ultimately the transformation of the candidate into a scientist. Thus, it is not just a contribution, but it also leads to a better way of thinking, and in that sense it seems appropriate to *dedicate* my work to those without which it would have been impossible:

I would like to dedicate my work jointly to my family and especially to my grandfathers who passed away before they could see me complete my degree, and to my supervisor who worked very hard for me to achieve it.

Contents

List of Figures

1 Introduction

1.1 Motivation and Objectives

Concurrency is becoming increasingly pervasive at all levels of computer programming, from low level and embedded software to operating systems and globally deployed web services. Nevertheless, concurrent programming suffers from the lack of mature and usable typing disciplines that have proven very successful in sequential programming. Moreover, the majority of concurrent software is written using shared memory models and threads, which are widely considered notoriously difficult to verify and understand. Adding to the above, modern trends in hardware, pronounced with the emergence of multicore microprocessors many of which utilise distributed memory models, create a need for new programming languages that can bridge the expanding chasm between hardware and software architecture. And it is not an exaggeration to say that these developments in hardware technology testify to the fact that the sequential model of computation has reached the limits of its scalability.

Presently, communication oriented software is mostly implemented using either sockets, facilitating the transmission of arbitrary messages, or with remote method invocation. Sockets provide untyped stream abstractions, and remote method invocation allows methods to be called in a distributed setting, using sockets as the underlying transport mechanism. Both have shortcomings: socket-based code requires a significant amount of dynamic checks and type-casts on the values exchanged, in order to ensure type safety; remote method invocation does ensure that methods are used as mandated by their type signatures, but does not capture behaviour arising from the combination of invocations that may implement a conceptual unit of interaction.

If we consider the type-based methodologies used in mainstream industrial-grade languages, we see increasingly sophisticated techniques for the verification of functions, objects, and other sequential constructs. Perhaps paradoxically, there is very little provision for the typing of communication primitives, such as sockets. This shows that most of these languages were not designed for concurrency and communication: rather, they impose typing demands on programs written using their sequential core, and no demands at all when it comes to primitives for the exchange of messages. Such untyped communications are the weakest link even in otherwise strongly-typed languages.

The essence of communication in programming can be abstractly captured in the notion of *message passing* between independently executing components within a larger software composition; if there is no communication then there is no need to verify the behaviour at the level of interaction between the components. When there is interaction through communication, it is intuitive to consider not only individual messages, but also the structure and compatibility of a protocol that implements a complete *structured dialogue* between components. The basic abstraction for message passing concurrency is that of a *communication channel*. Then the central question is, can we assign types to communication channels, in the way we give types to variables in a program? The requirements are different: a variable is a placeholder for a single value at a time, and its type does not change; a channel is a passageway for possibly heterogeneous values and control instructions, and if we are to give it a type, this type must facilitate change after each step of an interaction, so that the correctness of an implementation can be verified statically. When there is communication, there are normally at least two participants, and therefore we also need to capture the intended symmetries in communications, in the general sense that when sending a message it will be received at the other end, and vice versa.

Session typing addresses exactly these requirements, fortifying the communications-oriented primitives of a language with a type-based verification discipline that can statically ensure that interactions are indeed *well-behaved*. Session types enable the validation of programs with structured communications, assuring both *type* and *communication-safety* — not only is the value of each message correctly typed — but also the sequence of messages that are sent and received via a channel is performed according to the exact scenario specified by the session type, precluding communication mismatch. Moreover, session typed code is more concise than using sockets directly, eliminating the explicit checks which otherwise proliferate in interactions where different types of values need to be communicated following an evolving protocol. Finally, sessions are useful as an abstraction for concrete communication mechanisms: the use of sockets directly is not always desirable, as more efficient shared memory data structures can be used for sessions taking place within the same memory domain, and this can be done safely and behind the scenes.

This dissertation presents fundamental theories for session typing in programs that utilise the powerful abstractions of processes, code mobility, and object orientation.

1.2 Contributions

We formalise for the first time session typing for a process language that allows not only data but also runnable code to be the subject of structured type-safe communications. The ability to exchange code is fundamental in concurrent and distributed systems where programs cannot be fully fixed ab initio and dynamicity is a prerequisite. We then relax the strict compatibility requirements that govern pairs of interacting processes to allow certain classes of message-passing actions to be permuted in restricted ways, offering not only greater flexibility in composing programs, but also guidance toward type-safe optimisations. Finally, we introduce an original session typing system to a minimal object calculus that can serve as a theoretical foundation for the several disciplines of object-orientation, without restricting this powerful style of programming to class-based languages.

1.3 Publications & Detailed Contribution of the Author

The following publications, presented in reverse chronological order, are in varying degrees the source of the expanded and improved material presented in this dissertation. We indicate the main publications relating to each chapter; the remaining have nevertheless contributed to the understanding and intuition of this work, even if the material does not directly appear in the main body of the thesis.

Declaration Only the material relating directly to the author's research appears in this thesis. Any work, in the form of original ideas, formal systems, writing, and proofs, that have been contributed by my supervisor and other researchers and that appears in the following joint publications, has been omitted from the main body.

1. **Dimitris Mostrous** and Nobuko Yoshida. Session-Based Communication Optimisation for Higher-Order Mobile Processes. In Pierre-Louis Curien (Ed.), *9th International Conference on Typed Lambda Calculi and Applications* (TLCA'09), volume 5608 of Lecture Notes in Computer Science, pages 203–218, Springer, 2009 [64].

   Author's Contribution: The ideas, formulations, writing, and proofs of this work are my own.

   Corresponding Part: Chapter 5.

2. Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, **Dimitris Mostrous** and Nobuko Yoshida. Objects and session types. In *Information and Computation*, number 5, volume 207, pages 595–641, Elsevier, 2009 [30].

   Author's Contribution: This work is an extended and improved version of our previous work on objects (ECOOP 2006 [31], mentioned below). My contribution in this version was in the overall improvement, presentation and proofs, and in particular the sections of introduction, related work, and conclusion, are my own writing. I also contributed in the proof-reading of the remaining material.

   Corresponding Part: This work does not appear in the dissertation because it originates from research conducted during my Masters degree; it was nevertheless published during my Doctorate studies.

3. **Dimitris Mostrous,** Nobuko Yoshida and Kohei Honda. Global Principal Typing in Partially Commutative Asynchronous Sessions. In Giuseppe Castagna (Ed.), *18th European Symposium on Programming* (ESOP'09), volume 5502 of Lecture Notes in Computer Science, pages 316–332, Springer, 2009 [65].

   Author's Contribution: In this work I have contributed the main technical results in Section 3 of the paper, and in particular the theoretical framework for subtyping, the formalisation and proof of a coinductive subtyping system for multiparty sessions, and some contributions to the algorithmic subtyping system. The major part of the algorithmic typing system, and the second major contribution of the paper, which consists of a global approach to typing, were not developed by myself.

   Corresponding Part: This work does not appear directly in the dissertation because my part was highly technical and it was further developed in the TLCA 2009 paper mentioned above. My contribution to this work has motivated my further work presented in Chapter 5, but in this way only my own research is presented in the dissertation.

4. **Dimitris Mostrous** and Nobuko Yoshida. Two Session Typing Systems for Higher-order Mobile Processes. In S. Ronchi Della Rocca (Ed.), *8th International Conference on Typed Lambda Calculi and Applications* (TLCA'07), volume 4583 of Lecture Notes in Computer Science, pages 321–335, Springer, 2007 [63].

   Author's Contribution: This work presents two systems for session typing higher-order processes. My contribution is the ideas, design, and proofs of the primary system, which is the one presented in this thesis. The second system is referred to in related work.

Corresponding Part: Chapter 4.

5. Mariangiola Dezani-Ciancaglini, **Dimitris Mostrous,** Nobuko Yoshida and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas (Ed.), *The 20th European Conference on Object Oriented Programming* (ECOOP'06), volume 4067 of Lecture Notes in Computer Science, pages 328–352, Springer, 2006 [31].

   Author's Contribution: In this work my contribution is in the theoretical framework, the formalisation, and the proofs of the main system. It was conducted during my Masters degree at Imperial (final project).

   Corresponding Part: This work does not appear in my thesis, because the research took place during my Masters degree. However it has influenced my intuition and interest in subsequent work on object languages, which is independent and appears in Chapter 6.

Electronic versions are available at `http://www.doc.ic.ac.uk/~mostrous`.

1.4 Synopsis

The dissertation is divided into three parts, covering the foundations and related work, the main theories, and the future directions.

Part I: Background

Chapter 2 motivates the choice of languages and demonstrates, in an untyped setting, the power of processes with code mobility and the integration with object-oriented structuring. Session types are introduced with reference to desirable and undesirable compositions of processes.

Chapter 3 provides an extended review of the session typing literature and of closely related approaches.

Part II: Session Types and Subtyping in Higher-Order Processes and Objects

Chapter 4 introduces sessions into a synchronous higher-order process language with code mobility manifested through the exchange of functions.

Chapter 5 extends the language and typing discipline of the previous chapter to allow certain communications to be partially permuted, a feature enabled by non-blocking semantics for buffered communication.

Chapter 6 integrates sessions into a foundational object language with imperative semantics and buffered communication.

Part III: Conclusion & Future Directions

Chapter 7 focuses on important future work which can pave the way to practical implementations based on the fundamental theories developed in the main part.

Part I

Background

2 Foundations

Overview *In this chapter we lay the foundations of the work in this thesis, placing it in the context of the different theories that are used in the formal systems defined later. It follows a slightly informal style aiming at exposing the subtleties in modelling and expressing systems in process and object based programming calculi. We motivate and justify the main concepts of session typing as a discipline that allows desirable programs, rejecting behaviours we identify as unsafe.*

2.1 Higher Order Processes

A significant part of the formal study of concurrency develops around the notion of *processes*. By process we mean a unit of execution that can interact with other processes, by way of *communication*, that is, by sending and receiving values over *channels*. Several process calculi have been formalised — early examples are Milner's CCS [58] and Hoare's CSP [46], followed by the π-calculus [61, 62] by Milner *et al.*, which extends CCS with the ability to form dynamic link topologies — and we are interested in an extension of the π-calculus, Sangiorgi's Higher-order π-calculus [79], or HOπ, which integrates π-calculus with λ-calculus. The standard reference for λ-calculus is Barendregt's book [6]. Sangiorgi and Walker's book [80] is a complete reference of the fundamentals of π-calculus and Higher-order π-calculus.

By combining the foundational theories of processes (π-calculus) and functions (λ-calculus), HOπ facilitates not only communication and link mobility, but also *code mobility*, which is represented as the communication of a function. As Sangiorgi [78] has shown, at the semantic level the π-calculus is already higher-order, in the sense that HOπ can be encoded into first-order π, but as a basis for *programming language design* we are interested in adopting a formalism that can directly represent the essential elements of concurrent and distributed computation, where typically we encounter a combination of functions and communications. For instance, mobile code is necessary when modelling or implementing remote code installation which is very common in mobile devices and in personal computers, even at the operating system level.

First, we briefly introduce the main constructs of an untyped Higher-order π-calculus.

HOπ in a Nutshell

The syntax for the untyped HOπ variant that we use is shown in Figure 2.1. The reduction rules are in Figure 2.2. Structural equivalence axioms are in Figure 2.3. We denote processes with P, Q, R. Variables range over x, y, z, but we also use f in examples. Values V can be functions, communication channels ranging over a, b, or the constant unit, written ().

Identifiers

u,v	$::=$	x,y,z	variables
	$\mid$	a,b,c	channels

Values

V,V'	$::=$	u,v	identifier
	$\mid$	$()$	unit
	$\mid$	$\lambda x \,.\, P$	function
	$\mid$	$\mu x \,.\, \lambda y \,.\, P$	recursion

Terms

P,Q,R	$::=$	V	value
	$\mid$	$u?(x).P$	input
	$\mid$	$u!\langle V\rangle.P$	output
	$\mid$	$u \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$	branching
	$\mid$	$u \lhd l_m \,.\, P$	selection
	$\mid$	$P \mid Q$	parallel
	$\mid$	$(\nu a)\ P$	restriction
	$\mid$	$P \cdot Q$	application
	$\mid$	$\mathbf{0}$	nil

Figure 2.1: HOπ Syntax

Functions The term $\lambda x \,.\, P$ is a function with argument x and body P. The notation $(\lambda x \,.\, P) \cdot V$ denotes function application, where the value V takes the place of the variable x in the function body P, written $P\{V/x\}$, which can then use it to produce a result. We write $\mu z \,.\, \lambda x \,.\, P$ for a recursive function which is defined like $\lambda x \,.\, P$ with the difference that $(\mu z \,.\, \lambda x \,.\, P) \cdot V$ results in P

(beta)
$$(\lambda x\,.\,P)\cdot V \quad\longrightarrow\quad P\{V/x\}$$
(rec)
$$(\mu y\,.\,\lambda x\,.\,P)\cdot V \quad\longrightarrow\quad P\{V/x\}\{\mu y.\lambda x.P/y\}$$
(comm)
$$a?(x)\,.\,P \mid a!\langle V\rangle\,.\,Q \quad\longrightarrow\quad P\{V/x\} \mid Q$$
(label)
$$a \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \mid a \triangleleft l_m.P \quad\longrightarrow\quad P_m \mid P \qquad 1 \leq m \leq n$$

$$\text{(app-l)}\ \frac{P \longrightarrow P'}{P\cdot Q \longrightarrow P'\cdot Q} \qquad \text{(app-r)}\ \frac{Q \longrightarrow Q'}{V\cdot Q \longrightarrow V\cdot Q'} \qquad \text{(par)}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$\text{(resc)}\ \frac{P \longrightarrow P'}{(\nu a)\ P \longrightarrow (\nu a)\ P'} \qquad \text{(str)}\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Figure 2.2: HOπ Reduction

$P =_\alpha Q \Rightarrow P \equiv Q$	Renaming of bound variables
$P \mid Q \equiv Q \mid P$	Commutativity of parallel composition
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	Associativity of parallel composition
$P \mid \mathbf{0} \equiv P$	Inaction and parallel composition
$(\nu a)\ P \mid Q \equiv (\nu a)\ (P \mid Q) \quad a \notin \mathsf{fn}(Q)$	Scope extrusion
$(\nu a)\ (\nu b)\ P \equiv (\nu b)\ (\nu a)\ P$	Exchange
$(\nu a)\ \mathbf{0} \equiv \mathbf{0}$	Inaction and restriction

Figure 2.3: HOπ Structure Congruence

with V for x *and* $\mu z\,.\,\lambda x\,.\,P$ for z. In other words, when this function is applied, it has access to its own definition through the recursion variable z, and can therefore reuse itself within its body P. We also make use of the following abbreviated forms for function application:

$$\begin{array}{rcll} P\,;\,Q & \overset{\text{def}}{=} & \mathsf{let}\ z{=}P\ \mathsf{in}\ Q & \text{if } z \text{ does not occur in } Q \\ \text{with}\quad \mathsf{let}\ z{=}P\ \mathsf{in}\ Q & \overset{\text{def}}{=} & (\lambda z\,.\,Q)\cdot P & \end{array}$$

Note that in the above, if P does not reduce to a value, the computation becomes *stuck*.

Processes The process $\mathbf{0}$ represents inaction. The prefix form $a!\langle V\rangle\,.\,P$ consists of the sending of a value V on channel a, followed by whatever P specifies. Similarly to a function, $a?(x)\,.\,P$ can receive a value (say, V) via a and then do P but with V taking the place of x. With $P \mid Q$ the processes P and Q are executed concurrently, hence in $a!\langle V\rangle\,.\,P_1 \mid a?(x)\,.\,P_2$ a communication on a can be performed, followed by the remaining actions $P_1 \mid P_2'$ where P_2' is P_2 with V for x. The notation $(\nu a)\ P$ means that the communication channel a is private to P, thus, a can be thought of as being *freshly created* in $(\nu a)\ P$. Consequently, in $(\nu a)\ (a!\langle V\rangle\,.\,P) \mid a?(x)\,.\,Q$, the instances of a in $a!\langle V\rangle\,.\,P$ and $a?(x)\,.\,Q$ are considered distinct, and no communication can take place on a. To avoid obscuring the presentation, we just show the main use of fresh names through the following idiomatic example:

$$(\nu x)\ (\ a!\langle x\rangle\,.\,x!\langle V\rangle\,.\,\mathbf{0}\ ) \quad \mid \quad a?(z)\,.\,z?(y)\,.\,x!\langle V'\rangle\,.\,\mathbf{0}$$

Since instances of x in the left and right processes are meant to be different, we rename the variable x of the left term to x', obtaining:

$$(\nu x')\ (\ a!\langle x'\rangle\,.\,x'!\langle V\rangle\,.\,\mathbf{0}\ ) \quad \mid \quad a?(z)\,.\,z?(y)\,.\,x!\langle V'\rangle\,.\,\mathbf{0}$$

where we chose some x' which does not occur in the original term. Then, we can *extend* the scope of $(\nu x')$, specified here with parentheses, without causing conflicts, obtaining:

$$(\nu x')\ (\ a!\langle x'\rangle\,.\,x'!\langle V\rangle\,.\,\mathbf{0} \quad \mid \quad a?(z)\,.\,z?(y)\,.\,x!\langle V'\rangle\,.\,\mathbf{0}\ )$$

Now the communication on a can take place within the new scope of $(\nu x')$, and the fresh channel x' will become shared between the two processes, substituting z on the right. After that step, no other process can interact with the channel x', therefore we can guarantee the non-interference of the subsequent communication on it, in the sense that only the process on the right of | can receive the value V.

We write $u \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ for a process offering n alternative behaviours (branches) indexed by the labels $l_1 \ldots l_n$. Then, $u \triangleleft l_m . Q$ with $1 \leq m \leq n$ represents a selection of the behaviour indexed by l_m, followed by Q. For example, $u \triangleleft l_2.Q \mid u \triangleright \{l_1 : P_1, l_2 : P_2\}$ becomes $Q \mid P_2$.

The combination of recursion and branching can encode complex repetitive protocols, as we demonstrate in the next section.

Mobile Processes in Industry

We give evidence to the expressiveness and relevance of HOπ with a set of examples based on a scenario from industry. The general idea is not fictional: this is how a large corporation processes photographs in real-time from live events in disparate locations, forwarding the edited versions to clients such as news websites and publishers. An important aspect of this case study is that mobile code is not just an optimisation, but rather the *enabling* factor, the sine qua non for an extensible implementation.

Real-time Photo Feeds [75]

The physical actors in the scenario are: Photographers, which carry mobile devices that store the pictures once they are taken; Editors, whose task is to prepare the pictures for publication, by applying transformations and adding metadata; Image server, where edited pictures are stored in order to be distributed; Clients, which receive feeds of edited pictures from the server.

For the purposes of this example, we focus on the interaction, at the software level, between the Editor and Photographer's systems. We expose this part of the scenario in more detail:

- Image files are large, and the network links between photographers and the editor are slow.
- To achieve good performance, when an Editor has to work on the pictures from an event, it receives a list of thumbnails instead of the originals.
- The Editor applies transformations on chosen thumbnails, and the same instructions are then sent to the Photographer's system, so that the same editing can be applied to the original picture.
- The edited pictures are sent to the Image server directly from the Photographer. This is an optimal strategy because many times the transformations reduce the size of an image

dramatically, for example by scaling or cropping, which means that in some cases the whole process can be completed without the transfer of any large files. Therefore, this is preferred to the alternative of first moving the originals of chosen thumbnails at the Editor's location, then editing and sending from there. Moreover, this strategy avoids the overhead of the Editor acting as an intermediate transit point between the pictures and the server.

- The transformations for each picture are sent to the Photographer's system one by one as they are applied at the Editor, to facilitate maximum concurrency especially for time-consuming actions.

A Higher-order π-calculus Implementation of the Scenario

We now show how the above case study can be precisely expressed in HOπ. First, we assume some system provided functions (such as print and DB.load) and constants (such as img42), written in different typeface. To keep the use of those functions concise, we use the following abbreviated form:

$$\mathsf{F}(x_1,\ldots,x_n) \quad = \quad (((\mathsf{F}\cdot x_1)\cdot x_2\ldots)\cdot x_n)$$

We define the process at the Photographer, $\mathsf{PhotoSrv}(a)$, as follows:

$$\mathsf{PhotoSrv}(a) \quad \stackrel{\text{def}}{=} \quad \begin{array}{l} a?(x)\,.\,x?(image\_id)\,. \\ \mathsf{let}\ img = \mathsf{DB.load}(image\_id)\ \mathsf{in} \\ \Big(\mu y\,.\,\lambda z\,.\,x \triangleright \{\,\mathsf{nextFilter} : x?(f)\,.\,(\,\mathsf{let}\ z' = f\cdot z\ \mathsf{in}\ y\cdot z'\,)\,, \\ \qquad\qquad \mathsf{done} : \mathsf{DB.save}(image\_id, z)\,;\,\mathbf{0}\,\}\Big)\cdot img \end{array}$$

The definition $\mathsf{PhotoSrv}(a)$ is parameterised with the channel a, which can be thought of as the location of the process. Note that, for simplicity, we do not show how the thumbnail images are obtained, and assume that when interacting with this code they are known; it is easy to model their transmission as a series of outputs. First, a new channel is received over a, replacing x, and then it is used to receive the key *image_id* identifying the picture to be edited. Next, the picture is loaded, as *img*, from the local database DB using DB.load. Now a repetitive behaviour is defined: the prefix $\mu y\,.\,\lambda z.\ldots$ specifies a recursive function which can access itself using y, taking an argument for z. This function is applied to the image *img* which replaces z in its body. The body of this function is a process that offers the choices nextFilter and done on x. If the first choice is made, then an image transformation function is received over x, replacing occurrences of f, and then it is applied to the image with $f \cdot z$. The last step is a recursive function application $y \cdot z'$, which repeats the process by applying y to the new image z', so that further transformations can be applied. If the choice done is made, then the image is saved locally with $\mathsf{DB.save}(\mathit{image\_id}, z)$ and the code terminates, since there is no recursion here.

Note that for simplicity, we define $\mathsf{PhotoSrv}(a)$ so that it can be used only once and then it vanishes. However, we can easily redefine it, encapsulating it with an additional recursion and making it available in a persistent way.

Next we define a process $\mathsf{Editor}_1(a)$ which can interact with $\mathsf{PhotoSrv}(a)$:

$$\mathsf{Editor}_1(a) \overset{\text{def}}{=} \begin{array}{l} (\nu x)\ (\nu ack) \\ \Big(\mathsf{let}\ f = \lambda\, img\,.\, ack!\langle()\rangle\,.\, P\ \mathsf{in} \\ \quad a!\langle x\rangle\,.\, x!\langle \mathsf{img42}\rangle\,. \\ \qquad x \triangleleft \mathsf{nextFilter}\,.\, x!\langle f\rangle\,.\, x \triangleleft \mathsf{done}\,. \\ \qquad\quad ack?(z)\,.\, \mathsf{print}(\text{“}ok\text{”})\,;\, \mathbf{0}\Big) \end{array}$$

This definition begins with a declaration of two fresh channels, unique in every instance of $\mathsf{Editor}_1(a)$. The first, declared in (νx) , will be sent over a, facilitating further communications. If we avoided this apparent indirection and used a directly for all the communications, then another process in parallel, possessing a, could interfere at any stage (since a is not restricted to some term), destroying the determinacy of the intended protocol which is between the two processes only. The initial step in $\mathsf{Editor}_1(a)$ is the creation of a transformation function $\lambda\, img\,.\, ack!\langle()\rangle\,.\, P$ bound to f. The function takes an image as its *img* argument. When this function is applied, it first sends an acknowledgement over *ack*, acting as a confirmation that it is being executed. We then assume that specific image manipulations are performed in P using *img*. Next, the fresh channel x is sent over a, facilitating a unique connection with the receiving process, then $\mathsf{img42}$ is chosen for editing with the output $x!\langle \mathsf{img42}\rangle$, then the choice $x \triangleleft \mathsf{nextFilter}$ is made followed by the output of the function f in $x!\langle f\rangle$, followed by the choice $x \triangleleft \mathsf{done}$ terminating the interaction on x. The process then blocks waiting to receive the acknowledgement with $ack?(z)$, before printing a confirmation message.

The above processes can be composed in parallel to obtain:

$$\mathsf{PhotoSrv}(a) \mid \mathsf{Editor}_1(a)$$

implementing the intended interactions. Note that the first step is to rename all the variables to make them distinct, most importantly by extending the scope of (νx) and (νack) to ensure that x and ack do not appear in $\mathsf{PhotoSrv}(a)$.

An Extended Use Case Suppose that we modify $\mathsf{PhotoSrv}(a)$, adding the two more choices upload and undo, to obtain $\mathsf{ExtPhotoSrv}(a)$ defined below:

$$\mathsf{ExtPhotoSrv}(a) \quad \stackrel{\text{def}}{=} \quad \begin{array}{l} a?(x)\,.\,x?(image\_id)\,. \\ \quad \mathsf{let}\; img = \mathsf{DB.load}(image\_id)\;\mathsf{in} \\ \quad \Big(\mu y\,.\,\lambda z\,.\,x \triangleright \{\,\mathsf{nextFilter} : x?(f)\,.\,(\,\mathsf{let}\; z' = f \cdot z\;\mathsf{in}\; y \cdot z'\,)\,, \\ \qquad\qquad\qquad \mathsf{done} : \mathsf{DB.save}(image\_id, z)\,;\,\mathbf{0}\,, \\ \qquad\qquad\qquad \mathsf{upload} : x?(dest)\,.\,dest!\langle z\rangle\,.\,x!\langle()\rangle\,.\,(\,y \cdot z\,)\,, \\ \qquad\qquad\qquad \mathsf{undo} : \mathsf{let}\; z' = \mathsf{DB.load}(image\_id)\;\mathsf{in}\; y \cdot z'\,\}\Big) \cdot img \end{array}$$

The new functionality allows $\mathsf{ExtPhotoSrv}(a)$ to be instructed to send the current image to a given destination, and also to undo the transformations by reloading the original image from the database and invoking the recursion with it. Moreover, we can now define:

$$\mathsf{Editor}_2(a,b) \quad \stackrel{\text{def}}{=} \quad \begin{array}{l} (\nu x) \\ \Big(a!\langle x\rangle\,.\,x!\langle \mathsf{img42}\rangle\,.\,\ldots \\ \quad x \triangleleft \mathsf{upload}\,.\,x!\langle b\rangle\,.\,x?(y)\,.\,x \triangleleft \mathsf{done}\,. \\ \qquad \mathsf{print}(\textit{“upload complete”})\,;\,\mathbf{0}\Big) \end{array}$$

which uses the new option upload after applying some transformations (which we omit for clarity), and we can define the composition:

$$\mathsf{ExtPhotoSrv}(a) \mid \mathsf{Editor}_2(a,b) \mid b?(x)\,.\,Q$$

as well as:

$$\mathsf{ExtPhotoSrv}(a) \mid \mathsf{Editor}_1(a)$$

These compositions indicate, as expected, that the extended process $\mathsf{ExtPhotoSrv}(a)$ can interact with a superset of processes compared to the original $\mathsf{PhotoSrv}(a)$.

Remarks

The above industry-inspired example was easily modelled in the untyped HOπ, giving testament to the expressive power of the calculus, and also to the invaluable contribution of mobile code, without which the scenario would be impossible to implement both efficiently and extensibly. In fact, we can postulate that the same strategy applies to many cases where there is a large data set and we wish to process it from a remote location. We can further distill a useful principle in deciding when utilisation of mobile code is appropriate or even necessary: *When a computation requires a piece of code and a piece of data which are not co-located, and if optimal performance is the objective, then the smallest of the two should move, assuming there is sufficient computational power at the target site, relative to the difference in transfer time induced by the sizes of the code and the data.*

2.2 Objects

The Higher-order π-calculus has the primitives needed to express complex communication patterns, but it does not have much provision for *modular structuring* and *dynamic extensibility* of programs. Moreover, it is evident that the combination of recursion and choice enables the expression of complex interactive behaviour, and it is natural to ask whether a more high level construction combining all of these capabilities can simplify programming. Also, another matter of interest is the interplay between communication and imperative constructs.

Objects emerge as a natural candidate, because they combine functions, recursion, and branching in a compact and powerful abstraction. The untyped imperative object calculus **impς** of Abadi and Cardelli [2] is a small formalisation that distills the essential features of object-orientation, namely objects, methods, and state. A concurrent variant of **impς**, named **concς**, with parallel composition and mutex synchronisation was developed by Gordon and Hankin [43].

For the purposes of this section, we “borrow” the main object-oriented elements from the functional variant of **impς** — referring to it as ς-calculus hereafter — and integrate them into HO$\pi\varsigma$, an adaptation of the simple HOπ of the previous section, reusing definitions as necessary. We do not attempt a complete formalisation here, but rather provide a model that will make concrete the concepts pertaining to sessions in object languages. Note that the calculus HO$\pi\varsigma$ is original, it does not appear in the literature, and for this reason we are more precise in our description comparing to the HOπ of the previous section.

HOπς: An Integration of HOπ and the functional ς-calculus

We *remove* recursive functions $\mu y\,.\,\lambda z\,.\,P$ and the branching construct $u \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ from the HOπ of the previous section. We then add the following productions. First, for objects we add w, defined as:

$$\begin{array}{rcll} w, w\prime & ::= & x & \text{variables} \\ & | & [\, l_i {=} \varsigma(x_i)\, P_i{}^{i\in I}\,] & \text{objects} \end{array}$$

An object consists of a collection of methods, and each method begins with a *self* binder $\varsigma(x_i)$, as in [2]. The self variables x_i provide access to the object within its own methods, similarly to the recursion variable y in μy. There are differences in the typing of self and functional recursion, but in this untyped context this issue does not concern us; see [2].

We extend values to include objects:

$$\begin{array}{rcll} V & ::= & \ldots & \textit{as before except recursion} \\ & | & [\, l_i {=} \varsigma(x_i)\, P_i{}^{i\in I}\,] & \text{object} \end{array}$$

Then we extend processes with the following fundamental object primitives:

$$\begin{array}{rcll} P, Q, R & ::= & \ldots & \textit{as before except branching} \\ & | & w.l_m & \text{method select} \\ & | & w.l_m \Leftarrow \varsigma(x)\, P & \text{method update} \\ & | & x \triangleright w & \text{branching} \end{array}$$

We write $w.l_m$ for the *selection* of method l_m of object w. We define the result of performing this as follows (where $\longrightarrow$ denotes reduction):

$$[\, l_i {=} \varsigma(x_i)\, P_i{}^{i\in I}\,].l_m \quad \longrightarrow \quad P_m\{{}^{[\, l_i = \varsigma(x_i)\, P_i{}^{i\in I}\,]}/_{x_m}\} \qquad \text{if } m \in I$$

With $P_m\{{}^{[\, l_i = \varsigma(x_i)\, P_i{}^{i\in I}\,]}/_{x_m}\}$ we denote the capture avoiding substitution of the self variable x_m with the actual object being invoked. Moreover, in this and the following reduction rules, if $m \notin I$, the term is simply *stuck*. Next, we write $w.l_m \Leftarrow \varsigma(x)\, P$ for method *update*, where a new method definition replaces the existing one in the returned copy (in this functional setting) of object w. This reduction is defined as:

$$[\, l_i {=} \varsigma(x_i)\, P_i{}^{i\in I}\,].l_m \Leftarrow \varsigma(x)\, P \quad \longrightarrow \quad [\, l_i {=} \varsigma(x_i)\, P_i{}^{i\in I\setminus m}, l_m {=} \varsigma(x)\, P\,] \qquad \text{if } m \in I$$

Note that in the original ς-calculus there are no functions, since they can be encoded using a special "arg" method which is updated with the argument. However, we kept them, because they allow us to encode sequential protocols which in the ς-calculus can only be encoded with a more complicated version of method update that computes a value before the update, since in this simpler version the new method is not executed; it just modifies an object's definition. See [2] for more details.

Finally, we re-introduce branching to the language with $u \triangleright w$, reducing with the rule (which uses selection from HOπ):

$$\begin{array}{ccl} u \triangleleft l_m \,.\, Q & \mid & u \triangleright [\, l_i {=} \varsigma(x_i)\, P_i{}^{i\in I} \,] \\ & \longrightarrow & \\ Q & \mid & [\, l_i {=} \varsigma(x_i)\, P_i{}^{i\in I} \,].l_m \qquad\qquad \text{if } m \in I \end{array}$$

This reduction simply uses selection to choose the corresponding method that will implement the branch. Note that as Q can continue to use u, so does the object, if u appears within the method.

The notions of free and bound identifiers are standard, extending to the new constructions, and we reuse the structural congruence $\equiv$ from the previous section.

A HOπς Implementation of the Scenario

We can write $\mathsf{ObPhotoSrv}(a)$, an object-based version of $\mathsf{PhotoSrv}(a)$, using a combination of process, functional, and object-oriented features:

$$\mathsf{ObPhotoSrv}(a) \stackrel{\text{def}}{=} \begin{array}{l} a?(x)\,.\,x?(\mathit{image\_id})\,. \\ \quad \mathsf{let}\ \mathit{img}_0 = \mathsf{DB.load}(\mathit{image\_id})\ \mathsf{in} \\ \quad x \triangleright \left[\begin{array}{l} \mathsf{nextFilter} = \varsigma(y) \\ \qquad x?(f)\,.\,(\,\mathsf{let}\ \mathit{img}_1 = f \cdot (\,y.\mathsf{img}\,)\ \mathsf{in} \\ \qquad\qquad \mathsf{let}\ y_1 = (\,y.\mathsf{img} \Leftarrow \varsigma(x_1)\, \mathit{img}_1\,)\ \mathsf{in}\ x \triangleright y_1\,)\,, \\ \mathsf{done} = \varsigma(y)\ \mathsf{let}\ z{=}y.\mathsf{img}\ \mathsf{in}\ (\,\mathsf{DB.save}(\mathit{image\_id}, z)\,;\mathbf{0}\,)\,, \\ \mathsf{img} = \varsigma(y)\, \mathit{img}_0 \end{array} \right] \end{array}$$

In this version, the image is stored into the method img of the object. Initially it has the value img_0 from the outside scope, then in every invocation of $\mathsf{nextFilter}$ an update is performed on the self variable y producing a copy y_1 of the object with the new value for img. Then branching is performed on the updated object with $x \triangleright y_1$ and the process repeats.

In general, method update can be used to dynamically obtain modified versions of objects: suppose we call the object used above w, then if we have such an object in a program, we can update the implementation, for example, with:

$$\mathsf{let}\ w\prime = (\,\mathsf{let}\ orig = w.\mathsf{img}\ \mathsf{in}\ w.\mathsf{img} \Leftarrow \varsigma(y)\ \ \mathsf{lock}!\langle image\_id\rangle\,.\,(\,orig\,)\,)\ \mathsf{in}\ P$$

This way another component can be notified when the image is first accessed, perhaps by locking the local database, and with a similar modification of done, the object can be unlocked. Observe that after the first access in $\mathsf{ObPhotoSrv}(a)$ the method img is internally updated, hence our modified code is only reduced once. Also note that in the above program fragment we assumed that *image_id* is known; it could of course be stored in the object. As before we can write:

$$\mathsf{ObPhotoSrv}(a) \mid \mathsf{Editor}_1(a)$$

and obtain the expected behaviour.

Remarks

Objects can be naturally integrated with process-oriented features providing a way to structure and dynamically manipulate interacting components which is necessary in practical programming. An untyped formalism such as HO$\pi\varsigma$, although inherently unsafe, can give insight into the design requirements of safer variants for concurrent and distributed programming.

2.3 From Synchronous to Buffered Communication

In the context of programming, it is natural to think of communication as asynchronous, in the sense that sender and receiver need not synchronise on every action between them. Indeed, for many protocols such synchronisation imposes an undesirable and unnecessary overhead. At the level of concurrent calculi, it is easy to formalise this way of operation, by adding a basic level of indirection to communications, in the form of *buffers*. The idea is that every output action synchronises not with an input, but instead with a buffer, and similarly for inputs. When a value is sent, it is *appended* at the end of the associated buffer; when receiving, the first element in the buffer is returned. This formulation is a basic model for popular asynchronous transports, such as TCP.

We implement the idea by adding buffered channels to HOπ, and by extension, to HOπς. First, we extend the values to include labels, needed for branching:

$$\begin{array}{rcll} V & ::= & \ldots & \textit{as before} \\ & \mid & l & \text{label} \end{array}$$

Then, we extend processes with a notion of buffer:

$$\begin{array}{rcll} P,Q,R & ::= & \ldots & \textit{as before} \\ & \mid & a\!:\!\vec{V} & \text{buffer} \end{array}$$

Each buffer holds a vector of values $\vec{V}$. The empty vector is denoted ε. The reduction rules

(send)

$$u!\langle V\rangle.P \mid u\!:\!\vec{V} \longrightarrow P \mid u\!:\!\vec{V}V$$

(recv)

$$u?(x).P \mid u\!:\!V\vec{V} \longrightarrow P\{V/x\} \mid u\!:\!\vec{V}$$

(sel)

$$u \lhd l.P \mid u\!:\!\vec{V} \longrightarrow P \mid u\!:\!\vec{V}l$$

(HOπ-bra)

$$u \rhd \{l_1\!:\!P_1,\ldots,l_n\!:\!P_n\} \mid u\!:\!l_m\vec{V} \longrightarrow P_m \mid u\!:\!\vec{V} \qquad 1 \le m \le n$$

(HOπς-bra)

$$u \rhd [\, l_i{=}\varsigma(x_i)\, P_i{}^{i\in I}\,] \mid u\!:\!l_m\vec{V} \longrightarrow [\, l_i{=}\varsigma(x_i)\, P_i{}^{i\in I}\,].l_m \mid u\!:\!\vec{V} \qquad 1 \le m \le n$$

Figure 2.4: Asynchronous Reduction in HOπ and HOπς

(comm) and (label) of Figure 2.2 are removed, and the rules of Figure 2.4 are added. For branching there are two alternatives, one for each calculus. The new reductions simply model the movement of values, including labels, to and from the buffers in an order-preserving way. In this untyped setting, we do not impose that buffers exist for every name in a process; for example the term $a!\langle V\rangle\,.\,\mathbf{0}$ alone is simply *stuck*, in the same way that this term would be stuck if it appeared without a matching input in the synchronous calculus. Moreover, the following term can reduce non-deterministically:

$$a!\langle V\rangle\,.\,\mathbf{0} \mid a\!:\!\vec{V}_1 \mid a\!:\!\vec{V}_2$$

It is easy however, to ensure the correspondence of names in action prefixes with buffers, and the absence of multiple buffers per name, by imposing well-formedness conditions or by typing, but this is beyond the scope of this Section.

Other modifications are simple. We extend free names as follows:

$$\mathsf{fn}(l) = \emptyset \qquad\qquad \mathsf{fn}(a\,{:}\,V_1 \ldots V_n) = (\cup_{i\in 1..n}\mathsf{fn}(V_i)) \cup \{a\}$$

Then we add the following axiom to structural congruence:

$$(\nu a)\ a\,{:}\,\varepsilon \equiv \mathbf{0} \qquad \text{inaccessible buffer}$$

Modelling in the Asynchronous Calculi One observation that we can make immediately is that for deterministic behaviour, it is better to use multiple buffered channels and make sure that each process in a composition uses only one of the *input-output capabilities* induced by input and branching, and output and selection, respectively.

For instance, we can define the following processes, drawing as before from our example. We begin with the asynchronous extended Photographer process, $\mathsf{AsyncExtPhotoSrv}(a)$, as follows:

$$\mathsf{AsyncExtPhotoSrv}(a) \overset{\text{def}}{=} \begin{array}{l} a?(x_1)\,.\,x_1?(x_2)\,.\,x_1?(\mathit{image\_id})\,. \\ \quad \mathsf{let}\ \mathit{img} = \mathsf{DB.load}(\mathit{image\_id})\ \mathsf{in} \\ \quad \Big(\mu y\,.\,\lambda z\,.\,x_1 \triangleright \{\,\mathsf{nextFilter} : x_1?(f)\,.\,(\,\mathsf{let}\ z' = f\cdot z\ \mathsf{in}\ y\cdot z'\,)\,, \\ \qquad\qquad\qquad \mathsf{done} : \mathsf{DB.save}(\mathit{image\_id}, z)\,;\,\mathbf{0}\,, \\ \qquad\qquad\qquad \mathsf{upload} : x_1?(\mathit{dest})\,.\,\mathit{dest}!\langle z\rangle\,.\,x_2!\langle()\rangle\,.\,(\,y\cdot z\,)\,, \\ \qquad\qquad\qquad \mathsf{undo} : \mathsf{let}\ z' = \mathsf{DB.load}(\mathit{image\_id})\ \mathsf{in}\ y\cdot z'\,\}\Big)\cdot \mathit{img} \end{array}$$

Then we define an Editor:

$$\mathsf{AsyncEditor}_2(a,b,c) \stackrel{\text{def}}{=} \begin{array}{l} (\nu x_1)\ (\nu x_2) \\ \Big(a!\langle x_1\rangle \,.\, x_1!\langle x_2\rangle \,.\, x_1!\langle \mathsf{img42}\rangle \,.\, \dots \\ \quad x_1 \triangleleft \mathsf{upload} \,.\, x_1!\langle b\rangle \,.\, x_1 \triangleleft \mathsf{upload} \,.\, x_1!\langle c\rangle \,.\, x_1 \triangleleft \mathsf{done} \,. \\ \quad\quad x_2?(y) \,.\, x_2?(z) \,.\, \mathsf{print}(\textit{“upload complete”})\,;\, \mathbf{0} \\ \quad\quad \mid \quad x_1 : \varepsilon \quad \mid \quad x_2 : \varepsilon \quad \Big) \end{array}$$

The strategy we employed is to define two private channels with (νx_1) and (νx_2), with their corresponding buffers initialised to ε within the scope of the process, and program $\mathsf{AsyncEditor}_2(a,b,c)$ to send the first through channel a, and the second through channel x_1. The reason for this is that we want to ensure that once a process receives x_1, then the same process will also receive x_2, facilitating two buffered links between the two processes. If we had performed $a!\langle x_1\rangle \,.\, a!\langle x_2\rangle \dots$, then the two outputs could be read by different processes with access to a. This chaining of outputs is in fact the method used to encode the polyadic π-calculus (in which multiple values can be sent simultaneously) into the monadic π-calculus (in which only a single value is sent at each communication); for more details on that see [62, 80].

By using the two channels x_1 and x_2, designing each process so that one does all inputs on x_1 and all outputs on x_2, and vice versa for the other, we can ensure that the interaction is predictable. The conclusion is that, in general, when two asynchronous processes need to exchange values in both directions, *two buffered channels are preferred.*

One interesting consequence of the asynchronous semantics is that the ordering of outputs in $\mathsf{AsyncEditor}_2(a,b,c)$ does not correspond exactly with the ordering of inputs in $\mathsf{AsyncExtPhotoSrv}(a)$. In particular, at $\mathsf{AsyncEditor}_2(a,b,c)$ we have:

$$\dots x_1 \triangleleft \mathsf{upload} \,.\, x_1!\langle b\rangle \,.\, x_1 \triangleleft \mathsf{upload} \,.\, x_1!\langle c\rangle \,.\, x_1 \triangleleft \mathsf{done} \,.\, x_2?(y) \,.\, x_2?(z) \dots$$

and, considering that the recursion unrolls twice with $x_1 \triangleleft \mathsf{upload}$ from the above, we eventually have the following communications at $\mathsf{AsyncExtPhotoSrv}(a)$:

$$\dots x_1 \triangleright \{\, \mathsf{nextFilter} \dots x_1?(dest) \dots .\, x_2!\langle()\rangle \dots x_1 \triangleright \{\, \mathsf{nextFilter} \dots x_1?(dest) \dots .\, x_2!\langle()\rangle \dots$$

Observe that in $\mathsf{AsyncEditor}_2(a,b,c)$ the second $x_1 \triangleleft \mathsf{upload}$ happens before the expected $x_2?(y)$

that matches the third action of $\mathsf{AsyncExtPhotoSrv}(a)$, which is $x_2!\langle()\rangle$. This is not a problem, as the channels are separate and by the use of buffers there is no need to synchronise at each for communication to take place.

Next consider a composition of the processes as follows:

$$\begin{array}{lllll} & (\nu b_1)\ (\nu c_1)\ (\ \ \mathsf{AsyncEditor}_2(a,b_1,c_1) & \mid\ \ b_1:\varepsilon & \mid\ \ c_1:\varepsilon\ \ ) \\ \mid & \mathsf{AsyncExtPhotoSrv}(a) & \mid\ \ \mathsf{AsyncExtPhotoSrv}(a) & \mid\ \ a:\varepsilon \\ \mid & (\nu b_2)\ (\nu c_2)\ (\ \ \mathsf{AsyncEditor}_2(a,b_2,c_2) & \mid\ \ b_2:\varepsilon & \mid\ \ c_2:\varepsilon\ \ ) \end{array}$$

An interesting point here is that both instances of $\mathsf{AsyncEditor}_2(a,b_i,c_i)$ will interact with an instance of $\mathsf{AsyncExtPhotoSrv}(a)$ each, without interference, validating the strategy of sharing unique channels between pairs of processes, together with the separation of input-output capability between them.

To summarise, the untyped calculi with buffers enable (but do not guarantee) the programming of complex asymmetric protocols, and different interactions can be shielded from interference with each other, by a careful sharing of unique channels.

2.4 When things "go wrong": The Need to Discipline Processes

Until now we have been slightly biased: we designed all our examples so that nothing goes wrong. In fact there are so many things that *can* go wrong that we can only provide a small sample. There is nothing to prevent us from writing terms such as:

$$\mathsf{IncompleteEditor}_1(a) \overset{\text{def}}{=} \begin{array}{l} (\nu x)\ (\nu ack) \\ \Big(\mathsf{let}\ f = \lambda\, img\,.\, ack!\langle()\rangle\,.\,P\ \mathsf{in} \\ \quad a!\langle x\rangle\,.\,x!\langle \mathsf{img42}\rangle\,.\,x \triangleleft \mathsf{nextFilter}\,.\,\mathbf{0}\Big) \end{array}$$

and composing it with $\mathsf{PhotoSrv}(a)$ in:

$$\mathsf{PhotoSrv}(a) \mid \mathsf{IncompleteEditor}_1(a)$$

resulting in a stuck computation as $\mathsf{PhotoSrv}(a)$ expects more actions on x after $\mathsf{nextFilter}$ is chosen but $\mathsf{IncompleteEditor}_1(a)$ terminates with $\mathbf{0}$.

Similarly we can write:

$$\mathsf{PhotoSrv}(a) \mid \mathsf{Editor}_2(a,b)$$

and again the computation will eventually get stuck as $\mathsf{Editor}_2(a,b)$ selects upload which is not supported by $\mathsf{PhotoSrv}(a)$.

In general the ordering, sequencing, and expected values can be different than expected, resulting in stuck computation, and type errors in the case of typed calculi. Moreover, we can write processes such as:

$$\mathsf{NonDetEditor}_1(a) \overset{\text{def}}{=} \begin{array}{l} (\nu x)\ (\nu ack) \\ \Big(\mathsf{let}\ f = \lambda img\,.\,ack!\langle()\rangle\,.\,P\ \mathsf{in} \\ \quad a!\langle x\rangle\,.\,x!\langle\mathsf{img42}\rangle\,.\,(\,x \lhd \mathsf{nextFilter}\,.\,\mathbf{0} \mid x!\langle\mathsf{img24}\rangle\,.\,\mathbf{0}\,)\Big) \end{array}$$

composed in:

$$\mathsf{PhotoSrv}(a) \mid \mathsf{NonDetEditor}_1(a)$$

in which behaviour becomes non-deterministic, by performing actions in parallel, when $\mathsf{PhotoSrv}(a)$ has a sequential and in fact incompatible protocol, so again this may result in an error or a blocked communication which has no receiver.

In the asynchronous semantics, consider the case where we define:

$$\mathsf{BadAsyncEditor}_2(a,b,c) \overset{\text{def}}{=} \begin{array}{l} (\nu x_1)\ (\nu x_2) \\ \Big(\ldots \mid \quad x_1 : \varepsilon \quad \mid \quad x_2 : \mathsf{done} \quad \Big) \end{array}$$

in which the buffer where the other process will output contains a choice from the start, or the following:

$$\mathsf{NaughtyAsyncEditor}_2(a,b,c) \overset{\text{def}}{=} \begin{array}{l} (\nu x_1)\ (\nu x_2) \\ \Big(\ldots \mid \quad x_1 : \varepsilon \quad \mid \quad x_1 : \varepsilon \quad \mid \quad x_2 : \varepsilon \quad \Big) \end{array}$$

where buffer x_1 occurs twice causing non-determinism. Consider this process:

$$\mathsf{RudeAsyncEditor}_2(a,b,c) \overset{\text{def}}{=} \begin{array}{l} (\nu x_1)\ (\nu x_2) \\ \Big(\ldots \mid \quad x_1 : \varepsilon \quad \Big) \end{array}$$

in which the channel x_2 has no queue and therefore the other process will never be able to communicate. Next, consider this case:

$$\mathsf{CarelessAsyncEditor}_2(a,b,c) \overset{\text{def}}{=} \begin{array}{l} (\nu x_1) \\ \Big(\ldots \mid \quad x_1 : \varepsilon \quad \mid \quad x_2 : \varepsilon \ \Big) \end{array}$$

where the channel x_2 is not bound and thus another process can interfere with it. Finally, observe that in a term like the following:

$$\mathsf{CrazyAsyncEditor}_2(a,b,c) \overset{\text{def}}{=} \begin{array}{l} (\nu x_1)\ (\nu x_2) \\ \Big(\ldots \mid \quad \lambda z.x_1 : \varepsilon \quad \mid \quad x_2 : \varepsilon \ \Big) \end{array}$$

the buffer x_1 is trapped in a function, and cannot be accessed.

Therefore, although the calculi described in the previous sections have significant expressiveness and desirable features, they allow many classes of bad behaviour, and this motivates our investigation of a static verification discipline that will accept only safe interaction.

2.5 Sessions

Starting from 1994 with the works of Takeuchi, Honda and Kubo [81], and then Honda, Vasconcelos and Kubo [48], *Sessions* and *Session types* have emerged as a tractable and expressive theoretical substrate, which offers direct language support for high-level, type-safe and uniform abstraction subsuming a wide range of communication patterns.

A session is defined as a series of typed communications between *two* processes which form a meaningful logical unit, just like a web session between a browser and a server, created when a human user interacts with an e-commerce site, or like the intended protocol between the Editor and Photographer's systems in the case study we described.

There are two components to sessions: *session types* and the associated typing discipline, verifying the communication behaviour of a private link between processes, and *session primitives* that establish the actual link.

2.5.1 Session Types

Session types model interactions as an abstract structure, a process-like description consisting sequences of typed inputs, outputs, internal and external choices, and repetition. To facilitate typ-

$$
\begin{array}{rcll}
S & ::= & ![U].S & \textsf{output} \\
 & | & ?[U].S & \textsf{input} \\
 & | & \oplus[l_1 : S_1, \dots, l_n : S_n] & \textsf{selection} \\
 & | & \&[l_1 : S_1, \dots, l_n : S_n] & \textsf{branching} \\
 & | & \mathbf{t} & \textsf{type variable} \\
 & | & \mu\mathbf{t}.S & \textsf{recursion} \\
 & | & \textsf{end} & \textsf{ending} \\
 & & & \\
U, T & ::= & S \quad | \quad \dots &
\end{array}
$$

Figure 2.5: Session Types

ing, session types are associated to *communication channels*, and the behaviour of those channels throughout a program is verified against a given type. In Figure 2.5 we show a standard definition. The constructors of session types have a direct correspondence with the process constructors that implement the respective behaviour. For example, x in an input prefixed process:

$$x?(y) \,.\, P$$

will be given an input prefixed session type of the shape:

$$?[U].S$$

where U is the expected type of y, and S is the session type of x in P. Similarly for an output prefixed process $x!\langle V \rangle \,.\, Q$ inducing a type of the shape $![U]\,.\,S$ where U is a type assigned to V and S is the type of x in Q. Next, a process branching on a channel x, such as:

$$x \triangleright \{l_1 : P_1, \dots, l_n : P_n\}$$

induces a type of the shape:

$$\&[l_1 : S_1, \dots, l_n : S_n]$$

where the type of x in each P_i is equal to S_i. Selection on $x \triangleleft l_m \,.\, P$ induces a type $\oplus[l_i : S_i]^{i \in I}$ where $m \in I$, and S_m describes the use of x in P. In session typing the term:

$$\lambda x \,.\, x!\langle V \rangle \,.\, \mathbf{0}$$

will be given a functional (arrow) type of the shape:

$$![U]\,.\,\mathsf{end} \rightarrow T$$

where the session type end can be thought of as the session equivalent of the **0** process. Recursive functions induce recursive session types, typically with branching in order to have a termination condition, although this is not necessary. As a simple example, the variable x in the term:

$$\mu y\,.\,\lambda x\,.\,x!\langle V\rangle.(y\cdot x)$$

can be given a type:

$$\mu\mathbf{t}.![U]\,.\,\mathbf{t}$$

corresponding to the repetitive outputs.

Note that there is no session type $S \mid S$ corresponding to a parallel composition of communications over a single channel, which implies that each process possessing a channel must use it in an inherently deterministic way. Then, as we describe next, the different types assigned to the same channel in two processes within a session are compared, to determine their *compatibility*.

Capturing Symmetric Interaction

The essence of session typing is to ensure that the two behaviours associated with a session are deterministic with respect to each other. This implies that a certain symmetry should be applicable to the respective types, turning one into the other and vice versa. In the most basic setting we use a syntactic *duality* transformation, given a session type S, that produces the *dual* type $\overline{S}$, using the following rules:

$$\overline{![U].S} = ?[U].\overline{S} \qquad \overline{?[U].S} = ![U].\overline{S} \qquad \overline{\mathbf{t}} = \mathbf{t} \qquad \overline{\mu\mathbf{t}.S} = \mu\mathbf{t}.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

$$\overline{\oplus[l_1 : S_1,\ldots,l_n : S_n]} = \&[l_1 : \overline{S_1},\ldots,l_n : \overline{S_n}] \qquad \overline{\&[l_1 : S_1,\ldots,l_n : S_n]} = \oplus[l_1 : \overline{S_1},\ldots,l_n : \overline{S_n}]$$

Duality is an idempotent operation, that is, $\overline{\overline{S}} = S$, interchanging input and output, and also branching and selection, and for all other type constructors it is the identity function.

Example Types for the Scenario

Let us consider the processes $\mathsf{PhotoSrv}(a)$ and $\mathsf{Editor}_1(a)$ from our previous examples. Observing the structured communications in $\mathsf{PhotoSrv}(a)$, we can postulate that the session type of x within the definition (where it occurs free) will be as follows, assuming the necessary ground types:

$$S_1 = ?[\mathsf{String}]\,.\,\mu\mathbf{t}\,.\,\&[\,\mathsf{nextFilter} : ?[\mathsf{Image} \to \mathsf{Image}]\,.\,\mathbf{t}\,,\,\mathsf{done} : \mathsf{end}\,]$$

Then, in $\mathsf{Editor}_1(a)$, again observing the structure, the variable x can be typed with:

$$S_2 = ![\mathsf{String}]\,.\,\oplus[\,\mathsf{nextFilter} : ![\mathsf{Image} \to \mathsf{Image}]\,.\,\oplus[\,\mathsf{done} : \mathsf{end}\,]\,]$$

To achieve syntactic duality between the given types, we need to understand the type S_2 as *an instance of* S_2', defined as:

$$S_2' = ![\mathsf{String}]\,.\,\mu\mathbf{t}\,.\,\oplus[\,\mathsf{nextFilter} : ![\mathsf{Image} \to \mathsf{Image}]\,.\,\mathbf{t}\,,\,\mathsf{done} : \mathsf{end}\,]$$

Trivially we have that $S_2' = \overline{S_1}$. For the comparison between S_2 and S_2' to be obtained, we can consider the infinite expansion of the types, and utilise a *coinductive* method to verify the correspondence. Pierce's book [76] contains a detailed overview of the techniques and the mathematical background. Coinductive subtyping of recursive session types is first studied in the work of Gay and Hole [39], adapting the standard methods for IO-subtyping in the π-calculus, by Pierce and Sangiorgi [77]. Using a coinductive method, we can verify that the session usage in $\mathsf{ExtPhotoSrv}(a)$ is a *subtype* of the one in $\mathsf{PhotoSrv}(a)$, which enables the substitution of the first process in place of the latter, offering flexibility in programming.

2.5.2 Primitive Support for Establishing Sessions

From the desirable and undesirable process compositions in the previous sections, we can extract a first principle for safe program behaviour: there needs to be a *private link* between two communicating processes to ensure *determinism*. Moreover, the inherently deterministic session types cannot correspond to the non-determinism induced by uncontrolled parallel actions on a channel.

We already saw how this can be achieved: one of the processes can first create a fresh communication channel, unknown to any other process, and then this private channel can be communicated to the intended co-process over a known channel. Thus the only communication on unrestricted channels should be the exchange of a private link, through which all subsequent interactions take place. In process-algebraic terms, this corresponds to the principle of only allowing *bound* output over known (unrestricted) channels, where bound output is defined as the output of a locally restricted name. In π-calculus, we write this as $(\nu c)\ (\,a!\langle c\rangle\,.\,P\,)$ which can be used as in:

$$(\nu c)\ (\,a!\langle c\rangle\,.\,P\,) \quad | \quad a?(x)\,.\,Q$$

The above evolves, modulo alpha-conversion (uniform renaming of both c and instances of c in $a!\langle c\rangle\,.\,P$ to c' so that c' does not appear in Q), to:

$$(\nu c)\ (\,a!\langle c\rangle\,.\,P \quad | \quad a?(x)\,.\,Q\,)$$

The *scope* of (νc) has been extended (*scope extrusion*) to encompass Q, making c private to both P and Q. In other terms, this can be thought of as choosing a suitable identity for a private link between P and Q. Then the communication on a takes place, which is only possible after the scope of (νc) has been opened to encompass both processes, resulting in:

$$(\nu c)\ (\,P \quad | \quad Q\{^{c}\!/\!_{x}\}\,)$$

Now we arrived at a configuration in which c is uniquely shared between P and Q.

We can distill this process of private link creation in *session connection primitives* for both synchronous and asynchronous processes, while forbidding other types of interaction over non-private channels, to establish a principle where all communications except connection initialisation are performed through private links. Moreover, the explicit constructors make session typing easier to formalise and implement.

Primitives for Synchronous Sessions

A first attempt First, to separate private and shared channels, let us define a new class of *session channels* ranging over s, and redefine the communication rules in HOπ (and HO$\pi\varsigma$) to be:

(comm)

$$s?(x)\,.\,P \mid s!\langle V\rangle\,.\,Q \quad \longrightarrow \quad P\{^{V}\!/\!_{x}\} \mid Q$$

(HOπ-label)

$$s \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \mid s \triangleleft l_m.P \quad \longrightarrow \quad P_m \mid P \qquad 1 \leq m \leq n$$

(HOπς-label)

$$s \triangleleft l_m \,.\, Q \mid s \triangleright [\, l_i{=}\varsigma(x_i)\, P_i{}^{i\in I}\,] \quad\longrightarrow\quad Q \mid [\, l_i{=}\varsigma(x_i)\, P_i{}^{i\in I}\,].l_m \qquad m \in I$$

Thus, we cannot perform any communication on channels such as a any more, but only on this new class of session channels.

Then, without resorting to the different notations in the many works on sessions, which are discussed in the next chapter, we can formulate a basic *connection* primitive that establishes the desired private link between two processes, over a shared channel a, with the following dynamic semantics:

$$\text{(conn)}\quad \mathsf{connect}\ a(x)\,.\,P \quad\mid\quad \mathsf{connect}\ a(z)\,.\,Q \qquad\longrightarrow\qquad (\nu s)\ (\,P\{{}^{s}\!/\!_{x}\} \quad\mid\quad Q\{{}^{s}\!/\!_{z}\}\,)$$

Hence, we arrive at a new formulation where there is a class of shared names a that can *only* be used to establish a private link between two processes, with actual communications taking place using the new class of channels denoted by s.

Undesirable configurations Consider the following process adapted from Dezani *et al.* [31], further examined in the work of Vasconcelos and Yoshida [92]:

$$P \quad=\quad \mathsf{connect}\ a(x)\,.\,x?(y)\,.\,y!\langle V\rangle\,.\,x?(y_1)\,.\,\mathbf{0} \quad\mid\quad \mathsf{connect}\ a(z)\,.\,z!\langle z\rangle\,.\,\mathbf{0}$$

This process can be typed with the following types for x and z, respectively, within the body of the connections, assuming a type U for V:

$$\begin{aligned} S_x &= ?[\,![U]\,.\,\mathsf{end}\,]\,.\,?[U]\,.\,\mathsf{end} \\ S_z &= ![\,![U]\,.\,\mathsf{end}\,]\,.\,![U]\,.\,\mathsf{end} \end{aligned}$$

Now consider the reductions of P:

$$\begin{aligned} P &\longrightarrow (\nu s)\ (\,s?(y)\,.\,y!\langle V\rangle\,.\,s?(y_1)\,.\,\mathbf{0} \quad\mid\quad s!\langle s\rangle\,.\,\mathbf{0}\,) \\ &\longrightarrow (\nu s)\ (\,s!\langle V\rangle\,.\,s?(y_1)\,.\,\mathbf{0} \quad\mid\quad \mathbf{0}\,) \end{aligned}$$

The contractum is now typed with:

$$S_s \quad=\quad ![U]\,.\,?[U]\,.\,\mathsf{end}$$

This typing is unexpected, and breaks the property of *type preservation*, an issue further discussed in the aforementioned work. The reason is that we started with two complementary types for x and z but we ended up with a single type for s, as if a session can exist on its own. We can make an observation, however, on the cause of the problem, which is that we use the same channel name in both parallel processes. In session typing the two processes use the channel in different but complementary ways, but we have to take into account the possibility that they can reduce to a single process possessing an interleaved usage of the channel. In other words, we need to consider the possibility of *aliasing*. Hence, the solution, also present in the study of Gay and Hole [39], is to distinguish the channel name used at each process in a session, so that the above contractum can be typed with a type corresponding to the completion of the session (even though the term is stuck) since type safety is not violated.

A correct solution Let k be defined as either s or $\overline{s}$, with $\overline{\overline{s}} = s$. This is equivalent to the use of polarised channels s^+ and s^- in the work by Gay and Hole [39]. Next we redefine the rules, as follows:

(comm)

$$k?(x)\,.\,P \mid \overline{k}!\langle V\rangle\,.\,Q \quad \longrightarrow \quad P\{{}^{V}\!/\!_{x}\} \mid Q$$

(HOπ-label)

$$k \rhd \{l_1 : P_1, \ldots, l_n : P_n\} \mid \overline{k} \lhd l_m.P \quad \longrightarrow \quad P_m \mid P \qquad 1 \leq m \leq n$$

(HOπς-label)

$$k \lhd l_m\,.\,Q \mid \overline{k} \rhd [\, l_i = \varsigma(x_i)\, P_i{}^{i\in I}\,] \quad \longrightarrow \quad Q \mid [\, l_i = \varsigma(x_i)\, P_i{}^{i\in I}\,].l_m \qquad m \in I$$

We then establish connections using the associated channels, providing one to each process:

$$\text{(conn)} \quad \mathsf{connect}\ a(x)\,.\,P \quad \mid \quad \mathsf{connect}\ a(z)\,.\,Q \quad \longrightarrow \quad (\nu s, \overline{s})\ (P\{{}^{s}\!/\!_{x}\} \quad \mid \quad Q\{{}^{\overline{s}}\!/\!_{z}\})$$

Note that in the above, $(\nu s, \overline{s})$ does not declare two independent channels, but rather the two *endpoints of a single session*. Now the typing of P and of all the processes to which P reduces is correct.

As a final note, an alternative but more restricted solution, in the context of an imperative class-based language with thread spawning and no parallel composition at the level of user syntax, is given by Dezani *et al.* [31].

Primitives for Asynchronous Sessions

For the asynchronous semantics, we can easily adapt our previously given buffered communication rules to use the above solution, with a connection rule:

$$\textsf{(conn)}$$
$$\textsf{connect}\ a(x).P \ \mid\ \textsf{connect}\ a(z).Q \ \longrightarrow\ (\nu s,\bar{s})\ (P\{{}^{s}\!/\!_{x}\} \ \mid\ Q\{{}^{\bar{s}}\!/\!_{z}\} \ \mid\ s\!:\!\varepsilon \ \mid\ \bar{s}\!:\!\varepsilon)$$

Now we also have the two associated buffers initialised to empty, as required.

2.6 The Thesis

Both *higher-order processes* and *objects* provide powerful primitives for structuring interactions in a concurrent setting. A common denominator of many useful protocols is the need to facilitate private connections between two or more programs, and a deterministic behaviour that is characterised by compatibility of the composed systems. *Sessions* and *session types* offer a precise discipline that can be used to statically verify the correctness of many interesting applications. However, until our recent work [63], it was not possible to type higher-order processes in which mobile code may make use of session communications. For example, if *ack* in $\textsf{Editor}_1(a)$ belongs to a session, then the systems in the literature before our work cannot type the process. In addition, in an asynchronous setting, before our work [64], it was impossible to type a composition of processes such as:

$$\textsf{AsyncEditor}_2(a,b,c) \quad\mid\quad \textsf{AsyncExtPhotoSrv}(a)$$

where the order of actions is different than expected but safety is not violated, while rejecting unsafe asynchrony, in a higher-order setting. Finally, we address the question of how to integrate sessions and objects in a foundational calculus, which has also not been done before.

Session typing is important for concurrent programming, as much as typing theories for functional and object languages have proved to be essential for sequential computation: sessions control and discipline the power of processes, which is what is needed to incorporate them in practical settings. Thus, a formalisation of sessions applicable to the fundamental range of primitives found in functions, mobile processes and objects, can constitute a basis for the development of future programming languages in which verifiable concurrency is — and it *must* be — a core feature.

3 Related Work

Overview *We present the most important works on sessions in the context of different languages, as well as closely related approaches, providing an exposition to the state of the art in this area of type-based structured protocol verification. Moreover, we offer pointers to references on implementation-related issues and session-based prototypes, exposing the practical aspects of sessions.*

3.1 Session Typing for Binary Sessions

The genesis of session types can be traced to the work of Honda [47] in 1993. Then, the formulation matured into its present form, starting from 1994 with the works of Takeuchi, Honda and Kubo [81], and then Honda, Vasconcelos and Kubo [48].

The study of session typing systems is now wide-spread due to the need for structured type-safe communications in various distributed computing scenarios. Below we give the most closely related work.

3.1.1 Sessions in CCS and π-calculus

In the original work of Honda [47] we find a formulation built upon the now revived concept of sessions with generic internal and external choice, and of type compatibility based on a bisimilarity on a labelled transition system. The language endowed with these sessions is a π-calculus like process algebra. Interestingly, in this work we can also find the first reference to *deadlock-free* sessions, based on a characterisation of terms.

Then, in the work of Takeuchi, Honda and Kubo [81], the process algebra $\mathcal{L}$ is presented, resembling CCS/CSP but with explicit session initiation primitives, deterministic label-indexed branching (external choice with input of label) and selection (internal choice with output of label), recursive process definitions, and a primitive for the dynamic creation of parallel processes (threads). The concept of label-indexed choice resembles record-field selection and method selection in object languages, and provides a powerful control flow mechanism that retains determinacy in a simple framework. The types are finite (no recursion), and there is no name passing for sessions, prohibiting certain classes of dynamic topology from being programmed. Finally, the idea of typing the two usages of a session channel using distinct polarities is introduced in this work.

Honda, Vasconcelos and Kubo [48] address the shortcomings of the aforementioned work, and present a language based on a polyadic π-calculus, supporting both name passing (where a session channel is being "thrown") and recursive sessions (implemented by passing session channels as arguments to recursive process invocations), in addition to the capabilities of $\mathcal{L}$. One note on delegation (session passing) is that it is not implemented with substitution, but rather with α-conversion, hence every communicated name is fixed; this is not a significant restriction but it does imply a runtime test to convert names if needed.

As we saw in the previous chapter, some systems suffer from a subtle maltreatment of session channels in typing, and that invalidates the basic theorems of type soundness and type safety. We already identified the root of the problem in the absence of polarities, combined with the presence of session delegation which enables the aliasing of sessions that are treated as independent at the stage of typing. Yoshida and Vasconcelos [92] revisit the previous work [48], which does not suffer from this problem due to the more limited alpha-conversion based delegation method used which forbids a free channel from being received in the same scope. They extend the language with the more powerful form of delegation which causes the problem, and provide a correct formulation with some additional smaller corrections. Their extension of the original work to allow a more liberal style of session passing, where the names are not fixed, enables arbitrary session channels to be received, by utilising polarised endpoints in the style of Gay and Hole [39], thus avoiding situations where the two ends of a session are merged (aliased). Also, there is no need for runtime α-conversion. The notion of polarities at the typing level, but not at the term level, is first utilised in the work of Takeuchi, Honda and Kubo [81].

Recently, Giunti *et al.* [42] showed that although the polarity-free languages may not enjoy subject reduction, type safety is not violated. Moreover, they allow unrestricted (non-linear) use of session channels in some harmless settings, such as when testing the identity of a channel (a feature not found in other session systems) even after it has been delegated away.

3.1.2 Sessions in Functional Languages

Neubauer and Thiemann [68] encode sessions in Concurrent Haskell. Vasconcelos, Gay and Ravara [86], add session primitives to a concurrent multi-threaded functional language. Their language supports sending of channels and higher-order values (not containing free session channels), labelled branching and selection, recursive sessions and channel sharing. It has an explicit multi-threading primitive (fork) and explicit stores. Gay and Vasconcelos [40] define a functional language with sessions, following substructural techniques as discussed in § 3.2.

3.1.3 Sessions in Higher-order Processes

Sessions have been integrated in a Higher-order π-calculus by Mostrous and Yoshida [63, 64]. The systems in these papers combine the use of linear λ-calculus and standard session typing, treating free session channels in mobile code as linear components that are only used in one unit of abstraction, which may include a sequence of session actions identified as a larger usage. Mobile code with free sessions is then treated as a linear function, ensuring that its contents are used exactly once.

3.1.4 Sessions in the Ambient Calculus

Garralda, Compagnoni, and Dezani [36], define the language BASS that introduces session types to boxed ambients, preventing session interruption when an ambient crosses nested boundaries. Essentially, mobility is forbidden when there are pending sessions between an ambient and its parent, ensuring that sessions remain safe; otherwise once an ambient boundary is dissolved communications with its original enclosing scope can be lost or become mixed with the new context.

3.1.5 Sessions in Object-oriented Languages

CORBA Vallecillo, Vasconcelos, and Ravara [83] give a preliminary account of how sessions could be incorporated into CORBA interfaces. Their approach governs the use of remote method invocations using session typing that mandates the sequencing of calls, facilitating fine-grained control of remote interfaces. The ideas are expository and not fully formalised, but can inform further work in the area.

Class-Based Languages There are a number of works on class-based languages with session typing, which are very related, albeit different, in their approach. We present the most related ones in chronological order.

Session typing for a multi-threaded Java-like language has been studied in our previous work by Dezani *et al.* [31, 30] and its distributed precursor [32]. The language in [31, 30] is more of an extension rather than an integration of objects and sessions. There are no recursive session types or linear objects, but on the other hand, in their language the property of *progress* holds, i.e., there are no deadlocks. Note that due to synchronous semantics this property comes at the cost of rejecting all session interleavings. Another achievement of this work is a type inference system for session environments and types. Their sessions are established over shared channels, and session bodies are not identified with methods. This is one of the reasons session iteration is used, instead of recursion.

In the extended and updated version of the above work by Dezani *et al.* [30], we relax the condition on interleavings, and achieve progress in a setting where in every sequential execution there can be multiple sessions but only one may perform (blocking) input actions.

Coppo, Dezani, and Yoshida [26] define an asynchronous class-based language with session typing. Their reduction semantics for input and output use buffers, and their session types are similar to those of Dezani *et al.* [31, 30]. The essence of their work is an effect system that decides if a term has the *progress* property, restricting certain classes of interleavings.

Drossopoulou, Dezani, and Coppo [33] describe a class-based language in which sessions are identified with methods. In their approach, session invocation combines a method of an object (which is spawned) and a block of code that will interact with the method body. Communication is asynchronous using buffers. Their main primitive is written $a.l\{b\}$, which invokes the method (session) l of object a and places it in parallel to the code b; a fresh session channel is created and is shared between the method and the code that interacts with it. In their system there is only one session per scope and the session channel is implicit. Hence, endpoints are not first class values, although there is a form of delegation resembling synchronised method invocation. Their branching primitive decides which path to follow based on the class of a received value, and branches can contain the keyword continue for iteration. However, it is not clear how this data-driven branching approach would scale to object-oriented recursion, especially as threads are created for every method call. On the other hand, their sessions enjoy progress like the systems by Dezani *et al.* [31, 30] and by Coppo, Dezani, and Yoshida [26]. Capecchi *et al.* [18] extend the above work by Drossopoulou, Dezani, and Coppo [33], to include parametric polymorphism, or generics.

Bettini *et al.* [8] improve the above work, with the introduction of *union* types of the shape $U_1 \vee \ldots \vee U_n$, which are used to drive their class-dependent branching and selection in a more flexible and compositional way, since a received value can command the choice of a subsequent selection within a program.

More recently, Gay *et al.* [41], proposed a modular approach to sessions in distributed class-based object-oriented programming, building on their previous work on dynamic interfaces [87].

Their system integrates the idea of *typestates* as a way to type classes that implement sessions: the availability of methods is non-uniform, but rather follows a set of parameterised states that specify the visible interface of an object, which changes after every method invocation. In their system there are no session endpoints (but there are shared channels for the initiation of a session). The trace of method invocations themselves effectively implements the session, allowing mutually recursive definitions, and the evolving interface of the object ensures that the session behaves as required by its type. The operational semantics specify a synchronous communication model.

An interesting aspect of their work is that it is safe to store a session endpoint in an object field, and in fact this enables different methods to access the session, during a single run. They allow this because their typing system ensures that the reference to the object is linear and the object behaves in accordance with its typestates; the use of the stored session channels remains deterministic. In their language, self-application within methods requires type annotations, reflecting different conditions when dealing with method invocation from inside and from outside of an object, respectively. In their formulation there are no threads but a "spawn" primitive is provided that places a new configuration in parallel, with similar operational effect to local threads. Finally, a decidable typechecking algorithm and a prototype implementation are provided, indicating the practical usability of the language.

3.2 Linear Type Theory Techniques in Sessions

The precise conditions restricting the usage of sessions within terms have prompted some research into Linear type theory approaches to session typing. These works draw from the techniques of Linear λ-calculus, treating session channels similarly to linear variables. See Walker's exposition [89] for a detailed account of the techniques for λ-calculus, which are based on the idea of *substructural* type environments, where weakening, needed when a variable is not used but appears in the environment, and contraction, required when a variable is used more than once in the subterms of a term, are explicitly forbidden.

The use of such linear techniques at the core of a session typed π-calculus is described in detail in the work of Vasconcelos [85]; for a more extended technical exposition see the article by Gay and Vasconcelos [40], where the language is a functional calculus with session primitives. Their typing uses standard linear function types and linear pairs, re-binding session variables after every action so they can be typed with the remaining session type in the process continuation. Algorithmic type checking is achieved using the standard linear type theory techniques. The work by Gay and Vasconcelos [40] follows an asynchronous, buffered mode of communication; we return to this later.

A different approach in which sessions are not reduced to linear usages, but where code that contains free sessions is treated as a linear function, appears in the work for mobile processes by Mostrous and Yoshida [63, 64], mentioned previously.

3.3 Alternative Formulations

Channel Dependent Sessions The paper by Mostrous and Yoshida [63] provides an alternative formulation of session typing based on the *channel dependent types* of Yoshida [91]. Our comparison between the two approaches via [63, Theorem 4.3] which defines the embedding of the linear typing system into the channel dependent one makes the relationship between controlling usage of functional variables and effects of channel accessibility clear: the channel dependency system types more processes, while the linear typing approach is simpler, it requires less type annotations, and is more tractable. This line of study has not been explored in the previous literature. Moreover, it can provide insight into the extension to distributed (location-aware) processes such as those of the SAFEDPI calculus with channel dependency types of Hennessy, Rathke, and Yoshida [45], developed further in Hennessy's book [44].

Foundations of Session Types Recently, an alternative formulation was proposed by Castagna *et al.* [23]. The session types in this system consist internal and external choice, and also input-output followed by a continuation of the session. Due to the use of standard choice constructors, where external choice is not constrained to be input-prefixed (contrary to label-indexed branching in the standard approach), and internal choice is similarly free from the duty to announce the selection by starting with an output (of a label in the standard approach), there is a need to precisely describe more complex dualities than usual. For example, if a process uses a session with type $![\mathsf{int}].\mathsf{end} + ![\mathsf{bool}].\mathsf{end}$ where $+$ denotes external choice (or branching in the standard approach), then a *dual* should be able to handle both possible outputs in a type-directed way. The authors solve this by adding boolean combinators only used to more accurately describe the carried types of input and output actions, for example $?[\mathsf{int} \vee \mathsf{bool}].\mathsf{end}$ is the type of a channel that is used to receive either an int or a bool, and is a suitable co-type for the above external choice. Also, duality is not syntactic but rather behavioural, and for each session type there can be a (possibly empty) set of duals. Session types with a non-empty set of dual co-types are called *viable*, while those with no duals are undesirable. Note that for higher-order sessions (session passing) the boolean operators remain applicable, producing extended session types called *sieves*, and resulting in a rich language of interactions.

A set-theoretic semantics is given for subtyping based on the interpretation of the session types based on the values that can inhabit the input-output actions and additionally, the conditions imposed by boolean operators on those sets of values. Duality is defined using a relation on a labelled transition system, and the definitions of *subsessioning* and *subsieving* (which correspond to subtyping) are formalised in a coinductive framework. The coinductive and denotational notions of subsessioning coincide, and the set-theoretic definitions for subsessioning are algorithmically decidable. Then, a typed π-calculus variant is given, demonstrating that their system can be used to type sessions in a language without special primitives, consisting standard (internal and external) choice, and (bound) outputs for establishing sessions. Duality is enforced for the initiation of a session. The language enjoys the property of *progress* (absence of deadlock in a constrained universe of interleaved sessions), by placing restrictions on the use of interleaved inputs which may block against interleaved outputs appearing in different order.

Session Types at the Mirror Following a similar approach, Padovani [73] defines a system where the main concept is to denote session types as value-passing CCS processes that describe precisely, within a labelled transition system, how processes in a term language use a session. First, the usual linearity constraints of session types are relaxed, allowing a parallel composition of session usages corresponding to a copied session channel used by multiple processes. Second, there is internal and external session choice which is not indexed by labels but rather follows the actual non-deterministic choices in the term language. Third, there is a session type representing failure, which is used to identify that an undesirable state has been reached. The first and third points are not present in the previously discussed work by Castagna *et al.* [23], and for simplicity in this system there are no *sieves* consisting boolean combinations of session types. The author defines a *subsessioning* relation that acts similarly to subtyping. Well-typed compositions use *viable* session types, which as before are types for which a co-type exists. The concept of *session completeness* captures the familiar concept of duality, as a special case. A typing system is presented for a π-calculus based process language, for which we can make the following essential observations. First, a process receiving a session channel can only use that specific channel in the continuation, for type preservation (this is similar to the receive-and-spawn technique for delegation in the object language by Dezani *et al.* [31, 30]), since there are no *polarities* to guard against typing aliased channels in a contractum. Second, a delegated channel can continue to be used at the sender, yielding a parallel non-deterministic usage between sender and receiver. Finally, parallel composition of processes demands that both use the same session channels. In general, this approach views control flow and data communication as different, and (due to parallel-composed session types) drops the linearity constraints which are the cornerstone of determinism in sessions, yielding a formalism closer to the behavioural types of Igarashi and Kobayashi [52].

Asymmetric Client Server Interactions Barbanera, Capecchi, and de' Liguoro [5], inspired by works on *contracts* (which we discuss at the end of this chapter), define a session typed π-calculus in which the client can perform an initial *prefix* of the expected session with the server. The prefix relation $\leqslant$ essentially allows a session with end terminals to be considered compatible to one where on each ended part there are further actions, implemented with an axiom $\mathsf{end} \leqslant S$ supported with congruence rules for the other session type constructors, propagating deep into the session structure. A condition of *weak compliance* is formulated based on a labelled transition system, stating that a server may not perform less actions than the client, thus validating the desired principles of the design.

One possible objection to this design is that some protocols may require a strict correspondence of client-server actions, for example in a banking transaction, and therefore it would be useful to have allowed session types to be demarcated as requiring full completion, precluding the use of the prefix relation; anyway, this is an easy addition to their theory, and simplicity outweighs the benefits of trivially extending it.

3.4 Asynchronous Communication in Sessions

The work of Gay and Vasconcelos [40] defines a functional language with asynchronous sessions, where the reduction semantics are using buffers, hence message sending is non-blocking. The two buffered endpoints of a session are associated by explicitly storing an entry for the "other" endpoint together with the values of each buffer; this functionality is the same as the use of two polarised session channels for distinguishing two endpoints (similarly to Gay and Hole [39]). An achievement of their work is that they calculate the upper bound on the size of the queues, with important ramifications for efficiency and static memory allocation. Several other works discussed in this chapter utilise buffered semantics, for example by Neubauer and Thiemann [69], Carbone, Honda, and Yoshida [49], Bonelli and Compagnoni [10], Coppo, Dezani, and Yoshida [26], Bettini [9], Mostrous, Yoshida, and Honda [65], and Mostrous and Yoshida [64].

3.5 Session Subtyping and Polymorphism

Session primitives can be smoothly integrated with traditional *subtyping* of object and functional languages, to obtain a more flexible behavioural composition that brings sessions frameworks closer to software engineering. The first study of subtyping in sessions is the work of Gay and Hole [38, 39]. The authors introduce standard value subtyping, such that for example an output $![U]$ can be performed instead of $![U']$ when U and U' are in a suitable subtype relation. Selection and branching is subtyped similarly to records, allowing less choices (than those demanded by the type) to be made in a process, and dually, more branches to be offered. Recursion imposes a coinductive subtyping method which is first adapted to sessions in their work, following the standard techniques for IO-subtyping for the π-calculus by Pierce and Sangiorgi [77].

Gay [37] takes the previous work a step further, and introduces bounded polymorphism to session types. Using this system, types such as $\&[\,l\,(\mathsf{int} \leqslant X \leqslant \mathsf{real}) :?[X]\,.\,![X]\,.\,\mathsf{end}\,]$ are allowed, in this example denoting the reception of a value of some type X which is a supertype of int and a subtype of real, followed by the sending of a value of the same type. The authors chose to annotate branching (at the type and term levels) with the bounds declarations, so that the information is “piggybacked” onto the communicated label. The basic algorithmic representation of the bounded subtyping is proved decidable. In conclusion, this system facilitates fine-tuned control over polymorphic communications.

Following the above work, Capecchi *et al.* [18] include parametric polymorphism (generics) into a class based object language with sessions.

3.6 Asynchronous Subtyping

Our recent work by Mostrous, Yoshida, and Honda [65] developed a new subtyping, *asynchronous subtyping*, founded on the ordered asynchrony of inputs and outputs, respectively, that arises with the use of buffered session channels. This subtyping characterises compatibility between classes of permutations of communications within asynchronous protocols, offering greater flexibility in programming. The target system is based on the multiparty session calculus of Carbone, Honda, and Yoshida [49] (discussed later); however, it does not support *higher-order sessions* (delegation) and *higher-order code* (code mobility). Both of these features provide powerful abstractions for structured distributed computing. Mostrous and Yoshida [64] expand the previous theory in a framework based on the Higher-order π-calculus, allowing the same communication order permutations in the presence of code mobility, effectively incorporating higher-order sessions and linear functions into asynchronous subtyping, and therefore addressing the previously open issues. As a simple example, these theories admit $\mu\mathbf{t}\,.\,![U_2]\,.\,?[U_1]\,.\,\mathbf{t}$ as a subtype of $\mu\mathbf{t}\,.\,?[U_1]\,.\,![U_2]\,.\,\mathbf{t}$, but not the other way around, because moving an input ahead of time might block unless if the dual endpoint performs the respective output also in advance, which is not guaranteed. Thus, the constraints that forbid subtypes that are inhabited by processes that may block can be regarded as a safety guarantee, protecting the desired invariant that a session should not be blocked on its own, that is, without depending on another interleaved session. This has practical considerations since the replacement of a process with one that uses sessions according to subtypes (compared to the original) will not block if the original process did not also block.

As a final note, a similar concept of session actions appearing in a different than expected (by duality against the other end of the session) order is seen in the unpublished work of Neubauer and Thiemann [69], as part of an *acceptance relation* on traces induced by session types within a π-calculus based language.

3.7 Progress and Deadlock-Freedom in Sessions

By *progress* in the context of a session language we mean the deadlock-free execution of multiple interleaved sessions. A simple counterexample to such progress is the following composition (to which a well-typed initial program may arrive during reduction):

$$s_1?(x)\,.\,s_2!\langle V_1\rangle\,.\,\mathbf{0} \;\mid\; \overline{s_2}?(z)\,.\,\overline{s_1}!\langle V_2\rangle\,.\,\mathbf{0}$$

in which both sessions, although type-safe, are waiting on each-other to output a value, ad infinitum. This notion of session progress is first considered in the work of Dezani *et al.* [31, 30], and a solution is provided in the form of a restriction on the interleaving of sessions.

Coppo, Dezani, and Yoshida [26] reconsider the problem in the context of a class based object language with buffered communications. In this approach, the authors revisit and successfully relax the interleaving restriction, allowing some of the interleavings in which only one session is expecting inputs, but in which many may perform outputs. This is possible due to the use of buffered semantics for communication, since output becomes a non-blocking action, and hence one that is sometimes safe to interleave.

Dezani, de' Liguoro, and Yoshida [29] formulate a new solution in a synchronous π-calculus setting, defined as an *interaction* typing system, and utilising a generic label-based ordering that is used to detect circularities in interleaved binary sessions, as those can lead to a deadlocked state.

Contrary to the typing method of Igarashi and Kobayashi [52] (for the π-calculus), the labels do not originate as annotations in the term language, but rather appear as fresh assignments in the typing derivation, lessening the programming burden otherwise imposed. Notably, only one session can input values in any given sequentially executed scope. Bettini *et al.* [9] extend the previous methodology to multi-party sessions.

The property of progress is also considered in the typed Conversation calculus of Caires and Vieira [17]. In this system labels are attached to session actions in a natural way, and are reused in the progress conditions to ensure acyclicity. Also, a session that has been delegated can continue to be used at the sender's continuation, providing an extra degree of flexibility while preserving the desirable deadlock-freedom properties.

3.8 Correspondence Assertions and Logics for Sessions

The basic systems view sessions as independent entities interleaved within a program, and make no effort to verify the intended *interdependencies* between different sessions that may interact together. For example, ignoring session constructors, consider a simple process $a?(x).b!\langle x\rangle.c!\langle 5\rangle.\mathbf{0}$, and its variation $a?(x).b!\langle c\rangle.x!\langle 5\rangle.\mathbf{0}$; session typing does not distinguish the two processes when the types of x and c coincide. Bonelli, Compagnoni and Gunter [11] address this consideration, by introducing *correspondence assertions* to a session-based calculus, offering a system that, for instance, distinguishes the above terms at the type level. This is achieved by adding constraints to sessions, such as one mandating that a value received on one session (a from above) is the exact same value subsequently sent in another (b from above).

Berger, Honda, and Yoshida [7] developed a modal logic for session-typed mobile processes. Their work studies an extension of Hennessy-Milner logic for typed π-calculi, giving a sound and complete characterisation of representative behavioural equivalences on typed processes; three compositional proof systems are obtained, characterising the May/Must testing preorders and bisimilarity. Using their logical framework, fine-grained properties of processes can be embedded into the specification (against which the implementation is verified), such as, for example, the property that in a bank transaction modelled as a session the amounts are as expected after every action. Using this method, a much finer control of processes is achieved, compared to correspondence assertions.

3.9 Exceptions for Error Handling in Sessions

Carbone, Honda, and Yoshida [20] introduce *exceptions* into a process language with sessions, using a familiar try/catch block structure. Their main contribution is the uniform propagation of exceptions to all affected processes (due to session nesting) which can be executing asynchronously evolving sessions, and the incorporation of exception types that validate potentially error-raising protocols. Vieira, Caires, and Seco [88] incorporate similar exception handling in their Conversation calculus, discussed at the end of this chapter.

3.10 Implementations

Functional Languages Neubauer and Thiemann [68] developed an encoding of sessions in the type system of Haskell. Mostrous [66] implemented an initial prototype for a subset of OCaml extended with finite session types (without branching) used for the typechecking of client-server socket based connections. Corin *et al.* [27] introduced session types to $F\sharp$, an implementation of a ML dialect. The work describes a system for ensuring security of multi-role sessions in the absence of trust. Session types are compiled to cryptographic protocols in a way such that during execution every party is guaranteed to play their role. Runtime verification is used to detect behaviour incompatible with a session.

Object Languages The Masters thesis of Hu [50] and the subsequent work by Hu, Yoshida, and Honda [51] have been investigating the incorporation of session types with Sockets in Java. Communication is asynchronous and the implementation has been measured to have a very small performance overhead compared to untyped socket communication. Its language and runtime is also extended to work under various transports such as shared memory and HTTP/HTTPS [51]. More recently, Gay *et al.* [41] implemented a prototype of their language for sessions in distributed class-based object-oriented programming, including a decidable typing algorithm.

An Operating system Fähndrich *et al.* [34] use general ideas from sessions to facilitate efficient and reliable message-based communications in the Singularity operating system. Behaviour in this system is defined in *contracts*, that contain definitions that form a state machine of desired message exchange patterns. Messages encapsulate asynchronous method invocation, and consist of information on which method should be invoked, along with the actual arguments to use, when the message is received. Values are exchanged using bidirectional channels, where each channel has two explicit endpoints. At the endpoints, the specific methods required for each state of the contract are defined. Asynchronous transmission is implemented using message queues. Their system has the property that each endpoint can only be used by a single thread at a time, which corresponds to the usual conditions of linearity, and messages at the endpoint queues are always ordered. Also, they allow to send channel endpoints, which corresponds to higher-order sessions. When different messages can be received, they use a form of switch to group the program behaviours for each case. Their contracts are verified statically.

3.11 Sessions in Industry Specifications

At the industry specification level, languages with variants of session types have been used in the W3C CDL (Choreography Web Description Language) [90, 21, 19] and ISO UNIFI (International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme) [82]. From these experiences, we find that not only type checking by session types after writing a protocol, but also declaring session types before compilation, greatly helps programmers implement error-free interactions.

3.12 Multi-Party Sessions: Typing Protocols with Many Participants

The first papers formalising sessions and session types with more than two participants are by Carbone, Honda, and Yoshida [49] (which supersedes a more preliminary version [19]), and by Bonelli and Compagnoni [10].

In [49], the concept of *multi-party* session types is introduced, acting as a global specification consisting participant-directed messages and selections. For example, the global type $G = \mathtt{p} \to \mathtt{p}' \colon k\,\langle U\rangle; G'$ says that participant $\mathtt{p}$ sends a message of type U to channel k (represented as a natural number) received by participant $\mathtt{p}'$ and then interactions described in G' take place. From the global viewpoint of G, the first prefix represents both an output and an input, and the *local* types representing the viewpoint of each participant can be recovered using a *projection* operator $G \upharpoonright \mathtt{p}$ which, given a participant identity $\mathtt{p}$, produces a type with the (directed subset of the) session protocol involving $\mathtt{p}$ as sender or receiver. For example, $(\mathtt{p} \to \mathtt{p}' \colon k\,\langle U\rangle; G') \upharpoonright \mathtt{p} = k![U]; (G' \upharpoonright \mathtt{p})$, $(\mathtt{p} \to \mathtt{p}' \colon k\,\langle U\rangle; G') \upharpoonright \mathtt{p}' = k?[U]; (G' \upharpoonright \mathtt{p}')$ and $(\mathtt{p} \to \mathtt{p}' \colon k\,\langle U\rangle; G') \upharpoonright \mathtt{q} = (G' \upharpoonright \mathtt{q})$. Thus, local types represent the ordered use of (possibly multiple) channels by a single process. Session connections are established between multiple participant processes, using a multi-participant request that interacts, synchronously, with a number of session acceptance primitives, followed by the instantiation of a number of buffers used within the protocol. The number of buffers may be different than the number of processes in a single session. Beyond the usual safety properties, the authors identify *session fidelity* as an important property: the actions of a typable process exactly follow the specification described by the global type.

The work [10] follows a similar approach based on a distributed calculus where each channel connects a master endpoint and one or more slave endpoints. The authors utilise annotation labels (like participants above) on (global) session types, and a *simplify* operator acting similar to projection. *Trace* types describe the behaviour of individual participants in a session corresponding to local types. Contrary to [49], this language does not allow higher-order sessions (delegation). The basic soundness and safety results hold as expected.

In the recent work by Mostrous, Yoshida, and Honda [65] we generalise the theory of multiparty session types of Carbone, Honda, and Yoshida [49] with asynchronous communication subtyping, which allows partial commutativity of actions offering greater flexibility and a way to identify safe optimisations in message choreography. As a complementing result, we show a type inference method for deriving the principal global specification from end-point processes which is minimal with respect to subtyping. The resulting theory allows a programmer to choose between a top-down and a bottom-up style of communication programming, ensuring the same desirable properties of typable processes. The top-down approach in multiparty session types is first studied in the work of Carbone, Honda, and Yoshida [49], but local refinement (asynchronous subtyping) is not proposed there. The problem of synthesising a global specification from endpoint behaviours has been an open question since the inception of the notion of global descriptions for business protocols (see Choreography Description Language [90]), and has been posed as an open problem in several previous works mentioned above [10, 49, 9]. Inference of principal types for simplified binary sessions is studied in the work of Mezzina [57], but in the context of Service Oriented Computing languages, discussed in the next section. Finally, we recall that a typing system offering a strong progress property in multi-party sessions is studied by Bettini *et al.* [9].

3.13 Service Oriented Computing

Castagna, Gesbert and Padovani [24, 25] study formal theories of *contracts* specifying and using multiparty interaction structures other than multiparty session types; specifically, the authors utilise CCS-like processes as a type representation. The subsequent work by Padovani [72] extends [24] with a treatment of asynchronous behaviours using orchestrators, through the use of bounded buffers that control message flows between a client and servers. A defining characteristic of their approach is that a client can choose to fulfil only some initial part of an interaction with a server, as long as the corresponding communications agree behaviourally.

Conformance and refinement based on agreement of clients to service specifications is studied in the work of Bravetti and Zavattaro [14, 15], using a synchronous CCS-based calculus as a contract language, and testing-preorders to check sub-contract compliance. Neither type-checking of end-point processes using projected contracts nor a bottom-up strategy is presented there.

The work by Bruni *et al.* [16] proposes a distributed calculus with sessions, which act as an enclosing context constraining (intra-session) communications to be between processes belonging to it. Services can be invoked, placing the service instance under the client session, therefore ensuring multiple subsequent interactions (over otherwise known channels) are between the same client and service pair. Sessions can be *merged*, allowing more processes to interact dynamically. Locations represent logical groupings of processes/services and allow direct intra-site communications crossing the boundaries of sessions. Reduction is defined via a labelled transition system, and terms are related by a weak bisimulation, useful for equating abstract specifications and more concrete implementations. Unlike type-based systems, there are no guarantees that executions are well-behaved.

The work of Vieira, Caires, and Seco [88] presents the Conversation calculus, a language for service oriented computing, by extending the π-calculus with context-sensitive interactions, equipped with service and request primitives and local exceptions. A crucial difference compared to standard session approaches is that endpoints are not channels, but rather interactive processes encapsulated within "conversation" contexts (like in the above work by Bruni *et al.* [16]). Essentially, communication within conversation contexts depends on the identity of the session associated with the context, and on the relative position of the context in a possibly nested context hierarchy. Communication has three modes: within a context, between a context and its parent context (the outer scope), and with the other endpoint of the session to which the context belongs. The last mode transcends arbitrary contextual boundaries. The identity of the current session can be dynamically accessed using a special prefix that returns a *self*-reference. Exceptions are raised with a `throw` primitive, and handled within the scope of a `try/catch` constructor. The behavioural semantics are checked by defining a strong bisimilarity on the label transition system upon which reduction is defined. In the follow-up work by Caires and Vieira [17], conversation types are considered for the aforementioned calculus. One point of interest is the flexibility in typing multi-participant interactions with an unconstrained, dynamic number of parties. Labels decorate actions within a conversation context, facilitating an analysis of progress based on the acyclicity of label occurrence in traces of actions.

Recently, Laneve and Padovani [54] provided an encoding between session types and contracts, based on a limited session type language without value passing which crucially excludes session delegation. They prove that under those assumptions the two methods can be used interchangeably, and observe that although the encoding of session types to contracts is almost direct, the other direction from contracts to session types causes an exponential growth of the generated type with respect to the size of the given contract. The authors attribute this result to the greater expressiveness of contracts, and indeed, it is intuitively justified considering the expansions needed in order to accommodate for the non-deterministic internal and external choices of contract languages.

Another work in the intersection of sessions and contracts is that of Boreale *et al.* [13]. The authors formalise the Calculus of Sessions and Pipelines (CaSPiS) for service oriented computing, utilising the context-based sessions used also in the above works. The novelty of this work is the introduction of a *pipelining* operator $P > Q$, which can be used to compose service invocations in a chain. Special outputs in P, called *return* messages, can be used to pass values to the context of the pipeline which can then receive the value, spawning a copy of the pipelined code Q to handle the invocation. Hence, in this calculus there are two forms of communication, one within a session, and one from a producer to a consumer in a pipeline of services. Pattern matching can be used to guide a protocol, operating on input values and channel names. The authors utilise a compositional termination handling mechanism, which allows complex hierarchies of nested sessions to terminate consistently; furthermore, the authors define a property called *gracefulness*, which holds when all sessions in a composition are equipped with a component that can handle early termination from either side.

Many of the above languages, specifically those using session contexts of the shape $r \triangleright P$ where r is the session identity and P communicates using an implicit intra-session communication mechanism, have evolved from the Service Centred Calculus (SCC) of Boreale *et al.* [12], developed within the collaborations in the SENSORIA Project [1].

Part II

Session Types and Subtyping in Higher-Order Processes and Objects

4 Sessions and Higher-Order Processes

Overview *Here we introduce session typing into a process language which models higher-order mobile code, i.e., one in which a process can be packaged (or "thunked") into a value sent to another process. Session types for the HOπ-calculus capture high-level structures of communication protocols and code mobility as type abstraction, and can be used to statically verify safe and consistent process composition in communication-centric distributed software. Integration of arbitrary higher-order code mobility and sessions leads to technical difficulties in type soundness, because both linear usage of session channels and the completion of sessions are required in executed code. By using techniques from the linear λ-calculus, we develop a coherent and tractable session typing system for the HOπ-calculus.*

4.1 Introduction

In global computing environments, applications are executed across multiple distributed sites or devices. The use of mobile code is prominent in such environments, where several participants are synthesised by communication of not only passive values but also of runnable code: for example a service can be delegated to different participants, by sending either a channel via which it is accessible, or code that accesses it; and incoming code may transit through several devices that alter their computational behaviour or their data through interaction with it.

The Higher-Order π-calculus (HOπ-calculus) [79] is a general formalism of interaction in which two kinds of mobility, name passing and process passing, are integrated in a simple and universal form: in this model, processes can be instantiated by names and other processes, just like a piece of mobile code is instantiated with local capability after migration. This additional expressiveness inherited from the λ-calculus provides a powerful basis for describing and analysing dynamic behaviour in global computing scenarios.

While many advanced session types for the π-calculus and programming languages have been studied, before our work [63] there existed no session typing systems for the HOπ-calculus. Incorporation of sessions into this language offers a general theoretical basis for examining the interplay between two non-trivial features in communication-based programming, higher-order mobility and session-based structured interaction.

This Chapter, based on [63], establishes the first session type theory for the HOπ-calculus which can statically validate the type safety of complex distributed scenarios with code mobility. In spite of their simple type syntax, the previous literature have shown that obtaining type soundness for session types is an intricate task because of delegation of sessions [92]. In addition, in the presence of higher-order process passing, with the instantiation of names within executable code, preservation of typability becomes even more non-trivial.

4.2 The Higher-Order π-Calculus with Sessions

The Higher-order π-calculus with sessions, HOπ^{s}, is a variant of the HO π-calculus [79]. The main difference is that in HOπ^{s} each communication occurs not freely, but in the context of an initiated session synchronising two processes to perform a prescribed protocol. HOπ^{s} encompasses two types of mobility: name passing, with which dynamic communication topologies can be programmed, and code passing, where by transmitting processes a dynamic behaviour can be achieved. Note that the calculus is monadic, i.e., only one value is sent/received at each communication step, but this does not affect the results and serves for simplicity.

4.2.1 Syntax

The syntax of HOπ^{s} is given in Figure 4.1. The calculus extends the HOπ with a small kernel of session primitives: a way to initiate a session over a shared channel, a class of session names — which we call *endpoints* — used for communications within sessions, and primitives for offering and making choices indexed by labels.

Identifiers Variables range over $x,y,z,\ldots$. *Shared* channel names, which are used only to initiate sessions (we describe this in detail further below), are ranged over $a,b,c,\ldots$. We write $u,v,w,\ldots$ to represent shared identifiers, that is, those that are either variables or shared channel names. *Session* channels, ranged over $s,\ldots$ and $\bar{s},\ldots$, are the *endpoints* through which values are communicated *within* an established session (which as we shall see is always between exactly two processes). The name $\bar{s}$ denotes the *dual* of s, that is, if one process in a session uses s, the other process uses $\bar{s}$, and in this way each of the two processes possess a unique endpoint. This separation of endpoints is similar to the use of two *polarities* in [39, 92]. We define duality to be idempotent,

Identifiers

u, v, w	$::=$	x, y, z	variables
	$\mid$	a, b, c	shared channels
k	$::=$	x, y, z	variables
	$\mid$	$s, \overline{s}$	session channels

Values

V, V', W	$::=$	u, v, w	shared identifier
	$\mid$	k, k', k''	linear identifier
	$\mid$	$()$	unit
	$\mid$	$\lambda(x : U).P$	abstraction
	$\mid$	$\mu(x{:}U \to T).\lambda(y{:}U).P$	recursion

Terms

P, Q, R	$::=$	V	value
	$\mid$	$u(x).P$	connect
	$\mid$	$\overline{u}(x).P$	connect dual
	$\mid$	$k?(x).P$	input
	$\mid$	$k!\langle V \rangle.P$	output
	$\mid$	$k \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$	branching
	$\mid$	$k \lhd l.P$	selection
	$\mid$	$P \mid Q$	parallel
	$\mid$	$(\nu a : \langle S \rangle)\, P$	restriction
	$\mid$	$(\nu s)\, P$	restriction
	$\mid$	PQ	application
	$\mid$	$\mathbf{0}$	nil process

Abbreviations

$\ulcorner P \urcorner$	$\stackrel{\text{def}}{=}$	$\lambda(x{:}\mathtt{unit}).P \quad (x \notin \mathsf{fv}(P))$	thunk
run	$\stackrel{\text{def}}{=}$	$\lambda x.(x\,())$	run

Figure 4.1: Syntax

thus, we have that $\overline{\overline{s}} = s$. This property of endpoint names is used in the reduction semantics, where a communication is synchronised over the two endpoints of a session. We write $k, k', k'', \ldots$ for linear identifiers, consisting of variables and session channels.

Values We write $V, V', W, \ldots$ for those terms that may be used as values, that is, as the object of a communication or as the argument in function application. First, we have identifiers, shared and linear (as standard). Abstraction, written $\lambda(x : U).P$, encapsulates a process P, where x may occur free, into a function over x (with type annotation U). This is the basic mechanism for the exchange of processes, and the unit () is useful when we wish to obtain a value from an arbitrary process P: take a variable x not free in P, then $\lambda(x\!:\!\mathsf{unit}).P$ is a value, usually referred to as a *thunk*, and abbreviated to $\ulcorner P \urcorner$. To reveal and execute the process within a thunk, we use the *run* function $\lambda(x\!:\!\mathsf{unit} \to \diamond).(x\,())$ which takes a thunk as argument and applies it to the unit value to obtain the hidden process.

To facilitate terms that exhibit infinitary behaviour, we introduce a recursive function constructor $\mu(x\!:\!U \to T).\lambda(y\!:\!U).P$. In this fixpoint representation, instances of the variable x within P represent the function itself.

Terms The terms of the calculus are written $P, Q, R, \ldots$. The main constructs are:

Session initialisation $u(x).P$ and $\overline{u}(x).Q$ are prefixed processes that may synchronise and commence a session. The interactions will adhere to the session type assigned to the shared identifier u, and since each session consists of two endpoints used in a complementary way, we distinguish the two different behaviours with respect to this type using u and $\bar{u}$. The bound variable x is a placeholder for a fresh session endpoint, initialised after the prefixes react to establish a session.

Input and Output $k?(x).P$ is the standard input prefixed process, with linear subject k and using x as a placeholder for the received value. $k!\langle V \rangle.P$ is an output prefixed process, sending value V over session k.

Branching and Selection $k \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$ offers a set of label-indexed choices $l_i : P_i$ on endpoint k, with a process continuation P_i corresponding to each label l_i. It is often written $k \rhd \{l_i : P_i\}_{i \in I}$ with index set I. The dual (or co-action) of a branch is a process ready to perform a selection $k \lhd l.P$ where the chosen label is within the domain of the branch set. Essentially a branching is an input expecting a label and performing case analysis (which covers all cases) on this label to choose a continuation. Dually, a selection is an output of a label designating a choice. Clearly, it is undesirable to allow the empty set in branching, since no selection can be made (that is, there is no effective co-action), and henceforth we assume that there is at least one branch (and the respective indexing sets, when used, are non-empty).

Fresh names We write $(\nu a : \langle S \rangle)\, P$ to denote a process P in which the shared channel a (typed by $\langle S \rangle$) is unique. With $(\nu s)\, P$ we denote that the two endpoints s and $\overline{s}$ are unique in P, that is, no external process can perform a session action on either of these endpoints; this gives non-interference within a session.

Other constructs are the nil process $\mathbf{0}$, parallel composition $P \,|\, Q$, and functional application PQ, which are standard from π-calculus and λ-calculus. We often omit $\mathbf{0}$ and some type annotations when not relevant.

The *bindings* are induced by $(\nu a : \langle S \rangle)P$, $(\nu s)P$, $u(x).P$, $\overline{u}(x).P$, $k?(x).P$, $\lambda(x : U).P$, and $\mu(x : U \to T).\lambda(y : U).P$. The derived notions of bound and free identifiers, alpha equivalence and substitution are mostly standard. We write $\mathsf{fv}(P)$/$\mathsf{fn}(P)$ for the set of free variables/names, respectively; the definition is in Figure 4.2. Moreover, when dealing with proofs we assume the variable convention, that is, free and bound variables are always chosen to be different, and all bound variables are distinct; the same applies to names.

4.2.2 Reduction Semantics

We define the standard structural congruence, denoted '$\equiv$', as the smallest equivalence relation which is congruent with respect to the calculus constructors (parallel composition, name restriction, prefixes) and respects the axioms and rules in Figure 4.3. The single-step call-by-value reduction relation, denoted $\longrightarrow$, is a binary relation from closed terms to closed terms, defined by the rules in Figure 4.4. Rule (beta) is standard from the call-by-value λ-calculus. The case of (rec) is similar, with the added step of unfolding the recursive function, by substituting it in place of the variable y within the function body P.

Rule (conn) establishes a new session between two processes $a(x).P$ and $\overline{a}(z).Q$ ready to synchronise on a. The result of this rewriting is a parallel composition of the session bodies P and Q with a fresh set of endpoints s and $\overline{s}$ substituted for the session variables x and z, respectively. The sidecondition ensures that the new endpoints do not already appear free in either P or Q.

Rule (comm) realises session communication between endpoints k and $\overline{k}$: a value is delivered

Term	fv	fn
x	$\{x\}$	$\emptyset$
a	$\emptyset$	$\{a\}$
s	$\emptyset$	$\{s\}$
$\overline{s}$	$\emptyset$	$\{\overline{s}\}$
$()$	$\emptyset$	$\emptyset$
$\lambda x.P$	$\mathsf{fv}(P) \setminus \{x\}$	$\mathsf{fn}(P)$
$\mu x.\lambda y.P$	$\mathsf{fv}(P) \setminus \{x,y\}$	$\mathsf{fn}(P)$
$u(x).P \,/\, \overline{u}(x).P$	$\mathsf{fv}(u) \cup (\mathsf{fv}(P) \setminus \{x\})$	$\mathsf{fn}(u) \cup \mathsf{fn}(P)$
$k?(x).P$	$\mathsf{fv}(k) \cup (\mathsf{fv}(P) \setminus \{x\})$	$\mathsf{fn}(k) \cup \mathsf{fn}(P)$
$k!\langle V \rangle.P$	$\mathsf{fv}(k) \cup \mathsf{fv}(V) \cup \mathsf{fv}(P)$	$\mathsf{fn}(k) \cup \mathsf{fn}(V) \cup \mathsf{fn}(P)$
$k \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$	$\mathsf{fv}(k) \cup \mathsf{fv}(P_1) \cup \ldots \cup \mathsf{fv}(P_n)$	$\mathsf{fn}(k) \cup \mathsf{fn}(P_1) \cup \ldots \cup \mathsf{fn}(P_n)$
$k \lhd l.P$	$\mathsf{fv}(k) \cup \mathsf{fv}(P)$	$\mathsf{fn}(k) \cup \mathsf{fn}(P)$
$P \,\vert\, Q \,/\, PQ$	$\mathsf{fv}(P) \cup \mathsf{fv}(Q)$	$\mathsf{fn}(P) \cup \mathsf{fn}(Q)$
$(\nu a : \langle S \rangle)\, P$	$\mathsf{fv}(P)$	$\mathsf{fn}(P) \setminus \{a\}$
$(\nu s)\, P$	$\mathsf{fv}(P)$	$\mathsf{fn}(P) \setminus \{s, \overline{s}\}$
$\mathbf{0}$	$\emptyset$	$\emptyset$

Figure 4.2: Free Variables and Free Names

$P =_\alpha Q \Rightarrow P \equiv Q$	Renaming of bound variables
$P \,\vert\, Q \equiv Q \,\vert\, P$	Commutativity of parallel composition
$(P \,\vert\, Q) \,\vert\, R \equiv P \,\vert\, (Q \,\vert\, R)$	Associativity of parallel composition
$P \,\vert\, \mathbf{0} \equiv P$	Inaction and parallel composition
$(\nu a : \langle S \rangle)\, P \,\vert\, Q \equiv (\nu a : \langle S \rangle)\, (P \,\vert\, Q) \quad a \notin \mathsf{fn}(Q)$	Scope extrusion
$(\nu s)\, P \,\vert\, Q \equiv (\nu s)\, (P \,\vert\, Q) \quad s, \overline{s} \notin \mathsf{fn}(Q)$	
$(\nu a : \langle S \rangle)\, (\nu s)\, P \equiv (\nu s)\, (\nu a : \langle S \rangle)\, P$	Exchange
$(\nu a : \langle S \rangle)\, (\nu b : \langle S' \rangle)\, P \equiv (\nu b : \langle S' \rangle)\, (\nu a : \langle S \rangle)\, P$	
$(\nu s)\, (\nu s')\, P \equiv (\nu s')\, (\nu s)\, P$	
$(\nu a : \langle S \rangle)\, \mathbf{0} \equiv \mathbf{0} \qquad (\nu s)\, \mathbf{0} \equiv \mathbf{0}$	Inaction and restriction

Figure 4.3: Structural Congruence

(beta)
$$(\lambda(x:U).P)V \longrightarrow P\{V/x\}$$

(rec)
$$(\mu y.\lambda x.P)V \longrightarrow P\{V/x\}\{\mu y.\lambda x.P/y\}$$

(conn)
$$a(x).P \mid \overline{a}(z).Q \longrightarrow (\nu s)\,(P\{s/x\} \mid Q\{\overline{s}/z\}) \quad s,\overline{s} \notin \mathsf{fn}(P,Q)$$

(comm)
$$k?(x).P \mid \overline{k}!\langle V\rangle.Q \longrightarrow P\{V/x\} \mid Q \qquad k = s \text{ or } k = \overline{s}$$

(label)
$$k \rhd \{l_1:P_1,\dots,l_n:P_n\} \mid \overline{k} \lhd l_m.P \longrightarrow P_m \mid P \qquad k = s \text{ or } k = \overline{s},\ 1 \le m \le n$$

$$\text{(app-l)}\ \frac{P \longrightarrow P'}{PQ \longrightarrow P'Q} \qquad \text{(app-r)}\ \frac{Q \longrightarrow Q'}{VQ \longrightarrow VQ'} \qquad \text{(par)}\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$\text{(resc)}\ \frac{P \longrightarrow P'}{(\nu a:\langle S\rangle)P \longrightarrow (\nu a:\langle S\rangle)P'} \qquad \text{(ress)}\ \frac{P \longrightarrow P'}{(\nu s)P \longrightarrow (\nu s)P'} \qquad \text{(str)}\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Figure 4.4: Reduction

from $\overline{k}!\langle V\rangle.Q$ to $k?(x).P$. Due to the self-inverse duality property of endpoints, if $k = s$ then we have an output from $\overline{s}$ to s, and if $k = \overline{s}$, the output is from s to $\overline{s}$. The result is the input process continuation $P\{V/x\}$ where the value V is substituted for the variable x, in parallel to the output continuation Q. When V is a function, we have *higher-order code passing*; when V is a session endpoint, we call it *higher-order session passing*.

Rule (label) is a communication version of case reduction in the λ-calculus. With $k \rhd \{l_1 : P_1,\dots,l_n:P_n\}$, the session on k offers a set of label-indexed choices, and reduces against a selection of a label on the dual endpoint $\overline{k}$, using $\overline{k} \lhd l_m.P$. The index m of the selected label l_m must be in the branch set as indicated by the sidecondition. Finally, the result is the selected branch P_m in parallel with the continuation P of the selector process.

In the remaining rules: (app-l) and (app-r) implement a left to right reduction order for functional application; (par) reduces the leftmost parallel process; (resc) and (ress) are standard and reduce a process under name hiding. The last rule, (str), introduces standard structural congruence [60] into the reduction relation. This is necessary for re-arranging terms to match reduction rules.

Remark In our original work [63], processes of the shape $a(x).P$ (called "servers") were replicated, but this is not needed here as we have introduced recursive functions which can encode replication.

4.2.3 Examples

Example 4.2.1 (Encoding Replication)**.** By using recursion, we can represent infinite behaviours of processes such as, e.g., the definition agent def, or the replication $!u(x).P$ of [59, 92, 48, 63]. Replication on a shared name, useful for defining persistent servers, can be encoded as follows:

$$!u(x).P \quad \stackrel{\text{def}}{=} \quad (\mu y.\lambda z.z(x).(P \mid y\,z))\,u \qquad \text{taking} \quad y,z \notin \mathsf{fv}(P)$$

Hereafter when writing a replicated connection-prefixed process we shall mean that this encoding is used. Note that we did not (and by typing we cannot) replicate a session endpoint, since that would violate linearity. To validate the encoding, we can observe a reduction using a replicated connection $!a(x).P$ and a suitable co-action $\overline{a}(z).Q$:

$$\begin{array}{cll}
 & !a(x).P \mid \overline{a}(z).Q & \\
\longrightarrow & a(x).(P \mid !a(x).P) \mid \overline{a}(z).Q & (\mathsf{rec}) \\
\longrightarrow & (\nu s)\,(P\{^{s}\!/\!_{x}\} \mid !a(x).P \mid Q\{^{\overline{s}}\!/\!_{z}\}) & (\mathsf{conn}) \\
\equiv & (\nu s)\,(P\{^{s}\!/\!_{x}\} \mid Q\{^{\overline{s}}\!/\!_{z}\}) \mid !a(x).P &
\end{array}$$

Note that in the application of rule (conn), since x is bound in $!a(x).P$, the substitution $\{^{s}\!/\!_{x}\}$ has no effect on this subterm. Once a connection is established via (conn), we can apply structural congruence $\equiv$ to obtain a term where $!a(x).P$ can react again; for this we used the fact that s and $\overline{s}$ do not occur free in $!a(x).P$, which is ensured by the conditions of the previous reduction with (conn).

4.2.4 Example: Business Protocol with Code Mobility

We show a simple protocol which contains essential features by which we can demonstrate the expressivity of the code mobility and session primitives for the HOπ-calculus; it consists of a combination of session establishing, code mobility, session delegation and branching. This extends a typical collaboration pattern that appears in many web service business protocols [90, 19] to code mobility. In Figure 4.5, we show the sequence diagram for a protocol which models a hotel booking: first, `Booking Agency` and `Client` initiate interaction at session x over channel a; then `Client` starts exchanging a series of information with `Agency`; during this initial communication, `Agency` calculates its Round Trip Time (RTT) between `Client` and `Agency`; `Agency` selects an appropriate `Hotel` and creates a new session y over channel b with that `Hotel`. If the RTT is short (Figure 4.5 (a)), then `Agency` delegates to `Client` its part of the remaining activity with `Hotel`, by

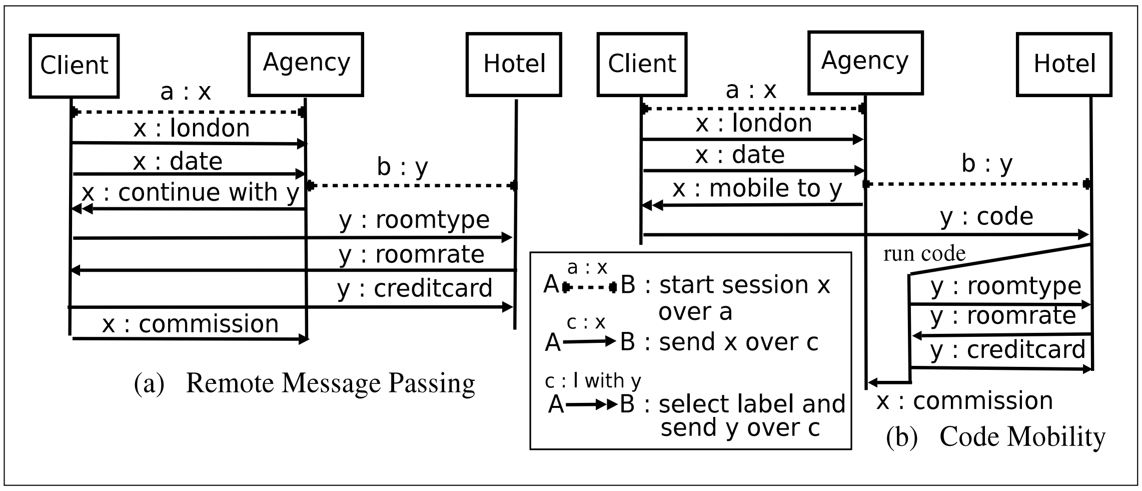

Figure 4.5: Sequence Diagram for Hotel Booking

sending session channel *y*; then `Client` and `Hotel` continue negotiations by message passing. If the RTT is long (Figure 4.5 (b)), since many remote interactions increase the communication time as well as the danger of communication failures, `Agency` asks back `Client` to *send mobile code* which contains the communication of the `Client`'s room plan and negotiation behaviour. `Agency` sends the code to `Hotel`, then `Hotel` runs it locally, finishing a series of interactions in its location. Finally `Agency` receives a commission fee (10 percent of the room rate) via session *x*, concluding the transaction.

The given scenario is straightforwardly encoded in our calculus, where session primitives make the structure of interactions clearer. `Agency` first initiates at *a* and starts the interactions with `Client`; then it initiates at *b* and establishes session *y*; next it invokes either label cont or label move in `Client` depending on the RTT and sends *y* (higher-order session passing) to it, and waits for completion of the transaction between `Client` and `Hotel` at *x* ("if-then-else" can be encoded using branching, and we use other base types and their operators).

$$\texttt{Agency} \stackrel{\text{def}}{=} !a(x).x?(area)\ldots\overline{b}(y).\texttt{if}\ \text{rtt} < 100 \quad \texttt{then}\ x \triangleleft \mathsf{cont}.x!\langle y\rangle.x?(z)..P \quad (4.1)$$

$$\texttt{else}\ x \triangleleft \mathsf{move}.x!\langle y\rangle.x?(z).P \quad (4.2)$$

`Client` requests a service at *a* and starts a series of interactions with `Agency`, and either continues the remaining activity with `Hotel` or sends the code (a thunk in Line 4.4). Note that `Client` can safely send back the commission fee to `Agency` because the return message $\overline{x}\langle z \times 0.1\rangle$ which uses session channel *x* is embedded in the thunk.

$$\texttt{Client} \overset{\text{def}}{=}$$

$$\overline{a}(x).x!\langle \mathit{london}\rangle..x \rhd \{ \quad \mathsf{cont} : x?(y).y \lhd \mathsf{cont}.y!\langle \mathit{roomtype}\rangle.y?(z)...x!\langle z \times 0.1\rangle \;, \tag{4.3}$$

$$\mathsf{move} : x?(y).y \lhd \mathsf{move}.y!\langle \ulcorner y!\langle \mathit{roomtype}\rangle.y?(z)...x!\langle z \times 0.1\rangle \urcorner\rangle \} \tag{4.4}$$

`Hotel` performs the interactions with `Agency` and `Client` via a single session at y (by the facility of higher-order session). In Line 4.6, the code sent by `Client` is run locally.

$$\texttt{Hotel} \overset{\text{def}}{=} \;!b(y).y \rhd \{ \quad \mathsf{cont} : y?(z).y!\langle \mathrm{roomrate}.y?(z)\rangle...Q \;; \tag{4.5}$$

$$\mathsf{move} : y?(\mathit{code}).(\mathit{run}\ \mathit{code} \mid y?(z).y!\langle \mathsf{roomrate}(z)\rangle...Q)\} \tag{4.6}$$

This encoding shows a couple of subtle points whose slight modification breaks the session structures. First, in Line 4.4, if we send code which does not complete the session, then the protocol is broken: e.g. if we have interactions at y (say $y!\langle w\rangle$) after sending a thunk in Line 4.4 in `Client`, the session at y will appear in three threads (two in `Hotel`, one in `Client`), so the session at y is interfered with and values may get mixed up. Secondly, in Line 4.6, if we have two or more applications (say $\mathit{run}\ \mathit{code} \mid \mathit{run}\ \mathit{code}$) instead of one $\mathit{run}\ \mathit{code}$, it again breaks the session structure (both at y and x). Finally, if the code is not activated in Line 4.6 (like $(\lambda x.\mathbf{0})\mathit{code}$ instead of $\mathit{run}\ \mathit{code}$), the receiver $y(z)\,.\,y!\langle \mathsf{roomrate}(z)\rangle...Q$ cannot find a matching output. Hence the variable *code* must appear exactly once and become instantiated into a process exactly once.

4.3 Higher-Order Linear Typing

In this section we present the session typing system which uses techniques from linear typing.

4.3.1 Types

The syntax of types is given on Figure 4.6. It is an integration of the types from the simply typed λ-calculus with unit and the session types from the π-calculus. Term types range over T, and can be value types, ranging over U, or the process type $\diamond$. Value types consist the unit type unit, the type $U \to T$ of shared functions, the type $U \multimap T$ of linear functions, the type S of sessions, and the shared channel type $\langle S\rangle$ which enforces that sessions initiated on the corresponding channel will follow the protocol defined by S.

The session types are defined inductively as follows. The type $![U].S$ represents the sending of a value of type U, followed by the remaining session S. Dually, with $?[U].S$ the action will

Term

T	$::=$	U	value
	$\mid$	$\diamond$	process

Value

U	$::=$	unit	unit
	$\mid$	$U \rightarrow T$	shared function
	$\mid$	$U \multimap T$	linear function
	$\mid$	$\langle S \rangle$	shared channel
	$\mid$	S	session

Session

S	$::=$	$![U].S$	output
	$\mid$	$?[U].S$	input
	$\mid$	$\oplus[l_1 : S_1, \ldots, l_n : S_n]$	selection
	$\mid$	$\&[l_1 : S_1, \ldots, l_n : S_n]$	branching
	$\mid$	$\mathbf{t}$	type variable
	$\mid$	$\mu\mathbf{t}.S$	recursion
	$\mid$	end	ending

Figure 4.6: Types

be to receive a value of expected type at least U, followed by S as before. The selection type $\oplus[l_1 : S_1, \ldots, l_n : S_n]$ signifies that one of the choices $l_1, \ldots, l_n$ will be made (operationally this is an output of a label), and depending on this label the corresponding session continuation chosen from $S_1, \ldots, S_n$ will take place. The co-type of selection is the branch type $\&[l_1 : S_1, \ldots, l_n : S_n]$ corresponding to the reception of a label followed by the corresponding continuation type as in selection. Recursive session types are written $\mu\mathbf{t}.S$, where the type variable $\mathbf{t}$ is bound and may occur free in S. We only consider contractive recursive types [35, 92]. Practically, contractiveness of $\mu\mathbf{t}.S$ means that every free instance of $\mathbf{t}$ in S is *guarded* under at least one input, output, selection or branching constructor. For example $\mu\mathbf{t}.![\mathsf{nat}].\mathbf{t}$ is contractive, but $\mu\mathbf{t}.\mu\mathbf{t}'.\mathbf{t}$ is not. Moreover, we only consider *tail-recursive* session types, therefore types such as $\mu\mathbf{t}.![\mathbf{t}].\mathsf{end}$ are not well-formed. To indicate that a session is finished, we use the terminal end.

We write $\mathcal{T}$ for the set of types.

Abbreviated Forms We often write $\&[l_i : S_i]_{i \in I}$ and $\oplus[l_i : S_i]_{i \in I}$ for branching and selection types, $\ulcorner T \urcorner$ for $\mathsf{unit} \to T$ and $\ulcorner T \urcorner^1$ for $\mathsf{unit} \multimap T$. The terminal end is sometimes omitted.

Example 4.3.1 (Types). Session types can encode many common interactions. For example the following type can be used to iterate through a list containing elements of type U:

$$\mu\mathbf{t}. \oplus [\mathsf{hasnext} : \&[\mathsf{next} : ?[U].\mathbf{t}, \mathsf{finished} : \mathsf{end}], \mathsf{finished} : \mathsf{end}]$$

The type describes the behaviour of the client process accessing the list: first a choice is made, either to query the list and discover if it has more elements, by choosing hasnext; or alternatively the choice finished can be made in which case the protocol reaches its end. If hasnext is chosen, then the list can respond by choosing next, after which the client can receive a value of type U. Moreover the type variable $\mathbf{t}$ signifies that at this point the protocol is repeated from the point of definition, that is, from the μ-binder at the beginning. If the list replies by choosing finished, the protocol is complete.

Duality In the above example (4.3.1) we show the type of the iterator, but not of the list. In fact the list's type can be obtained by *duality*. Each session type S has a *dual* type, denoted by $\overline{S}$, which describes *complementary* behaviour. This is inductively defined by the rules in Figure 4.7. Essentially, dualisation interchanges input (?) with output (!), branching (&) with selection ($\oplus$), leaving end, type variables and μ binders unchanged. Duality is idempotent. Note that we do not need to define duality for other types such as the function types, as these are never dualised.

$$\overline{![U].S} = ?[U].\overline{S} \qquad \overline{?[U].S} = ![U].\overline{S} \qquad \overline{\mathbf{t}} = \mathbf{t} \qquad \overline{\mu\mathbf{t}.S} = \mu\mathbf{t}.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

$$\overline{\oplus[l_1 : S_1, \ldots, l_n : S_n]} = \&[l_1 : \overline{S_1}, ..., l_n : \overline{S_n}] \qquad \overline{\&[l_1 : S_1, \ldots, l_n : S_n]} = \oplus[l_1 : \overline{S_1}, ..., l_n : \overline{S_n}]$$

Figure 4.7: Type Duality

4.3.2 Subtyping

To formalise subtyping in the presence of recursive types a *simulation*-based (or *coinductive*) method is used, in which subtyping is determined by membership of the goal within a binary relation on types.

First, let us define:

$$(S, S')^{\circledast} = (S', S)$$
$$(T, T')^{\circledast} = (T, T') \qquad \text{if } T, T' \text{ are not session types}$$

which is used to adjust for the different variance of functional and session types, by reversing the variance of sessions with respect to the other types. A technically equivalent approach where the variance is the same for sessions and functions can be applied (see e.g. [39]) but we find that our approach is more natural for our effect-like typing system and for understanding subtyping in sessions programming in a way reminiscent of record subtyping.

In the following definition of coinductive subtyping we adapt standard simulation approaches from [39, 77].

Definition 4.3.1 (Coinductive Subtyping). A relation $\mathfrak{R} \in \mathcal{T} \times \mathcal{T}$ is a type simulation if $(T_1, T_2) \in \mathfrak{R}$ implies that at least one of the following conditions must hold:

1. If $T_1 = \diamond$, then $T_2 = \diamond$.

2. If $T_1 = \mathsf{unit}$, then $T_2 = \mathsf{unit}$.

3. If $T_1 = U_1 \rightarrow T_1'$, then $T_2 = U_2 \rightarrow T_2'$ or $T_2 = U_2 \multimap T_2'$ with $(U_2, U_1)^{\circledast} \in \mathfrak{R}$ and $(T_1', T_2')^{\circledast} \in \mathfrak{R}$.

4. If $T_1 = U_1 \multimap T_1'$, then $T_2 = U_2 \multimap T_2'$ with $(U_2, U_1)^{\circledast} \in \mathfrak{R}$ and $(T_1', T_2')^{\circledast} \in \mathfrak{R}$.

5. If $T_1 = \langle S_1 \rangle$, then $T_2 = \langle S_2 \rangle$ and $(S_1, S_2) \in \mathfrak{R}$ and $(S_2, S_1) \in \mathfrak{R}$.

6. If $T_1 = \mathsf{end}$, then $T_2 = \mathsf{end}$.

7. If $T_1 = ![U_1].S_1$, then $T_2 = ![U_2].S_2$, $(U_1,U_2)^{\circledast} \in \mathfrak{R}$ and $(S_1,S_2) \in \mathfrak{R}$.
8. If $T_1 = ?[U_1].S_1$, then $T_2 = ?[U_2].S_2$, $(U_2,U_1)^{\circledast} \in \mathfrak{R}$ and $(S_1,S_2) \in \mathfrak{R}$.
9. If $T_1 = \oplus[l_i : S_{1i}]_{i\in I}$, then $T_2 = \oplus[l_j : S_{2j}]_{j\in J}$, $I \subseteq J$ and $\forall i \in I.(S_{1i},S_{2i}) \in \mathfrak{R}$.
10. If $T_1 = \&[l_i : S_{1i}]_{i\in I}$, then $T_2 = \&[l_j : S_{2j}]_{j\in J}$, $J \subseteq I$ and $\forall j \in J.(S_{1j},S_{2j}) \in \mathfrak{R}$.
11. If $T_1 = \mu\mathbf{t}.S$, then $(S[\mu\mathbf{t}.S/\mathbf{t}],T_2) \in \mathfrak{R}$.
12. If $T_2 = \mu\mathbf{t}.S$, then $(T_1,S[\mu\mathbf{t}.S/\mathbf{t}]) \in \mathfrak{R}$.

The integration of subtyping for higher-order (linear) functions and asynchronous sessions requires a careful formulation: (1,2,6) are standard identity rules. (3) says that an unlimited function can be used as a linear function. Note that the reverse is unsafe: suppose $f = \lambda x.k!\langle x\rangle$ with a linear type $\mathsf{nat} \multimap \diamond$. If we apply the reverse direction, $\lambda(y{:}\,\mathsf{nat} \to \diamond).(y\,1 \mid y\,2)f$ becomes typable, destroying the linearity of session k.

Also in (3), when U_i is a session type, we use the relation $(S_1,S_2)^{\circledast} = (S_2,S_1)$ to transpose the tuple. Session types are dualised since the session channel is going to be used in a process in a contravariant manner. To see this condition, suppose process $P = (\lambda(x{:}\,S).x?(y).x!\langle 2\rangle.\mathbf{0})\,s$ with $S = ?[\mathsf{real}].![\mathsf{nat}].\mathsf{end}$. Then P can safely interact with $Q = \overline{s}!\langle 5\rangle.\overline{s}?(z).\mathbf{0}$ where the type of $\overline{s}$ is $S' = ![\mathsf{nat}].?[\mathsf{real}].\mathsf{end}$. The types S and S' are not dual, but each subsumes a type which is dual to the other. For example, taking $\mathsf{nat} \leqslant_c \mathsf{real}$, we can obtain $S \leqslant_c ?[\mathsf{nat}].![\mathsf{real}].\mathsf{end} = \overline{S'}$. This subtyping is intuitive when we understand S as the *actual* behaviour of variable x (and consequently of s) in process P, and all supertypes as the behaviours that P also satisfies. In this example, P can safely substitute a process that is expected to use s according to $?[\mathsf{nat}].![\mathsf{real}].\mathsf{end}$: it can receive any value of type nat (since it can receive any real) and will send a value that inhabits the type real (since any nat is also a real). For P to compose with Q we must have $S \to \diamond \leqslant_c \overline{S'} \to \diamond$, with $S \leqslant_c \overline{S'}$, that is, the subtype ordering of session types left of $\to$ is covariant. The contravariance of session types on the right of $\to$ can be justified in the same way. The case when T_i is a session type is also similarly explained. (4) is similar.

Remark The original session typing system uses a judgement "$\Gamma \vdash P : \Sigma$" where Γ is a shared (standard) environment and Σ is a mapping from a session channel to a session type. This means: P accesses the session channels specified at most by Σ. In contrast, in our typing system defined in the next section, Σ appears in the left-side position, so that we need to dualise the session types for subtyping, cf. [91].

(5) says that the shared channel type is invariant (as in the standard session types [39, 65, 48]). (7-8) match the output/input prefixes and check the continuations, as expected. The cases (9-10) for selection and branching subsume the traditional session branching/selection subtyping. The last two rules ensure that types are *unfolded* (or *unrolled*) adequately for the other rules to be applicable. We use the notation $S[\mu\mathbf{t}.S/\mathbf{t}]$ to mean that all free occurrences of $\mathbf{t}$ in S are replaced by the original type $\mu\mathbf{t}.S$. The notion of free type variables is simple to define: the only binder is μ, that is, $\mathbf{t}$ is bound in $\mu\mathbf{t}.S$, and all instances of type variables that are not bound (i.e., do not appear under a μ binder) within a type are free instances. We omit the formal definition which is trivial.

Our relations contain slightly more elements compared to those of [39] — where unfolding is done within the other rules — because in our definition all the pairs arising from unfolding are included in the type simulation. But this is not a significant difference because by the contractiveness restriction on types we know that unfolding of a μ-prefixed type does not generate an infinite relation.

As standard, the coinductive subtyping relation $\leqslant_c$ is the union of all type simulations and is defined, for types T_1 and T_2, when there exists a type simulation $\Re$ with $(T_1, T_2) \in \Re$. When the actual relation is not important we write $T_1 \leqslant_c T_2$.

4.3.3 Linear Higher-Order Typing System

Environments

We first define three kinds of finite mappings for environments, needed when typing a term with free identifiers:

$$\begin{array}{rrcl}
\text{(Shared)} & \Gamma & ::= & \emptyset \mid \Gamma, u : \mathsf{unit} \mid \Gamma, u : U \rightarrow T \mid \Gamma, u : \langle S \rangle \\
\text{(Linear)} & \Lambda & ::= & \emptyset \mid \Lambda, x : U \multimap T \\
\text{(Session)} & \Sigma & ::= & \emptyset \mid \Sigma, k : S
\end{array}$$

Γ is a finite mapping, associating *shared* value types to identifiers. Λ associates variables and *linear* function types. There is no need to define mappings here for identifiers such as channels or session endpoints as these cannot have a linear function type. Σ is a finite mapping from variables/session channels to session types. Σ, Σ' and Λ, Λ' denote disjoint-domain unions. $\Gamma, u\!:\!U$ means $u \notin \mathsf{dom}(\Gamma)$, and similarly for the other environments.

Typing Judgement

The typing judgement takes the shape:

$$\Gamma; \Lambda; \Sigma \vdash P : T$$

which is read: under a (global) shared environment Γ and a linear function environment Λ, a term P has type T with session usages described by Σ. We say that a judgement is *well-formed* if the environments (pairwise) do not share elements in their domains, that is, when the disjoint union $\mathsf{dom}(\Gamma) \uplus \mathsf{dom}(\Lambda) \uplus \mathsf{dom}(\Sigma)$ is defined.

Typing Rules

The typing rules for identifiers, subtyping, and functions are given in Figure 4.8. The rules for processes and sessions are given in Figure 4.9. In each rule, we assume that the environments in the consequence are defined.

Starting from Figure 4.8, the first group is **(Common)**. First we have a rule for the unit value (), assigning the type unit. In the conclusion, notice that an arbitrary Γ is allowed, but no recording of linear variables ($\Lambda = \emptyset$), or sessions ($\Sigma = \emptyset$). This restriction agrees with the use of weakening only for shared environments, a condition necessary for the preservation of linearity. (Shared) is an introduction rule for identifiers with shared types, i.e., not including $U \multimap T$ or S. (LVar) is for linear variables and (Session) is for session endpoints, recording $x : U \multimap T$ in Λ and $k : S$ in Σ, respectively. The general strategy is that the environments Λ and Σ record precisely the desired usages of linear variables/sessions, and then within a derivation these usages are combined using disjoint union (to ensure that no copying takes place) and prefixing composition in the case of sessions (to ensure that certain separated usages are seen as one largest use). The use of disjoint union effectively forbids contraction. The absence of weakening guarantees that all linear hypotheses are actually used.

The group **(Subtyping)** consists of one subsumption rule, (Sub), introducing the coinductive subtyping $\leqslant_c$ into typing derivations. For example this rule can lift from the shared function $U \to T$ to the linear function $U \multimap T$. The other direction would be unsafe as it would allow copying of a linear function by first promoting it to a shared type and then using it as such. We write $\Sigma \leqslant_c \Sigma'$ when $\mathsf{dom}(\Sigma) = \mathsf{dom}(\Sigma')$ and for all $k : S \in \Sigma$, we have $k : S' \in \Sigma'$ with $S \leqslant_c S'$. Notice that subsumption can apply to the session environment, but not to other environments, and it can also apply to the given type T for the term P. The reason for which $\leqslant_c$ should be applicable to

(Common)

(Unit) $\dfrac{}{\Gamma;\emptyset;\emptyset \vdash () : \mathsf{unit}}$ (Shared) $\dfrac{}{\Gamma, u : U;\emptyset;\emptyset \vdash u : U}$ (LVar) $\dfrac{}{\Gamma;\{x : U \multimap T\};\emptyset \vdash x : U \multimap T}$

(Session) $\dfrac{}{\Gamma;\emptyset;\{k : S\} \vdash k : S}$

(Subtyping)

(Sub) $\dfrac{\Gamma;\Lambda;\Sigma \vdash P : T \qquad \Sigma \leqslant_c \Sigma' \qquad T \leqslant_c T'}{\Gamma;\Lambda;\Sigma' \vdash P : T'}$

(Functional)

(Abs) $\dfrac{\Gamma, x : U;\Lambda;\Sigma \vdash P : T}{\Gamma;\Lambda;\Sigma \vdash \lambda(x : U).P : U \to T}$ (Abs$_L$) $\dfrac{\Gamma;\Lambda, x : U;\Sigma \vdash P : T}{\Gamma;\Lambda;\Sigma \vdash \lambda(x : U).P : U \to T}$ (Abs$_S$) $\dfrac{\Gamma;\Lambda;\Sigma, x : S \vdash P : T}{\Gamma;\Lambda;\Sigma \vdash \lambda(x : S).P : S \to T}$

(App) $\dfrac{\Gamma;\Lambda_1;\Sigma_1 \vdash P : U \multimap T \qquad \Gamma;\Lambda_2;\Sigma_2 \vdash Q : U \qquad (\dagger)}{\Gamma;\Lambda_1,\Lambda_2;\Sigma_1,\Sigma_2 \vdash PQ : T}$

(Rec) $\dfrac{\Gamma, x : U \to T;\emptyset;\emptyset \vdash \lambda(y : U).P : U \to T}{\Gamma;\emptyset;\emptyset \vdash \mu(x : U \to T).\lambda(y : U).P : U \to T}$

$(\dagger)$ if $U = U' \to T'$ then $\Sigma_2 = \Lambda_2 = \emptyset$.

Figure 4.8: Linear Session Typing: Common and Functional Rules

(Process)

(Nil)
$$\frac{}{\Gamma;\emptyset;\emptyset \vdash \mathbf{0}:\diamond}$$

(New)
$$\frac{\Gamma, a:\langle S\rangle;\Lambda;\Sigma \vdash P:\diamond}{\Gamma;\Lambda;\Sigma \vdash (\nu a:\langle S\rangle)P:\diamond}$$

(New$_s$)
$$\frac{\Gamma;\Lambda;\Sigma, s:S, \overline{s}:\overline{S} \vdash P:\diamond}{\Gamma;\Lambda;\Sigma \vdash (\nu s)P:\diamond}$$

(Conn)
$$\frac{\Gamma;\emptyset;\emptyset \vdash u{:}\langle S\rangle \qquad \Gamma;\Lambda;\Sigma,x:S \vdash P:\diamond}{\Gamma;\Lambda;\Sigma \vdash u(x).P:\diamond}$$

(ConnDual)
$$\frac{\Gamma;\emptyset;\emptyset \vdash u{:}\langle \overline{S}\rangle \qquad \Gamma;\Lambda;\Sigma,x:S \vdash P:\diamond}{\Gamma;\Lambda;\Sigma \vdash \overline{u}(x).P:\diamond}$$

(Recv)
$$\frac{\Gamma,x:U;\Lambda;\Sigma,k:S \vdash P:\diamond}{\Gamma;\Lambda;\Sigma,k{:}?[U].S \vdash k?(x).P:\diamond}$$

(Recv$_L$)
$$\frac{\Gamma;\Lambda,x:U;\Sigma,k:S \vdash P:\diamond}{\Gamma;\Lambda;\Sigma,k{:}?[U].S \vdash k?(x).P:\diamond}$$

(Recv$_S$)
$$\frac{\Gamma;\Lambda;\Sigma,k:S',x:S \vdash P:\diamond}{\Gamma;\Lambda;\Sigma,k{:}?[S].S' \vdash k?(x).P:\diamond}$$

(Send)
$$\frac{\Gamma;\Lambda_1;\Sigma_1 \vdash P:\diamond \qquad \Gamma;\Lambda_2;\Sigma_2 \vdash V:U \qquad k:S \in \Sigma_i \qquad i=1 \text{ or } i=2 \qquad (\dagger)}{\Gamma;\Lambda_1,\Lambda_2;(\Sigma_1,\Sigma_2)\setminus\{k:S\},k:![U].S \vdash k!\langle V\rangle.P:\diamond}$$

(Par)
$$\frac{\Gamma;\Lambda_{1,2};\Sigma_{1,2} \vdash P_{1,2}:\diamond}{\Gamma;\Lambda_1,\Lambda_2;\Sigma_1,\Sigma_2 \vdash P_1 \mid P_2:\diamond}$$

(Bra)
$$\frac{\Gamma;\Lambda;\Sigma,k:S_i \vdash P_i:\diamond \qquad (\forall i \in I)}{\Gamma;\Lambda;\Sigma,k:\&[l_i:S_i]_{i\in I} \vdash k \triangleright \{l_i:P_i\}_{i\in I}:\diamond}$$

(Close)
$$\frac{\Gamma;\Lambda;\Sigma \vdash P:T \qquad k \notin \mathsf{dom}(\Gamma,\Lambda,\Sigma)}{\Gamma;\Lambda;\Sigma,k:\mathsf{end} \vdash P:T}$$

(Sel)
$$\frac{\Gamma;\Lambda;\Sigma,k:S_j \vdash P:\diamond \quad j \in I}{\Gamma;\Lambda;\Sigma,k:\oplus[l_i:S_i]_{i\in I} \vdash k \triangleleft l_j.P:\diamond}$$

$(\dagger)$ if $U = U' \to T'$ then $\Sigma_2 = \Lambda_2 = \emptyset$.

Figure 4.9: Linear Session Typing: Processes

Σ is that, at some stage, the types of dual endpoints need to be compared syntactically (see rule (New_s) in Figure 4.9), and subsumption may be necessary for this, typically for recursive types. An alternative would be to not have $\Sigma \leqslant_c \Sigma'$ in (Sub) and to use $\leqslant_c$ directly in (New_s), but we preferred to only mention $\leqslant_c$ in a single rule.

The second group, **(Function)**, comes from the simply typed linear λ-calculus. There are three abstraction rules, each depending on the shape of the type U of the argument: (Abs) when it is a shared type (not linear function or session type); (Abs_L) when it is a linear function type; and (Abs_S) when it is a session type. In the conclusion of these rules, we remove x from the corresponding environment, because it is now λ-bound in the term. (App) is the rule for functional application; the side condition (†) ensures that when the term on the right is assigned a shared function type, it does not to contain free session endpoints or linear variables. This is a way of ensuring that shared functions that are used as arguments do not contain linear terms, as these unrestricted arguments may be used more than once, breaking linearity, or may not be used at all, again violating linearity by making endpoints or linear functions disappear. The conclusion says that the session environments and linear variable sets of P and Q must be disjoint; otherwise, there is copying (more than one usage) of the respective linear terms, which is forbidden. Note also that in order to obtain the linear type $U \multimap T$ for P, we may need to use subtyping to promote the type from $U \to T$. The purpose of defining the rule with a linear type for the function is to avoid the redundancy of having two similar rules for application, one for each type of function. Rule (Rec) is similar to (Abs), but with the addition of a hypothesis for x in the premise, representing the function itself, and used for typing instances of the function within its body. It is required that the linear function and session environments are empty, since a recursive function may rewrite itself repetitively copying all its contents.

In Figure 4.9 we have the final group, **(Process)**, for processes integrated with linear functional and session typing. Rule (Nil) types the empty process. (New) and (New_s) hide a shared name and a pair of session endpoints, respectively. The latter erases, in the session environment, complementary communication patterns for the two endpoints s and $\bar{s}$, in order to ensure compatible dyadic interactions. Subtyping may need to be used to verify that the session usages are dual.

(Conn) and (ConnDual) are for initiating sessions. In the premises of (Conn), the usage S of the endpoint x in P has to agree with the type $\langle S \rangle$ recorded for the shared identifier u in the typing environment Γ. Rule (ConnDual) is similar, however the type in the environment Γ is dual to the usage in the session body P. This is needed in order to indicate which side of the session is followed with respect to a shared channel type, since connecting processes must use their endpoints dually. As in the case of the abstraction rules, there are three rules for input, depending on the shape of the type of the expected value: (Recv) is for receiving values of a shared type; (Recv_L) types the input of a linear function; and (Recv_S) types the input of a session endpoint. The new session type is composed in the conclusion's session environment, in a way that agrees with the protocol, that is, the input is appended before any subsequent actions on k within P.

(Send) is the most complex rule, integrating session typing and linear typing. First, as in (App), (†) enforces safety when sending shared functions. Secondly, either Σ_1 or Σ_2 contains the complete session $k\!:\!S$, which in practice means that after sending a value, the rest of the session on endpoint k must appear (and be completed) either in the continuation P of the sending process, or inside the value V. In the latter case, we can have that $V = k$ which implements higher-order session passing. The composition Σ_1, Σ_2 is defined in the conclusion, which entails that no endpoint appears in both the remaining sender P and the sent value V, because, in that case, we would have a race condition between the receiver of V and P, in the usage of communications over these common sessions. The same applies to linear variables free in V and P. If V has a functional type, all session endpoints within it must be complete, that is, suffixed with end, because they should not compose further. This is achieved by the necessary use of a suitable instance of (Close). This rule uniformly generalises the corresponding rules in the session types literature [39, 81, 92, 48]. In the conclusion, we delete $k\!:\!S$ where it occurs, either in Σ_1 or Σ_2, and the updated type for k is recorded in the conclusion's session environment, consisting the original type S prefixed with the output $![U]$.

In (Par), we parallel-compose two processes, assuming disjointness of linear function and session environments, as in (App). (Bra) and (Sel) are the standard rules for branching and selection from [48]. In (Bra) all continuations P_i must have corresponding session usages on k that agree with the branch type. In (Sel) the continuation P must have a usage S_j on k that agrees with the type corresponding to the selected label l_j on the selection type of the conclusion.

Closing sessions In the above rules for session communication, the premises always contain a hypothesis for the subject of the session action, e.g. $k : S$ appears in Σ_i located in the premise of the typing for $k!\langle V\rangle.P$. This does not necessarily imply that k appears in P, as the usage $\{k : \mathsf{end}\}$ can be obtained using (Close). This rule is used to effectively close a session on k by introduction of a hypothesis $k\!:\!\mathsf{end}$, in order for further composition (i.e., more session actions on k) to be rejected.

4.3.4 Examples

Here we state a few examples and counter-examples that demonstrate the purpose of the type system.

1. Session endpoints must not become "forgotten":

   $$(\,\lambda(x\!:\!S).\mathbf{0}\,)\cdot s$$

   In the above term, after reduction by the (beta) rule, the endpoint s will not appear any more, and the session on $\bar{s}$ might become stuck. This term is only typable if $S = \mathsf{end}$, otherwise it is not typable because in the premises of rule (Abs_S) we require a session hypothesis $x\!:\!S$ which cannot be introduced in the typing of $\mathbf{0}$ except by use of (Closed).

2. Session endpoints must not be copied:

   $$(\,\lambda(x\!:\!S).(x!\langle V\rangle \mid x!\langle V'\rangle)\,)\cdot s$$

   The above term reduces to:

   $$s!\langle V\rangle \mid s!\langle V'\rangle$$

   in which we have copied s breaking the condition of linearity, which is undesirable as the endpoint $\bar{s}$ will nondeterministically interact with one of the outputs, leaving the other waiting forever. The first term is untypable because typing the body $x!\langle V\rangle \mid x!\langle V'\rangle$ with (Par) requires that the sessions in each parallel process are disjoint, which is not the case here due to the common presence of x.

3. Abstractions that contain running sessions must be used exactly once:

   (a) $(\,\lambda(x:U).x\cdot()\,)\ \cdot\ \ulcorner s!\langle 5\rangle.\mathbf{0}\urcorner \qquad U = \mathsf{unit} \multimap \diamond$

   This term is safe, since the thunk which contains s is used exactly once within the function that receives it. The term is typed using (App) followed by (Abs) and (Abs_L) for the left and right subterms of the application, respectively.

   (b) $(\,\lambda(x:U).\mathbf{0}\,)\ \cdot\ \ulcorner s!\langle 5\rangle.\mathbf{0}\urcorner$

   This term is unsafe as the thunk which contains s does not appear in the function that receives it, after reduction. This is an indirect way for an endpoint to become "forgotten" as before. The typing fails because (Abs_L), used for the left subterm of the application, requires $x : U$ to appear in the linear function environment of the typing of $\mathbf{0}$, which is impossible.

(c) $(\ \lambda(x : U').\mathbf{0}\ ) \cdot \ulcorner \overline{a}(x).x!\langle 5\rangle.\mathbf{0}\urcorner \qquad U' = \mathsf{unit} \rightarrow \diamond$

This term is safe because, although the thunk will not be used in the function, it does not contain any linear or session element that needs to be preserved. The term is typed with (App) followed by (Abs) for the two subterms, respectively.

4.4 Type Soundness and Type Safety

We proceed to show that typed processes enjoy type soundness and type safety. We begin with a number of auxiliary properties, and then prove the Substitution Lemma (page 88), which is necessary in proving Type Soundness (page 94); we finish with Type Safety (page 102).

Lemma 4.4.1 (Closed Judgement). *If* $\Gamma;\Lambda;\Sigma \vdash P : T$ *and* $x \in \mathsf{fv}(P)$ *then* $x \in \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Lambda) \cup \mathsf{dom}(\Sigma)$.

Proof By induction on the typing derivation for P. The interesting cases are the axioms which form the leaves of a derivation. If the last rule is (Shared), (LVar), or (Session), then $P = x$ and x appears in one of typing environments, depending on which axiom was applied. The other cases are easy to obtain using the inductive hypothesis. □

We have the standard weakening and strengthening for Γ, but not for Λ and Σ.

Lemma 4.4.2 (Γ-Weakening). *If* $\Gamma;\Lambda;\Sigma \vdash P : T$ *and* $x \notin \mathsf{dom}(\Gamma,\Lambda,\Sigma)$ *then* $\Gamma, x{:}U;\Lambda;\Sigma \vdash P : T$.

Lemma 4.4.3 (Γ-Strengthening). *If* $\Gamma, x{:}U;\Lambda;\Sigma \vdash P : T$ *and* $x \notin \mathsf{fv}(P)$ *then* $\Gamma;\Lambda;\Sigma \vdash P : T$.

The typing rule (Close) can be used to introduce arbitrary, but ended, hypotheses to the session environment. This is a form of weakening, albeit restricted, and we introduce the following lemma so that we can strengthen the hypotheses by removing any one introduced by (Close). This lemma is used in the proof of Structural Congruence.

Lemma 4.4.4 (Σ-Strengthening). *If* $\Gamma;\Lambda;\Sigma, k{:}\mathsf{end} \vdash P : T$ *and* $k \notin \mathsf{fn}(P)$ *then* $\Gamma;\Lambda;\Sigma \vdash P : T$.

Proof By induction on the typing derivation for P.

Lemma 4.4.5 (Linear Variable Occurrence). *If* $\Gamma;\Lambda, x{:}U \multimap T;\Sigma \vdash P : T$ *then* $x \in \mathsf{fv}(P)$.

Proof By induction on the typing derivation for P. Most cases are straightforward, using the inductive hypothesis. The interesting case is for (LVar), where $P = x$, proving the occurrence of the linear variable. $\square$

Lemma 4.4.6 (Endpoint Occurrence). *If* $\Gamma;\Lambda;\Sigma,x\!:\!S \vdash P : T$ *and* $S \neq \mathsf{end}$ *then* $x \in \mathsf{fv}(P)$.

Proof By induction on the typing derivation for P. Most cases are straightforward, using the inductive hypothesis. The interesting case is for $(\mathsf{Session})$, where $P = x$, proving the occurrence of the endpoint. The sidecondition $S \neq \mathsf{end}$ serves to exclude the cases where x appears in the session environment by introduction through (Close). $\square$

Lemma 4.4.7 (Ended Session). *If* $\Gamma;\Lambda;\Sigma,x\!:\!S \vdash P : T$ *and* $x \notin \mathsf{fv}(P)$ *then* $S = \mathsf{end}$.

Proof By induction on the typing derivation for P. Most cases are straightforward, using the inductive hypothesis. The interesting case is when the last rule applied was (Close), which does not require x to be free in the term, and also implies that $S = \mathsf{end}$. $\square$

Lemma 4.4.8 (Linear Unique Occurrence). *If* $\Gamma;\Lambda,x\!:\!U \multimap T;\Sigma \vdash P : T$*, and* $P = Q_1 \cdot Q_2$ *or* $P = Q_1 \,|\, Q_2$ *or* $P = k!\langle Q_1\rangle.Q_2$ *(in the last case* $Q_1 = V$*), then* $x \notin \mathsf{fv}(Q_i)$ *for* $i = 1$ *or* $i = 2$.

Proof We proceed by induction on the typing derivation for P. Note that we have $x \in \mathsf{fv}(P)$ by Lemma 4.4.5. Suppose $x \in \mathsf{fv}(Q_1)$. Assume $\Lambda,x : U \multimap T \equiv \Lambda_1,\Lambda_2$ and $\Sigma \equiv \Sigma_1,\Sigma_2$. Let $\Gamma;\Lambda_1;\Sigma_1 \vdash Q_1 : T_1$ (1) and $\Gamma;\Lambda_2;\Sigma_2 \vdash Q_2 : T_2$ from the I.H. on the premises of the last rule applied; this was either (App) or (Par) or (Send). From Lemma 4.4.1 we know that since x is free in Q_1 it appears in one of the typing environments of (1), and in particular Λ_1 since by the well-formedness of the assumed judgement for P it cannot appear in Γ or $\Sigma_1 \subseteq \Sigma$ when it appears in $\Lambda,x\!:\!U \multimap T$. Now assume additionally that $x \in \mathsf{fv}(Q_2)$. Then by Lemma 4.4.1 we have that $x \in \mathsf{dom}(\Gamma,\Lambda_2,\Sigma_2)$ which is a contradiction since by the well-formedness of the judgement for P we have that x cannot appear in Γ or $\Lambda_2 \subseteq \Lambda$ or $\Sigma_2 \subseteq \Sigma$. Hence $x \notin \mathsf{fv}(Q_2)$. The case for $x \notin \mathsf{fv}(Q_1)$ is symmetric. $\square$

Lemma 4.4.9 (Endpoint Unique Occurrence). *If* $\Gamma;\Lambda;\Sigma,x\!:\!S \vdash P : T$*, and* $P = Q_1 \cdot Q_2$ *or* $P = Q_1 \,|\, Q_2$ *or* $P = k!\langle Q_1\rangle.Q_2$ *(in the last case* $Q_1 = V$*), then* $x \notin \mathsf{fv}(Q_i)$ *for* $i = 1$ *or* $i = 2$.

Proof The proof is by induction on the typing derivation, and follows the same pattern as in Lemma 4.4.8. When $x \notin \mathsf{fv}(P)$, which is a possibility due to (Close), the result is immediate. When $x \in \mathsf{fv}(P)$ we proceed as in Lemma 4.4.8. $\square$

4.4.1 Substitution

Then the substitution lemma follows. The subcases of the lemma reflect the possible value substitutions that may take place during reduction.

Lemma 4.4.10 (Substitution Lemma).

1. *Suppose* $\Gamma, x{:}U; \Lambda; \Sigma \vdash P : T$ *and* $\Gamma; \emptyset; \emptyset \vdash V : U$. *Then* $\Gamma; \Lambda, \Sigma \vdash P\{V/x\} : T$.

2. *Assume* $\Gamma; \Lambda_1, x{:}U \multimap T'; \Sigma_1 \vdash P : T$ *and* $\Gamma; \Lambda_2; \Sigma_2 \vdash V : U \multimap T'$ *with* Λ_1, Λ_2 *and* Σ_1, Σ_2 *defined. Then* $\Gamma; \Lambda_1, \Lambda_2; \Sigma_1, \Sigma_2 \vdash P\{V/x\} : T$.

3. *Suppose* $\Gamma; \Lambda; \Sigma, x : S \vdash P : T$ *and* $k \notin \mathsf{dom}(\Gamma, \Lambda, \Sigma)$. *Then* $\Gamma; \Lambda; \Sigma, k : S \vdash P\{k/x\} : T$.

Proof The proof is by induction on the last rule applied in the typing derivation for P. In Part (1) we do not state cases where substitution has no effect, as these can be shown trivially from the assumptions with strengthening on the hypothesis for x in $\Gamma, x{:}U$. In Part (2), we assume that substitution is only applied when $x \in \mathsf{fv}(P)$, which is correct since in any judgement $x \in \mathsf{dom}(\Lambda)$ implies that x occurs in the term (see Lemma 4.4.5). For Part (3) we cannot assume that $x \in \mathsf{fv}(P)$, since usages of the shape $x : \mathsf{end}$ can be obtained using (Close) even when x is not free in the term.

Part (1)

Case (Shared) $\quad P = x \quad T = U \quad \Lambda = \Sigma = \emptyset$

We have $P\{V/x\} = V$ by the hypotheses, then $T = U$ and $\Lambda = \Sigma = \emptyset$. Then we use $\Gamma; \emptyset; \emptyset \vdash V : U$ to obtain the required judgement $\Gamma; \Lambda, \Sigma \vdash P\{V/x\} : T$.

Case (LVar) $P = x$ is excluded because $x \in \mathsf{dom}(\Lambda)$ implies $x \notin \mathsf{dom}(\Gamma)$ by the well-formedness of the judgement for P. This case is proved in Part (2).

Case (Session) $P = x$ is excluded because $x \in \mathsf{dom}(\Sigma)$ implies $x \notin \mathsf{dom}(\Gamma)$ by the well-formedness of the judgement for P. This case is proved in Part (3).

Case (Sub) Trivial to show using the I.H. on the premise followed by an application of (Sub).

Case (Abs) $\quad P = \lambda(z : U_1).Q \quad z \neq x \quad T = U_1 \to T_1 \quad \Gamma' = \Gamma, z{:}U_1$

From the I.H. on the premises we have $\Gamma', x{:}U; \Lambda; \Sigma \vdash Q\{V/x\} : T_1$ (1). With an application of (Abs) on (1), binding variable z, we obtain $\Gamma, x{:}U; \Lambda; \Sigma \vdash \lambda(z : U_1).Q\{V/x\} : T$ (2). Now, since by the substitution we have that $x \notin \mathsf{fv}(\lambda(z:U_1).Q\{V/x\})$, we use strengthening on (2) to remove the hypothesis for x and obtain the required judgement.

Case (Abs$_L$) , (Abs$_S$), (Rec) very similar to the case for (Abs).

Case (App) $\quad P = Q_1 \cdot Q_2 \quad \Lambda = \Lambda_1, \Lambda_2 \quad \Sigma = \Sigma_1, \Sigma_2$

From the premises we obtain $\Gamma, x:U; \Lambda_i; \Sigma_i \vdash Q_i\{V/x\} : T_i$ with $T_1 = U' \multimap T$ and $T_2 = U'$. We then apply (App) with the above judgements in the premises, noting that the sidecondition $(\dagger)$ is satisfied, and obtain the result. Strengthening to remove the hypothesis for x (which is not free in the resulting term) is the last step.

Case (Nil) , (New), (New$_S$) are all straightforward to obtain from the premises using the I.H. followed by an application of the respective rule. Removing the hypothesis for x is used as before to obtain the desired shared environment for the final judgement.

Case (Conn) $\quad P = u(z).Q \quad z \neq x \quad T = \diamond$

We take the following cases:

1. Suppose $u = x$. Then we have $\Gamma, x:U; \Lambda; \Sigma \vdash x(z).Q : \diamond$ (1). Also $V = u'$ and from the assumptions $\Gamma; \emptyset; \emptyset \vdash u' : U$ (2) with $U = \langle S \rangle$. We have $P\{V/x\} = u'(z).Q\{V/x\}$. From (1) we obtain the premise $\Gamma, x:U; \Lambda; \Sigma, z:S \vdash Q : \diamond$ (3). Applying the I.H. on (3) we get $\Gamma; \Lambda; \Sigma, z: S \vdash Q\{V/x\} : \diamond$ (4). We now apply (Conn) with (2) and (4) to obtain $\Gamma; \Lambda; \Sigma \vdash u'(z).Q\{V/x\} : \diamond$ as required.

2. Suppose $u \neq x$. Then $P\{V/x\} = u(z).Q\{V/x\}$. From the assumption for P we obtain the premise $\Gamma, x:U; \Lambda; \Sigma \vdash u : \langle S \rangle$ (5). Then since $x \neq u$ we can strengthen the hypotheses and obtain $\Gamma; \Lambda; \Sigma \vdash u : \langle S \rangle$ (6). We obtain (4) as before, and apply (Conn) using (4) and (6) to obtain the required judgement.

Case (ConnDual) is very similar to (Conn).

Case (Recv) , (Recv$_L$), (Recv$_S$) very similar to (Abs).

Case (Send) is similar to (App).

Case (Par) is straightforward to obtain using the I.H. on the premises followed by an application of (Par).

Case (Close) is straightforward to obtain using the I.H. on the premises followed by an application of (Close).

Case (Bra) , (Sel) is easy to prove using the I.H. on the premises. Note that $k \neq x$ by the assumptions since x is assigned a shared type.

Part (2)

Case (Shared) $P = x$ is excluded because x appears in the linear function environment and therefore cannot also be in the shared environment as required by (Shared).

Case (LVar) $P = x \qquad T = U \multimap T' \qquad \Lambda_1 = \emptyset \qquad \Sigma_1 = \emptyset$

From the assumed judgement for P (note that $\Lambda_1 = \emptyset$ and $\Sigma_1 = \emptyset$) we have $\Gamma;\{x{:}U \multimap T'\};\emptyset \vdash x : T$. Then from the assumed judgement for V we have $\Gamma;\Lambda_2;\Sigma_2 \vdash V : U \multimap T'$ and since $P\{V/x\} = V$ and $\Lambda_1 = \Sigma_1 = \emptyset$, this is the required typing judgement.

Case (Session) $P = k = x$ is excluded because x appears in the linear function environment and therefore cannot also be in the session environment as required by (Session), by well-formedness.

Case (Sub) As in Part (1), trivial to show using the I.H. on the premise followed by an application of (Sub).

Case (Abs) $\qquad P = \lambda(z : U_1).Q \qquad z \neq x \qquad T = U_1 \rightarrow T_1$

From the I.H. on the premises of the judgement for P we have (with the hypothesis for x now removed from the linear environment) $\Gamma, z{:}U_1;\Lambda_1,\Lambda_2;\Sigma_1,\Sigma_2 \vdash Q\{V/x\} : T_1$ (1). With an application of (Abs) on (1), binding variable z, we obtain $\Gamma;\Lambda_1,\Lambda_2;\Sigma_1,\Sigma_2 \vdash \lambda(z : U_1).Q\{V/x\} : T$ as required.

Case (Abs$_L$) , (Abs$_S$), (Rec) very similar to the case for (Abs).

Case (App) $\qquad P = Q_1 \cdot Q_2 \qquad \Sigma_1 = \Sigma_{11},\Sigma_{12} \qquad \Lambda_1, x{:}U \multimap T' = \Lambda_{11},\Lambda_{12}$

From the assumption $\Gamma;\Lambda_1,x{:}U \multimap T';\Sigma_1 \vdash P : T$ (1) and Lemma 4.4.5 we have that $x \in \mathsf{fv}(P)$. Using $P = Q_1 \cdot Q_2$ with Lemma 4.4.8 on the assumption we also have that $x \notin \mathsf{fv}(Q_i)$ for some $i \in \{1,2\}$. We can therefore take two cases:

1. Take $x \in \mathsf{fv}(Q_1)$ and $x \notin \mathsf{fv}(Q_2)$. Then $P\{V/x\} = Q_1\{V/x\} \cdot Q_2$. The last rule applied (modulo (Sub)) is (App). By the I.H. on the premise $\Gamma;\Lambda'_{11},x{:}U \multimap T';\Sigma_{11} \vdash Q_1 : U_1 \multimap T$ of (1) we

obtain $\Gamma;\Lambda'_{11},\Lambda_2;\Sigma_{11},\Sigma_2 \vdash Q_1\{^V\!/_x\} : U_1 \multimap T$ (2). The other premise of (1) is $\Gamma;\Lambda_{12};\Sigma_{12} \vdash Q_2 : U_1$ (3). Now we can apply (App) with (2) and (3) as premises. We also need to respect the sidecondition (†): this is already satisfied from the premises of the original application of (App) with respect to (3). We thus obtain the required judgement.

2. The case $x \in \mathsf{fv}(Q_2)$ and $x \notin \mathsf{fv}(Q_1)$ is symmetric. One note is that if U_1 is an arrow-type, then by the sidecondition (†) of (App) we have that $\Lambda_{12} = \Sigma_{12} = \emptyset$, and $x \notin \mathsf{dom}(\Gamma)$ by WF of the assumption, hence in that case we have a contradiction since it must hold that $x \notin \mathsf{fv}(Q_2)$ by Lemma 4.4.1. This verifies our intuition that if a linear variable x appears in Q_2, then Q_2 cannot be typed with a shared function type.

Case (Nil) , (New), (New$_S$) are straightforward using the I.H.

Case (Conn) , (ConnDual) follow a similar pattern to the same cases in Part (1). The proof is slightly simpler since we have that if $P = u(x).Q$ then since x is a linear function variable $u \neq x$.

Case (Recv) , (Recv$_L$), (Recv$_S$) very similar to (Abs).

Case (Send) is similar to (App).

Case (Par) is straightforward to obtain, as before, using the I.H. on the premises followed by an application of (Par).

Case (Close) is straightforward as in the previous part.

Case (Bra) , (Sel) is easy to prove using the I.H. on the premises. Note that $k \neq x$ by the assumptions since x is assigned a linear type.

Part (3)

Most cases are straightforward as before. For the case (App), (Par), the proof is similar to the other parts but makes use of Lemma 4.4.9 (instead of Lemma 4.4.8).

Case (Recv) $\quad P = k'?(z).Q \qquad T = \diamond$

We have $\Gamma;\Lambda;\Sigma,x{:}S \vdash k'?(z).Q : \diamond$. Then we take cases on k'.

1. $k' = x$. Then $P\{^k\!/_x\} = k?(z).Q\{^k\!/_x\}$ and $S = ?[U].S'$. From the premises of (Recv) we obtain $\Gamma,z{:}U;\Lambda;\Sigma,x{:}S' \vdash Q : \diamond$. By the I.H. we have $\Gamma,z{:}U;\Lambda;\Sigma,k{:}S' \vdash Q\{^k\!/_x\} : \diamond$. Then with an application of (Recv) we obtain $\Gamma;\Lambda;\Sigma,k{:}?[U].S' \vdash k?(z).Q\{^k\!/_x\} : \diamond$ as required.

2. $k' \neq x$. Then $P\{^k/_x\} = k'?(z).Q\{^k/_x\}$ and $\Sigma = \Sigma', k' :?[U].S'$. As before by the premises $\Gamma, z:U;\Lambda;\Sigma', k':S', x:S \vdash Q : \diamond$. By the I.H. $\Gamma, z:U;\Lambda;\Sigma', k':S', k:S \vdash Q\{^k/_x\} : \diamond$. Then with an application of (Recv) we obtain $\Gamma;\Lambda;\Sigma, k:S \vdash k'?(z).Q\{^k/_x\} : \diamond$ as required.

Case (Recv$_L$) , (Recv$_S$) very similar to (Recv) above.

Case (Send) $P = k'!\langle V\rangle.Q$ $T = \diamond$ $\Sigma, x : S = (\Sigma_1, \Sigma_2) \setminus \{k' : S'\}, k' :![U].S'$
$\Lambda = \Lambda_1, \Lambda_2$

From the premises of (Send) we have:

$$\begin{aligned}
&\Gamma;\Lambda_1;\Sigma_1 \vdash Q : \diamond && (1)\\
&\Gamma;\Lambda_2;\Sigma_2 \vdash V : U && (2)\\
&k':S' \in \Sigma_i \qquad i = 1 \text{ or } i = 2 && (3)\\
&\text{if } U = U' \to T' \text{ then } \Sigma_2 = \Lambda_2 = \emptyset. && (4)
\end{aligned}$$

Then we perform case analysis on k':

1. Suppose $k' = x$. Then $S =![U].S'$. We now look at the occurrence of x in the session environments:

   (a) Let $x:S' \in \Sigma_1$, then $\Sigma_1 = \Sigma_1', x : S'$ and $P\{^k/_x\} = k!\langle V\rangle.Q\{^k/_x\}$. Using the I.H. on (1) we obtain $\Gamma;\Lambda_1;\Sigma_1', k:S' \vdash Q\{^k/_x\} : \diamond$ (3). Then we apply (Send) with (3) and (2), with the sideconditions clearly satisfied from the assumptions, to obtain $\Gamma;\Lambda;\Sigma_1', \Sigma_2, k:![U].S' \vdash k!\langle V\rangle.Q\{^k/_x\} : \diamond$ as required.

   (b) Let $x : S' \in \Sigma_2$, then $\Sigma_2 = \Sigma_2', x : S'$ and $P\{^k/_x\} = k!\langle V\{^k/_x\}\rangle.Q$. From the I.H. on (2) we have $\Gamma;\Lambda_2;\Sigma_2', k : S' \vdash V\{^k/_x\} : U$ (4). In this case we have $U \neq U' \to T'$ otherwise Σ_2 would be the empty set. We now apply (Send) with (1) and (4) to obtain $\Gamma;\Lambda;\Sigma_1, \Sigma_2', k:![U].S' \vdash k!\langle V\{^k/_x\}\rangle.Q : \diamond$ as required.

2. Suppose $k' \neq x$. As above we look at the occurrence of x in the session environments. Since $k' \neq x$ we have two cases:

   (a) Let $x : S \in \Sigma_1$, then $\Sigma_1 = \Sigma_1', x : S$ and $P\{^k/_x\} = k'!\langle V\rangle.Q\{^k/_x\}$. By the I.H. on (1), $\Gamma;\Lambda_1;\Sigma_1', k:S \vdash Q\{^k/_x\} : \diamond$ (5). We apply (Send) as before with (5) and (2), the side-conditions are satisfied (we do not check where k' occurs in the session environments Σ_1' and Σ_2 as the sidecondition is met by the assumptions), and obtain $\Gamma;\Lambda;(\Sigma_1', \Sigma_2, k: S) \setminus \{k':S'\}, k':![U].S' \vdash k'!\langle V\rangle.Q\{^k/_x\} : \diamond$ as required.

(b) Let $x{:}S \in \Sigma_2$, then $\Sigma_2 = \Sigma_2', x : S$ and $P\{{}^{k}\!/\!{}_{x}\} = k'!\langle V\{{}^{k}\!/\!{}_{x}\}\rangle.Q$. Using the same sequence of steps as before we obtain the result.

Case (Close)

Suppose (Close) was applied for some k'. Then the premise obtained is:

$$\Gamma;\Lambda;(\Sigma, x{:}S) \setminus \{k'{:}\mathsf{end}\} \vdash P : T \quad (1)$$

We now distinguish two cases:

1. $x = k'$. Then $S = \mathsf{end}$. Also, $x \notin \mathsf{dom}(\Gamma,\Lambda,\Sigma)$ by the well-formedness of (1). By Lemma 4.4.1 $x \in \mathsf{fv}(P)$ implies $x \in \mathsf{dom}(\Gamma,\Lambda,\Sigma)$ thus we have $x \notin \mathsf{fv}(P)$. Then $P\{{}^{k}\!/\!{}_{x}\} = P$. From (1) we obtain:

   $$\Gamma;\Lambda;\Sigma \vdash P : T \quad (2)$$

   We have $k \notin \mathsf{dom}(\Gamma,\Lambda,\Sigma)$ by assumption, and we can apply (Close) to obtain:

   $$\Gamma;\Lambda;\Sigma, k{:}\mathsf{end} \vdash P : T$$

   which is the desired result, since $S = \mathsf{end}$ and $P\{{}^{k}\!/\!{}_{x}\} = P$.

2. $x \neq k'$. Then from (1) we obtain:

   $$\Gamma;\Lambda;(\Sigma \setminus \{k'{:}\mathsf{end}\}), x{:}S \vdash P : T \quad (4)$$

   By the I.H. on (4) we have:

   $$\Gamma;\Lambda;(\Sigma \setminus \{k'{:}\mathsf{end}\}), k{:}S \vdash P\{{}^{k}\!/\!{}_{x}\} : T \quad (5)$$

   We now consider two cases:

   (a) $k'{:}\mathsf{end} \in \Sigma$. Then with an application of (Close) on (5) we obtain:

   $$\Gamma;\Lambda;\Sigma, k{:}S \vdash P\{{}^{k}\!/\!{}_{x}\} : T$$

   as required.

   (b) $k'{:}\mathsf{end} \notin \Sigma$. Then $\Sigma \setminus \{k'{:}\mathsf{end}\} = \Sigma$ and the result is immediate from (5).

The remaining session cases are straightforward. □

Lemma 4.4.11 (Shared Value Judgement). *If* $\Gamma;\Lambda;\Sigma \vdash V : U$ *and* $U \in \{\mathsf{unit}, \langle S\rangle\}$ *then* $\Lambda = \Sigma = \emptyset$.

Proof Straightforward to show by induction on the typing derivation. There are two cases to consider: if $U = \mathsf{unit}$ then by (Unit) the result is immediate; if $U = \langle S\rangle$ then by (Shared) the result follows. No other typing rule need to be considered for this type of value. □

Balanced Session Environments

Before stating the main theorems, we introduce the important notion of *balanced* session environments [39]. This formulation is used in theorems, to allow only typings where the two ends of a channel are of dual types, modulo subtyping; otherwise there could be incompatibilities in the structure of communications, which is undesirable.

Definition 4.4.12 (Balanced Session Environment). Formally, we say that a session environment Σ is *balanced*, written $\mathsf{balanced}(\Sigma)$, if whenever $s\colon S_1, \bar{s}\colon S_2 \in \Sigma$, then $S_1 \leqslant_c \overline{S}_2$.

Theorem 4.4.13 (Type Soundness).

1. *Suppose* $\Gamma;\Lambda;\Sigma \vdash P : \diamond$ *with* $\mathsf{balanced}(\Sigma)$. *Then* $P \equiv P'$ *implies* $\Gamma;\Lambda;\Sigma \vdash P' : \diamond$.
2. *Suppose* $\Gamma;\emptyset;\Sigma \vdash P : T$ *with* $\mathsf{balanced}(\Sigma)$. *Then* $P \longrightarrow P'$ *implies* $\Gamma;\emptyset;\Sigma' \vdash P' : T$ *with* $\mathsf{balanced}(\Sigma)$.

Proof In both parts the proof is by induction on the typing derivation for P, given by the assumed judgement, taking cases on the last rule applied, using $\equiv$ in (1), and $\longrightarrow$ in (2). Part (1) is standard, however we present one case which is slightly different. For Part (2) we show the important cases.

Part (1)

Most cases are standard. We are interested in the case of scope extrusion of session endpoints, that is, $(\nu s)\,P \,|\, Q \equiv (\nu s)\,(P \,|\, Q)$ when $s, \bar{s} \notin \mathsf{fn}(Q)$. We need to investigate two directions, corresponding to the two ways in which the rule can be applied.

Case (Scope extrusion (endpoints) $\Rightarrow$) $\quad P = (\nu s)\,R \,|\, Q \quad P' = (\nu s)\,(R \,|\, Q) \quad T = \diamond$

$\mathsf{balanced}(\Sigma) \quad s, \bar{s} \notin \mathsf{fn}(Q)$

The last rule applied is (Par), then from the premises, with $\Lambda = \Lambda_1, \Lambda_2$ and $\Sigma = \Sigma_1, \Sigma_2$, we obtain:

$$\Gamma; \Lambda_1; \Sigma_1 \vdash (\nu s) R : \diamond \quad (1)$$
$$\Gamma; \Lambda_2; \Sigma_2 \vdash Q : \diamond \quad (2)$$

In (1) the last rule applied was (New$_s$), and we obtain the premise:

$$\Gamma; \Lambda_1; \Sigma_1, s : S, \overline{s} : \overline{S} \vdash R : \diamond \quad (3)$$

From the well-formedness of (3), we know that $s, \overline{s} \notin \mathsf{dom}(\Sigma_1)$. However, even though $s, \overline{s} \notin \mathsf{fn}(Q)$, it is not guaranteed that $s, \overline{s} \notin \mathsf{dom}(\Sigma_2)$ because ended hypotheses can be introduced by the use of (Close). If s or $\overline{s}$ or both appear in Σ_2, we cannot apply (Par) on (3) and (2), because the Σ-environment will not be defined.

1. $\Sigma_2 = \Sigma_2', s : S_1$ and $\overline{s} \notin \mathsf{dom}(\Sigma_2')$. By Lemma 4.4.7 we have that $S_1 = \mathsf{end}$. Using Lemma 4.4.4 we can obtain

   $$\Gamma; \Lambda_2; \Sigma_2' \vdash Q : \diamond \quad (4)$$

   Then we apply (Par) on (3) and (4) and obtain:

   $$\Gamma; \Lambda; \Sigma_1, \Sigma_2', s : S, \overline{s} : \overline{S} \vdash R \mid Q : \diamond \quad (5)$$

   With an application of (New$_s$) we obtain:

   $$\Gamma; \Lambda; \Sigma_1, \Sigma_2' \vdash (\nu s)(R \mid Q) : \diamond \quad (6)$$

   Then with an application of (Close) we obtain:

   $$\Gamma; \Lambda; \Sigma_1, \Sigma_2 \vdash (\nu s)(R \mid Q) : \diamond$$

   as required.

2. $\Sigma_2 = \Sigma_2', \overline{s} : \overline{S_1}$ and $s \notin \mathsf{dom}(\Sigma_2')$. As above.

3. $\Sigma_2 = \Sigma_2', s : S_1, \overline{s} : \overline{S_1}$. As before but with two applications of Lemma 4.4.4 and two applications of (Close) in the end.

4. $s, \overline{s} \notin \mathsf{dom}(\Sigma_2)$. Then we apply (Par) followed by (New$_s$) and obtain the required result as before.

□

Case (Scope extrusion (endpoints) $\Leftarrow$) $\quad P = (\nu s)\,(R \mid Q) \quad P' = (\nu s)\,R \mid Q \quad T = \diamond$

$\mathsf{balanced}(\Sigma) \quad s, \overline{s} \notin \mathsf{fn}(Q)$

The last rule applied was (New_s). From the premises we obtain:

$$\Gamma;\Lambda;\Sigma, s\!:\!S, \overline{s}\!:\!\overline{S} \vdash R \mid Q : \diamond \quad (1)$$

Let $\Lambda = \Lambda_1, \Lambda_2$ and $\Sigma, s\!:\!S, \overline{s}\!:\!\overline{S} = \Sigma_1, \Sigma_2$. The last rule applied was (Par) giving the premises:

$$\Gamma;\Lambda_1;\Sigma_1 \vdash R : \diamond \quad (2)$$
$$\Gamma;\Lambda_2;\Sigma_2 \vdash Q : \diamond \quad (3)$$

We now examine the following four cases:

1. $\Sigma_1 = \Sigma_1', s\!:\!S, \overline{s}\!:\!\overline{S}$. Then we can apply ($\mathsf{New}_s$) on (2) followed by (Par) on the result and (3) to obtain the required judgement.

2. $\Sigma_1 = \Sigma_1', s\!:\!S \quad \wedge \quad \overline{s} \notin \mathsf{dom}(\Sigma_1')$. Then we have $\Sigma_2 = \Sigma_2', \overline{s}\!:\!\overline{S}$ and by Lemma 4.4.7 $\overline{S} = \mathsf{end} = S$. By Lemma 4.4.4 on (3) we obtain:

$$\Gamma;\Lambda_2;\Sigma_2' \vdash Q : \diamond \quad (4)$$

With an application of (Close) on (2) we obtain:

$$\Gamma;\Lambda_1;\Sigma_1, \overline{s}\!:\!\overline{S} \vdash R : \diamond \quad (5)$$

With an application of (New_s) on (5) we obtain:

$$\Gamma;\Lambda_1;\Sigma_1' \vdash (\nu s)\,R : \diamond \quad (6)$$

Then applying (Par) on (6) and (4) we obtain:

$$\Gamma;\Lambda;\Sigma_1', \Sigma_2' \vdash (\nu s)\,R : \diamond$$

which is the required judgement since $\Sigma_1', \Sigma_2' = \Sigma$.

3. $\Sigma_1 = \Sigma_1', \overline{s}\!:\!\overline{S} \quad \wedge \quad s \notin \mathsf{dom}(\Sigma_1')$. Similar to above.

4. $s, \overline{s} \notin \mathsf{dom}(\Sigma_1)$. Similar to the above case but with two uses of Lemma 4.4.4 on (3) and two applications of (Close) on (2).

$\square$

Part (2)

The interesting cases are (beta), (conn), and (comm). The case of (rec) is almost identical to (beta), and simpler due to the absence of hypotheses in the session environment due to the repetitive actions of the term. Notice that the assumed judgement has an empty linear environment, but this is not a restriction, because reduction is only defined between closed terms.

Case (beta) $\quad P = (\lambda(x\!:\!U).Q)\,V \qquad P' = Q\{^V\!/\!_x\} \qquad \Sigma = \Sigma_1, \Sigma_2 \qquad \mathsf{balanced}(\Sigma)$

The last rule in the derivation for the assumed judgement was (App) from which we obtain the premises:

$$\Gamma;\emptyset;\Sigma_1 \vdash \lambda(x\!:\!U).Q : U \multimap T \quad (1)$$
$$\Gamma;\emptyset;\Sigma_2 \vdash V : U \quad (2)$$

Also, by the sidecondition (†), if $U = U' \to T'$ then $\Sigma_2 = \emptyset$.

From (1) we have that the last rule applied was (Sub), as it is the only rule to introduce the linear function type, with premise:

$$\Gamma;\emptyset;\Sigma_1' \vdash \lambda(x\!:\!U).Q : U \to T' \quad (3)$$
$$U \to T' \leqslant_c U \multimap T \quad (4)$$
$$\Sigma_1' \leqslant_c \Sigma_1 \quad (5)$$

To accommodate for the different abstraction rules that could have been used to obtain (3), we will need to take cases on the type U.

1. $U \in \{\mathsf{unit}, \langle S\rangle, U_1 \to T_1\}$. Then the rule used to type (3) was (Abs), and we obtain the premise:

$$\Gamma, x\!:\!U;\emptyset;\Sigma_1' \vdash Q : T' \quad (6)$$

Looking at (2), by Lemma 4.4.11, if $U \in \{\mathsf{unit}, \langle S\rangle\}$, we have that $\Sigma_2 = \emptyset$. By the sidecondition (†), if $U = U_1 \to T_1$, we also have $\Sigma_2 = \emptyset$. We can therefore use Lemma 4.4.10, part (1), to obtain:

$$\Gamma;\emptyset;\Sigma_1' \vdash Q\{^V\!/\!_x\} : T' \quad (6)$$

Using (Sub) with premises (6), (5), (4) we obtain:

$$\Gamma;\emptyset;\Sigma_1 \vdash Q\{^V\!/_x\} : T$$

which, since $\Sigma_2 = \emptyset$, is equivalent to:

$$\Gamma;\emptyset;\Sigma \vdash Q\{^V\!/_x\} : T$$

with $\mathsf{balanced}(\Sigma)$, from the assumptions, as required.

2. $U = U_1 \multimap T_1$. Then the rule used to type (3) was (Abs_L), and we obtain the premise:

$$\Gamma;\{x{:}U\};\Sigma_1' \vdash Q : T' \quad (7)$$

Then using Lemma 4.4.10, part (2), we obtain from (7):

$$\Gamma;\emptyset;\Sigma_1',\Sigma_2 \vdash Q\{^V\!/_x\} : T' \quad (8)$$

Using (Sub) with premises (8), (5), (4), and since $\Sigma_1',\Sigma_2 \leqslant_c \Sigma_1,\Sigma_2$, we obtain:

$$\Gamma;\emptyset;\Sigma \vdash Q\{^V\!/_x\} : T$$

and as before $\mathsf{balanced}(\Sigma)$, as required.

3. $U = S$. Then the rule used to type (3) was (Abs_S), and we obtain the premise:

$$\Gamma;\emptyset;\Sigma_1',x{:}S \vdash Q : T' \quad (9)$$

We also have that $V = k$ and $\Sigma_2 = \{k{:}S\}$. Then using Lemma 4.4.10, part (3), we obtain from (9):

$$\Gamma;\emptyset;\Sigma_1',k{:}S \vdash Q\{^k\!/_x\} : T' \quad (10)$$

Using (Sub) with premises (10), (5), (4), and since $\Sigma_1',k{:}S \leqslant_c \Sigma_1,k{:}S$ and $\Sigma_2 = \{k{:}S\}$, we obtain:

$$\Gamma;\emptyset;\Sigma \vdash Q\{^k\!/_x\} : T$$

with $\mathsf{balanced}(\Sigma)$ as required.

□

Case (conn) $P = a(x).Q \mid \overline{a}(z).R$ $P' = (\nu s)\,(Q\{{}^{s}\!/\!{}_{x}\} \mid R\{{}^{\overline{s}}\!/\!{}_{z}\})$ $s, \overline{s} \notin \mathsf{fn}(Q,R)$

$T = \diamond$ $\Sigma = \Sigma_1, \Sigma_2$ $\mathsf{balanced}(\Sigma)$

The last rule applied is (Par), from the premises of which we obtain:

$$\Gamma;\emptyset;\Sigma_1 \vdash a(x).Q : \diamond \quad (1)$$
$$\Gamma;\emptyset;\Sigma_2 \vdash \overline{a}(z).R : \diamond \quad (2)$$

The last rule applied for (1) is (Conn), with premises:

$$\Gamma;\emptyset;\emptyset \vdash a : \langle S\rangle \quad (3)$$
$$\Gamma;\emptyset;\Sigma_1, x{:}S \vdash Q : \diamond \quad (4)$$

The last rule applied for (2) is (ConnDual), with premises:

$$\Gamma;\emptyset;\emptyset \vdash a : \langle S\rangle \quad (5)$$
$$\Gamma;\emptyset;\Sigma_2, z{:}\overline{S} \vdash R : \diamond \quad (6)$$

Above we used the fact that $\overline{\overline{S}} = S$. Since $s, \overline{s} \notin \mathsf{fn}(Q,R)$ we can apply Lemma 4.4.10, part (3), on (4) and (6), respectively, to obtain:

$$\Gamma;\emptyset;\Sigma_1, s{:}S \vdash Q\{{}^{s}\!/\!{}_{x}\} : \diamond \quad (7)$$
$$\Gamma;\emptyset;\Sigma_2, \overline{s}{:}\overline{S} \vdash R\{{}^{\overline{s}}\!/\!{}_{z}\} : \diamond \quad (8)$$

We can now apply (Par) with premises (7) and (8), since $\Sigma_1, \Sigma_2, s : S, \overline{s} : \overline{S}$ is defined, followed by an application of (New$s$), to obtain:

$$\Gamma;\emptyset;\Sigma \vdash P' : \diamond$$

with $\mathsf{balanced}(\Sigma)$ as required. □

Case (comm) $P = k?(x).Q \mid \overline{k}!\langle V\rangle.R$ $P' = Q\{{}^{V}\!/\!{}_{x}\} \mid R$ $k = s$ or $k = \overline{s}$

$T = \diamond$ $\Sigma = \Sigma_1, \Sigma_2$ $\mathsf{balanced}(\Sigma)$

The last rule applied was (Par), and from the premises we obtain:

$$\Gamma;\emptyset;\Sigma_1 \vdash k?(x).Q : \diamond \quad (1)$$
$$\Gamma;\emptyset;\Sigma_2 \vdash \overline{k}!\langle V\rangle.R : \diamond \quad (2)$$

In (2), the last rule was (send), and we obtain the premises:

$$\begin{array}{ll}
\Gamma;\emptyset;\Sigma_{21} \vdash V : U & (3) \\
\Gamma;\emptyset;\Sigma_{22} \vdash R : \diamond & (4) \\
\overline{k}:S \in \Sigma_{2i} & (5) \\
\Sigma_2 = (\Sigma_{21},\Sigma_{22}) \setminus \{\overline{k} : S\}, \overline{k} : ![U].S & (6) \\
U = U' \to T' \implies \Sigma_{21} = \emptyset & (7)
\end{array}$$

Since the environment Σ is balanced, and k occurs in Σ_1, we have that $\Sigma_1 = \Sigma_1', k:?[U].\overline{S}$. We now take cases on the shape of U:

1. $U \in \{\mathsf{unit}, \langle S\rangle, U_1 \to T_1\}$. Then the last rule applied in (1) was (Recv), and we obtain the premise:

   $$\Gamma, x:U;\emptyset;\Sigma_1', k:\overline{S} \vdash Q : \diamond \quad (8)$$

   In (3), by Lemma 4.4.11, if $U \in \{\mathsf{unit}, \langle S\rangle\}$, we have that $\Sigma_{21} = \emptyset$. Also, by (7) if $U = U' \to T'$ then again $\Sigma_{21} = \emptyset$. We can now apply Lemma 4.4.10, part (1), on (8) and (3), to obtain:

   $$\Gamma;\emptyset;\Sigma_1', k:\overline{S} \vdash Q\{V/x\} : \diamond \quad (9)$$

   We proceed by applying (Par) using (9) and (4) in the premises, and obtain:

   $$\Gamma;\emptyset;\Sigma_1', \Sigma_{22}, k:\overline{S} \vdash Q\{V/x\} : \diamond \quad (10)$$

   Since $\Sigma_{21} = \emptyset$ from (5) we have that $\overline{k}:S \in \Sigma_{22}$. Let $\Sigma_{22} = \Sigma_{22}', \overline{k}:S$. Then $\Sigma_1', \Sigma_{22}' \subset \Sigma$ and therefore the final session environment $\Sigma' = \Sigma_1', \Sigma_{22}', k:\overline{S}, \overline{k}:S$ is balanced, as required.

2. $U = U_1 \multimap T_1$. Then the rule used to type (1) was (Recv_L), and we obtain the premise:

   $$\Gamma;\{x:U\};\Sigma_1', k:\overline{S} \vdash Q : \diamond \quad (11)$$

   We then apply Lemma 4.4.10, part (2), with (11) and (3) to obtain:

   $$\Gamma;\emptyset;\Sigma_1', \Sigma_{21}, k:\overline{S} \vdash Q\{V/x\} : \diamond \quad (12)$$

Next we apply (Par) with (12) and (4) to obtain:

$$\Gamma;\emptyset;\Sigma_1',\Sigma_{21},\Sigma_{22},k:\overline{S} \vdash Q\{V/x\} : \diamond \quad (13)$$

Let Σ' be the final environment with $\Sigma' = \Sigma_1',\Sigma_{21},\Sigma_{22},k:\overline{S}$. From (5) we have that $\Sigma_2 = \Sigma_{21}',\Sigma_{22}',\overline{k}:S$. Therefore $\Sigma' = \Sigma_1',\Sigma_{21}',\Sigma_{22}',\overline{k}:S,k:\overline{S}$ and since $\Sigma_1',\Sigma_{21}',\Sigma_{22}' \subset \Sigma$ we have that Σ' is balanced as required.

3. $U = S'$. Then $V = k'$ and, from (3), $\Sigma_{21} = \{k':S'\}$. The rule used to type (1) was (Recv_S), and we obtain the premise:

$$\Gamma;\emptyset;\Sigma_1',k:\overline{S},x:S' \vdash Q : \diamond \quad (14)$$

We then apply Lemma 4.4.10, part (3), with (14) and (3) to obtain:

$$\Gamma;\emptyset;\Sigma_1',k:\overline{S},k':S' \vdash Q\{k'/x\} : \diamond \quad (15)$$

Next we apply (Par) with (15) and (4) to obtain:

$$\Gamma;\emptyset;\Sigma_1',\Sigma_{22},k:\overline{S},k':S' \vdash Q\{V/x\} : \diamond \quad (16)$$

Let Σ' be the final environment with $\Sigma' = \Sigma_1',\Sigma_{22},k:\overline{S},k':S'$. We consider two cases (recalling that a session can emit itself):

(a) $\overline{k} = k'$. Then using (5), we obtain that $S' = S$. The final environment Σ' becomes $\Sigma_1',\Sigma_{22},k:\overline{S},\overline{k}:S$ and $\Sigma_1',\Sigma_{22} \subset \Sigma$ hence $\mathsf{balanced}(\Sigma')$ as required.

(b) $\overline{k} \neq k'$. Then from (5) we have $\Sigma_{22} = \Sigma_{22}',\overline{k}:S$. The final environment Σ' becomes $\Sigma_1',\Sigma_{22}',\overline{k}:S,k:\overline{S},k':S'$ and since $\Sigma_{21} = \{k':S'\}$ we obtain $\Sigma_1',\Sigma_{22}',\overline{k}:S,k:\overline{S},\Sigma_{21}$. We have $\Sigma_1',\Sigma_{22}',\Sigma_{21} \subset \Sigma$ therefore $\mathsf{balanced}(\Sigma')$ as required.

□

Type safety

We now formalise type safety. First, a *k-process* is a prefixed process with subject k (such as $k?(x)$ and $k!\langle V\rangle$). Next, a *k-redex* is a parallel composition of a k-process and a $\overline{k}$-process, of the form $(k!\langle V\rangle P \mid \overline{k}?(x)Q)$ or $(k \lhd l_m.P \mid \overline{k} \rhd \{l_1 : Q_1;\cdots l_n : Q_n\})$ with $1 \leq m \leq n$. Then we say P is an *error* if $P \equiv (\nu\vec{a})(\nu\vec{s})(Q \mid R)$ where Q is, for some k, the $|$-composition of *either* a k-process and a $\overline{k}$-process that do not form a k-redex, *or* two k-processes. The last class of error subsumes the case of more than two k-processes in parallel. We then have:

Theorem 4.4.14 (Type Safety). *A typable process* $\Gamma;\Lambda;\Sigma \vdash P : \diamond$ *with* $\mathsf{balanced}(\Sigma)$ *never reduces into an error.*

Proof By Type Soundness (Theorem 4.4.13), it is enough to show that error processes are not typable. We prove this by contradiction. Assume $P \equiv (\nu\vec{a})(\nu\vec{s})(Q \mid R)$ and let Q be an error process. We consider two cases depending on the class of the error:

1. Q is the $|$-composition of a k-process and a $\overline{k}$-process that do not form a k-redex. All combinations are similar. Take for instance $Q = k!\langle V\rangle.Q_1 \mid \overline{k} \lhd l.Q_2$. Then by the typability of P we have that (Par) can be applied on Q giving $\Gamma;\Lambda_1,\Lambda_2;\Sigma_1,\Sigma_2 \vdash Q : \diamond$ (1) and from its premises we obtain $\Gamma;\Lambda_1;\Sigma_1 \vdash k!\langle V\rangle.Q_1 : \diamond$ (2) and $\Gamma;\Lambda_2;\Sigma_2 \vdash \overline{k} \lhd l.Q_2 : \diamond$ (3). In (2) we can apply (Send) and therefore $\Sigma_1 = \Sigma_1', k{:}![U].S_1$. In (3) we can apply (Sel) and thus $\Sigma_2 = \Sigma_2', \overline{k}{:}\oplus[l_1{:}S_1,\ldots,l_n{:}S_n]$. Since $\mathsf{balanced}(\Sigma)$ and $\Sigma_1,\Sigma_2 \subseteq \Sigma$ we have $\mathsf{balanced}(\Sigma_1,\Sigma_2)$. But then we require $![U].S_1 \leqslant_c \overline{\oplus[l_1{:}S_1,\ldots,l_n{:}S_n]}$ which is impossible to derive, hence we arrive to a contradiction. Other combinations that do not form redices are proved similarly.

2. Q is the $|$-composition of two k-processes. As before all combinations are proved in the same way. Let $Q = k!\langle V\rangle.Q_1 \mid k?(x).Q_2$. If we apply (Par) then from the premises we obtain $\Gamma;\Lambda_1,\Lambda_2;\Sigma_1,\Sigma_2 \vdash Q : \diamond$ and from the latter's premises we obtain $\Gamma;\Lambda_1;\Sigma_1 \vdash k!\langle V\rangle.Q_1 : \diamond$ and $\Gamma;\Lambda_2;\Sigma_2 \vdash k?(x).Q_2 : \diamond$. We then have that $\Sigma_1 = \Sigma_1', k{:}![U].S$ and $\Sigma_2 = \Sigma_2', k{:}?[U'].S'$ from typing the premises with (Send) and (Recv) (or $(\mathsf{Recv}_{L/S})$). But then Σ_1,Σ_2 is undefined as k occurs in the domain of both Σ_1 and Σ_2. Other cases are similar.

□

4.4.2 Typing the Hotel Booking Example

Using the typing system, we can now type the hotel booking example in § 4.2.4, guaranteeing its type safety. `Agent` has the following types at a and b.

$$a : \langle ![\mathtt{string}]\ldots \oplus [\mathsf{rtt} < 100 :\ S_1\ ;\ \mathsf{rtt} \geq 100 : S_1\ ]\rangle,\ b : \langle ![S_2].\mathsf{end}\rangle$$

$$\text{with } S_1 = \&[\mathsf{cont} : ?[S_2].![\mathtt{int}].\mathsf{end}\ ;\ \mathsf{move} : ?[S_2].![\mathtt{int}].\mathsf{end}]$$

$$\text{and } S_2 = \&[\mathsf{cont} :![\mathtt{string}].?[\mathtt{int}]\ldots\mathsf{end}\ ;\ \mathsf{move} :![\ulcorner\diamond\urcorner^1].\mathsf{end}]$$

Note that the type of a is dualised because a is used as the input in `Agent` (see (Acc)). S_1 consists of higher-order session passing, and the thunk has a linear arrow type. `Client` and `Hotel` just have the dual of `Agent`'s type at a and the dual of `Agent`'s type at b, respectively. Note that in `Client`, subject y is shared in the sent code V, which is typed by (Send) with a general side condition $k : S_1 \in \Sigma_2$ explained in § 5.4.

4.5 Concluding Remarks

Our typing system is substructural in the sense that for session environments Σ we do not allow weakening and contraction, ensuring that a session channel is recorded as having been used only when it actually occurs in session communication expressions. Similarly no structural transformations can apply to linear variable environments, ensuring that the occurrence of a variable manifests that it has indeed been used exactly once. Note that, in our system there is no need to enforce linear usage for other than functional types. Applying the inference techniques of [31, 30] and [84], with the algorithmic subtyping of [39], it may be possible to construct a type inference system.

$\boxed{5}$ Asynchronous Session Subtyping

Overview *Here we introduce session subtyping into a buffered version of the process language HOπ^{s} of the previous chapter. We define a coinductive subtyping relation between session types corresponding to what is intuitively understood as more asynchronous behaviour.*

5.1 Introduction

Our recent work [65] developed a new subtyping, *asynchronous subtyping*, that characterises compatibility between classes of permutations of communications within asynchronous protocols, offering much greater flexibility. However, an interesting development remained: *how to uniformly introduce communication optimisations in the presence of code mobility [63], incorporating higher-order sessions and functions into the asynchronous subtyping [65, § 6].* This is the question we address in the paper [64], which is the basis for the material in this chapter.

Higher-Order Processes with Asynchronous Sessions. We develop a session typing system for the Higher-order π-calculus [79], an amalgamation of call-by-value λ-calculus and π-calculus, extending [63]. Code mobility is facilitated by sending not just ground values and channels, but also abstracted processes that can be received and activated locally, reducing the number of transmissions of remote messages. The simplest code mobility operations are sending a thunked process $\ulcorner P \urcorner$ via channel s (denoted as $s!\langle \ulcorner P \urcorner \rangle$), and receiving and running it by applying the unit (denoted as $s?(x).x\,())$. In our calculus, communications are always within a *session*, established when accept and receive processes synchronise on a shared channel:

$$a(x).x!\langle 5\rangle.x!\langle \mathtt{true}\rangle.x?(y).(y() \mid R) \mid \overline{a}(x).x?(z_1).x?(z_2).x!\langle \ulcorner P \urcorner \rangle$$

resulting in a fresh session, consisting two channels s and $\overline{s}$, each private to one of the two processes, and their associated queues initialised to be empty:

$$(\nu s)(s!\langle 5\rangle.s!\langle \mathtt{true}\rangle.s?(y).(y() \mid R) \mid \overline{s}?(z_1).\overline{s}?(z_2).\overline{s}!\langle \ulcorner P \urcorner \rangle \mid s:\varepsilon \mid \overline{s}:\varepsilon)$$

To avoid conflicts, an output on a channel s (resp. $\bar{s}$) places the value on the *dual* queue $\bar{s}$ (resp. s), while an input on s reads from s (resp. for $\bar{s}$). Thus, after two steps the outputs of 5 and `true` are placed on queue $\bar{s}$ as follows:

$$(\nu s)(s?(y).(y() \mid R) \mid \bar{s}?(z_1).\bar{s}?(z_2).\bar{s}!\langle\ulcorner P\urcorner\rangle \mid s:\varepsilon \mid \bar{s}:5\cdot\mathtt{true})$$

and in two more steps the right process receives and reduces to $\bar{s}!\langle\ulcorner P\urcorner\{5/z_1\}\{\mathtt{true}/z_2\}\rangle$. Similarly the next step transmits the thunked process, and R can interact with P locally. The session type of $\bar{s}$, $S = ?[\mathsf{nat}].?[\mathsf{bool}].![U].\mathsf{end}$ (where U is the type of $\ulcorner P\urcorner$), guarantees that values are received following the order specified by S.

Asynchronous Communication Optimisation with Code Mobility Suppose the size of P is very large and it does not contain z_1 and z_2. Then the right process might wish to start transmission of P to $s:\varepsilon$ concurrently without waiting for the delivery of 5 and `true`, since the sending is non-blocking. Thus we send $\ulcorner P\urcorner$ ahead as in $\bar{s}!\langle\ulcorner P\urcorner\rangle.\bar{s}?(z_1).\bar{s}?(z_2).\mathbf{0}$. The interaction with the left process is *safe* as the outputs are ordered in an exact complementary way. However the optimised code is not composable with the other party by the original session system [81] since it cannot be assigned S. To make this optimisation valid, we proposed the *asynchronous subtyping* in [65] by which we can refine a protocol to maximise asynchrony without violating the session. For example, in the above case, $S' = ![U].?[\mathsf{nat}].?[\mathsf{bool}].\mathsf{end}$ is an asynchronous subtype of S, hence the resulting optimisation is typable.

The idea of this subtyping is intuitive and the combination of two kinds of optimisations is vital for typing many practical protocols [82, 90] and parallel algorithms [67], but it requires subtle formal formulations due to the presence of higher-order code. The linear functional typing developed in [63] permits to send a value that contains free session channels: for example, not only message $\bar{s}!\langle\ulcorner s'?(x).s'!\langle x\rangle\urcorner\rangle$ (for $\bar{s}!\langle\ulcorner P\urcorner\rangle$), but also one which contains its own session $\bar{s}!\langle\ulcorner\bar{s}?(x).\bar{s}!\langle x\rangle\urcorner\rangle$ is typable (if R conforms with the dual session like $R = s!\langle 7\rangle.s?(z).\mathbf{0}$). The first message can go ahead correctly, but the permutation of the second message (as $\bar{s}!\langle\ulcorner P\urcorner\rangle$) violates safety since the input action $\bar{s}?(x)$ will appear in parallel with $\bar{s}?(z_1).\bar{s}?(z_2)$, creating a race condition, as seen in:

$$(\nu s)(\bar{s}?(x).\bar{s}!\langle x\rangle \mid R \mid \bar{s}?(z_1).\bar{s}?(z_2).\mathbf{0} \mid s:\varepsilon \mid \bar{s}:5\cdot\mathtt{true})$$

Our paper [64] shows that the combination of two optimisations is indeed possible by establishing soundness and communication-safety, subsuming the original typability from [63]. The technical challenge is to prove the transitivity of the asynchronous subtyping integrated with higher-order (linear) function types and session-delegation, since the types now appear in arbitrary contravariant positions [65]. Moreover the definitions are now exposed more constructively. Another challenge is to formulate a runtime typing system which handles both stored higher-order code with open sessions and asynchronous subtyping. We demonstrate all facilities of type-preserving optimisations proposed in [64] by using an e-commerce scenario.

Terms

$$
\begin{array}{rcll}
P,Q,R & ::= & \ldots & \\
 & \mid & s : \vec{h} & \text{queue}
\end{array}
$$

Messages

$$
\begin{array}{rcll}
h & ::= & l & \text{label} \\
 & \mid & V & \text{value}
\end{array}
$$

Figure 5.1: Syntax modifications for Asynchronous Higher-order π-calculus

5.2 The Higher-Order π-Calculus with Asynchronous Sessions

5.2.1 Syntax and Reduction

We modify and extend the Synchronous Higher-Order calculus of Chapter 4 ($\mathrm{HO}\pi^{\mathbf{s}}$), adding to the syntax defined in Figure 4.1 the productions of Figure 5.1. We call the new calculus $\mathrm{HO}\pi^{\mathbf{as}}$, the Asynchronous Higher-Order π-calculus with sessions. The new components are *message queues*, also called *buffers*, and written $s : \vec{h}$. A queue $s : \vec{h}$ provides access, via a session that uses s, to the ordered messages $\vec{h}$. It can be thought of as a network pipe in a TCP-like transport mechanism. The messages can be values, or labels which are required for selection and branching. The *dual* of a queue endpoint s is denoted by $\bar{s}$, and represents the other endpoint of the same session. The operation is self-inverse hence

$$\bar{\bar{s}} = s$$

Note that queues and session restrictions appear only at runtime. A *program* is a process which does not contain runtime terms. Other primitives are standard. We often omit **0**.

The *bindings* remain the same. The derived notions of bound and free identifiers, alpha equivalence and substitution are standard; see Figure 4.2. Free names, however, are extended to queue processes (which can contain labels) as follows:

$$\mathsf{fn}(l) = \emptyset \qquad\qquad \mathsf{fn}(s : h_1 \ldots h_n) = (\cup_{i \in 1..n} \mathsf{fn}(h_i)) \cup \{s\}$$

The single-step call-by-value reduction relation, denoted $\longrightarrow$, is defined as a modification of the original rules of Figure 4.4, using the rules in Figure 5.2.

Rule (conn) is modified, to establish a new session between server and client via shared name u, generating two fresh session channels and the associated empty queues (ε denotes the empty sequence). The original (comm) and (label) are removed and replaced with asynchronous rules for session communication. Rules (send) and (sel) respectively enqueue a value and a label at the tail of the queue for a dual endpoint $\overline{k}$. Rules (recv) and (bra) dequeue, from the head of the queue, a value or label. (recv) substitutes value V for x in P, while (bra) selects the corresponding branch for index m.

Since (conn) provides a queue for each channel, these rules say that a sending action is never blocked (*asynchrony*) and that two messages from the same sender to the same channel arrive in the sending order (*order preservation*). Other rules are standard.

We use the standard structural congruence rules [59], defined in Figure 4.3. We add a non-standard rule, for *garbage collecting* queues from completed sessions:

$$(\nu s)\,(s\!:\!\varepsilon \mid \bar{s}\!:\!\varepsilon) \equiv \mathbf{0} \qquad \text{garbage collection}$$

With "$\twoheadrightarrow$" we denote the multi-step reduction defined as $(\equiv \cup \rightarrow)^*$.

5.2.2 Example: Optimised Business Protocol with Code Mobility

We show a business/financial protocol interaction from [90, 82] which integrates the two kinds of type-safe optimisations. We extend the scenario from [63] to highlight the expressiveness gained using the new method. Figure 5.3 draws the sequencing of actions modelling a hotel booking through a process `Agent`. On the left `Client` behaves dually to `Agent`; on the right, an optimised `MClient` utilises type-safe asynchronous behaviour.

The `Agent` behaves the same towards both clients: initially it calculates the round-trip time (RTT) of communication (`rtt`) and sends it; it then offers to the other party the option to consider the RTT and either send *mobile code* to interact with the `Agent` on its location, or to continue the protocol with each executing remotely their behaviour. When mobile code (after choice move) is

(beta)

$$(\lambda(x:U).P)V \longrightarrow P\{^{V}\!/_{x}\}$$

(rec)

$$(\mu y.\lambda x.P)V \longrightarrow P\{^{V}\!/_{x}\}\{^{\mu y.\lambda x.P}\!/_{y}\}$$

(send)

$$k!\langle V\rangle.P \mid \overline{k}:\vec{h} \longrightarrow P \mid \overline{k}:\vec{h}\cdot V$$

(recv)

$$k?(x).P \mid k:V\cdot\vec{h} \longrightarrow P\{^{V}\!/_{x}\} \mid k:\vec{h}$$

(sel)

$$k \lhd l.P \mid \overline{k}:\vec{h} \longrightarrow P \mid \overline{k}:\vec{h}\cdot l$$

(bra)

$$k \rhd \{l_1:P_1,\ldots,l_n:P_n\} \mid k:l_m\cdot\vec{h} \longrightarrow P_m \mid k:\vec{h} \qquad 1\leq m\leq n$$

(conn)

$$a(x).P \mid \overline{a}(z).Q \longrightarrow (\nu s)\,(P\{^{s}\!/_{x}\} \mid Q\{^{\overline{s}}\!/_{z}\} \mid s:\varepsilon \mid \overline{s}:\varepsilon) \quad s,\overline{s} \notin \mathsf{fn}(P,Q)$$

$$k = s \text{ or } k = \overline{s}$$

(app-l) $\dfrac{P\longrightarrow P'}{PQ\longrightarrow P'Q}$ (app-r) $\dfrac{Q\longrightarrow Q'}{VQ\longrightarrow VQ'}$ (par) $\dfrac{P\longrightarrow P'}{P\,|\,Q\longrightarrow P'\,|\,Q}$

(resc) $\dfrac{P\longrightarrow P'}{(\nu a:\langle S\rangle)P\longrightarrow(\nu a:\langle S\rangle)P'}$ (ress) $\dfrac{P\longrightarrow P'}{(\nu s)P\longrightarrow(\nu s)P'}$ (str) $\dfrac{P\equiv P'\longrightarrow Q'\equiv Q}{P\longrightarrow Q}$

Figure 5.2: Reduction

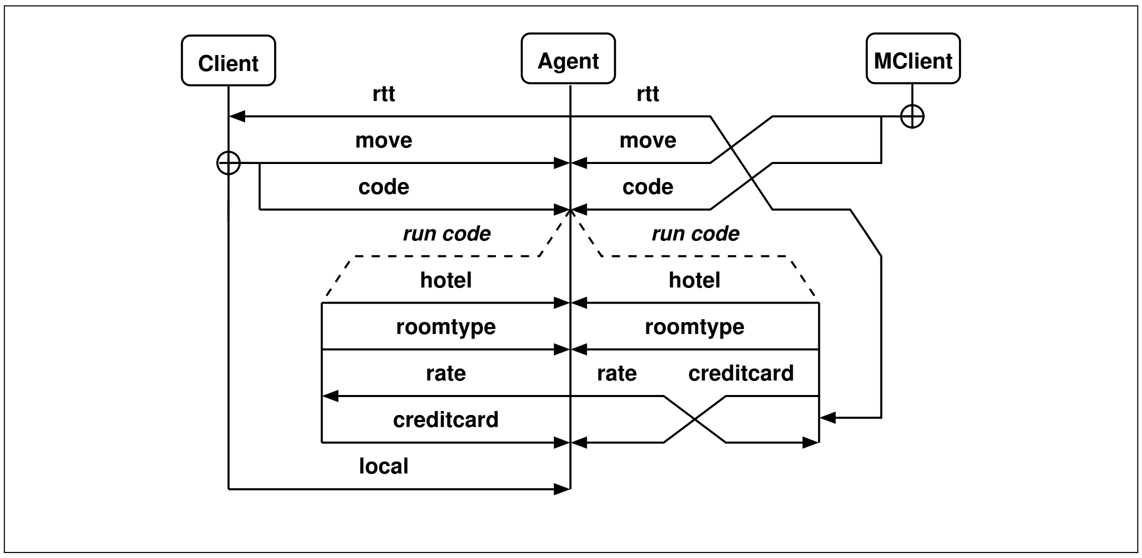

Figure 5.3: Standard (left) and Optimised (right) Interaction for Hotel Booking

received, it is *run* by the `Agent` completing the transaction on behalf of the client, in a sequence of steps. The behaviour of `Client` is straightforward and complementary to `Agent`, but `MClient` has special requirements: it represents a mobile device with limited processing power, and irrespective of the RTT it always sends mobile code; moreover, it does not care about money, and provides the creditcard number (`card`) before finding out the `rate`.

To represent this optimised scenario, we start from the process for `Agent`:

$$\begin{aligned} \texttt{Agent} &= a(x).x!\langle\texttt{rtt}\rangle.x \rhd \{\mathsf{move} : x?(code).(run\ code \mid Q), \mathsf{local} : Q\} \\ Q &= x?(hotel).x?(roomtype).x!\langle\texttt{rate}\rangle.x?(creditcard)\ldots \end{aligned}$$

The session is initiated over a, then the `rtt` is sent, then the choices `move` and `local` are offered. If the first choice is made then the received code is run in parallel to the process Q which continues the agent's session, performing optimisation by code mobility. As expected, `Client` has dual behaviour:

$$\texttt{Client} = \overline{a}(x).x?(rtt).x \lhd \mathsf{move}.x!\langle\ulcorner x!\langle\texttt{ritz}\rangle.x!\langle\texttt{suite}\rangle.x?(rate).x!\langle\texttt{card}\rangle\ldots.\urcorner\rangle$$

A more interesting optimisation is given by `MClient` which at first may seem to disagree with the intended protocol:

$$\texttt{MClient} = \overline{a}(x).x \lhd \mathsf{move}.x!\langle\ulcorner x!\langle\texttt{ritz}\rangle.x!\langle\texttt{suite}\rangle.x!\langle\texttt{card}\rangle.x?(rtt).x?(rate)\ldots\urcorner\rangle$$

After the session is established, it eagerly sends its choice move, ignoring `rtt`, followed by a thunk that will continue the session; and another important point is that in the mobile code the output of the `card` happens before `rtt` and `rate` are received.

Even without subtyping, the typing of sessions in the HOπ-calculus poses delicate conditions [63]; in the present system, we can further verify that the optimisation of `MClient` does not violate communications safety (but the similar example in § 1, $\bar{s}!\langle\ulcorner\bar{s}?(x).\bar{s}!\langle x\rangle\urcorner\rangle.\bar{s}?(z_1).\bar{s}?(z_2).\mathbf{0}$, must be untypable): when values are received they are always of the expected type, conforming to a new *subtyping* relation given in the next section.

5.3 Higher-Order Asynchronous Subtyping

This section presents a theory of asynchronous session subtyping: reordered communications between two processes, in the presence of higher-order values and session mobility, can preserve the desired type-safety and progress invariants of the original protocol.

A permutation of two inputs or two outputs is not permissible because it violates type-safety. Suppose $P = s!\langle 2\rangle.s!\langle\mathtt{true}\rangle.s?(x).\mathbf{0}$ and $Q = \bar{s}?(y).\bar{s}?(z).\bar{s}!\langle y+2\rangle.\mathbf{0}$. These processes interact correctly. If we permute the outputs of P to get $P' = s!\langle\mathtt{true}\rangle.s!\langle 2\rangle.s?(x).\mathbf{0}$, then the parallel composition $(P' \mid Q)$ causes a type-error. Similarly, an alteration in the order of inputs may cause deadlock, losing progress in session s. For example, consider exchanging $s!\langle\mathtt{true}\rangle$ and $s?(z)$ in $P_1 = s!\langle\mathtt{true}\rangle.s?(z).\mathbf{0}$, and $Q_1 = \bar{s}?(y).\bar{s}!\langle 2\rangle.\mathbf{0}$.

The syntax of types remains the same as in Chapter 4, defined in Figure 4.6 on page 75. We begin with some preliminary notions. An occurrence of a type constructor *not* under a recursive prefix in a recursive type is called a *top-level action*. For example, $![U_1]$ and $?[U_2]$ in $![U_1].?[U_2].\mu\mathbf{t}.![U_3].\mathbf{t}$ are top-level, but $![U_3]$ in the same type is not.

Consider the following types:

$$\begin{aligned} S_1 &= ![U_1].?[U_2].\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t} \\ S_2 &= \mu\mathbf{t}.?[U_2].![U_1].\mathbf{t} \end{aligned}$$

Intuitively, we want to include S_1 in the subtypes of S_2, because in the infinite expansion of the types any action of S_1 can be matched to one in S_2. The first output $![U_1]$ of S_1 needs to be matched with a copy of the same output obtained after unrolling the recursion in S_2 once, resulting in:

$$S_2' = ?[U_2].![U_1].S_2$$

This unrolling is necessary because under the μ binder, every action has multiplicative effect, and by unrolling once we can obtain one of the possibly infinite instances of the action. For this strategy to succeed, we need to obtain the output $![U_1]$ in S_2' which is *guarded* under the input action $?[U_2]$. Then, the input action can be compared, and the remaining types checked, following a coinductive method similar to the one in Chapter 4.

To summarise, in asynchronous coinductive subtyping we need to formalise both the *unfolding* of a type and also the type *contexts* specifying the top-level actions that may guard an output (or selection).

We generalise the type unfolding function defined in [39] so that it can be applied to guarded types, yielding the following definition, based on [65]:

Definition 5.3.1 (*n*-time unfolding).

$\mathsf{unfold}^0(S) = S$ for all S $\qquad$ $\mathsf{unfold}^{1+n}(S) = \mathsf{unfold}^1(\mathsf{unfold}^n(S))$

$\mathsf{unfold}^1(![U].S) = ![U].\mathsf{unfold}^1(S)$ $\qquad$ $\mathsf{unfold}^1(\oplus[l_i\colon S_i]_{i\in I}) = \oplus[l_i\colon \mathsf{unfold}^1(S_i)]_{i\in I}$

$\mathsf{unfold}^1(?[U].S) = ?[U].\mathsf{unfold}^1(S)$ $\qquad$ $\mathsf{unfold}^1(\&[l_i\colon S_i]_{i\in I}) = \&[l_i\colon \mathsf{unfold}^1(S_i)]_{i\in I}$

$\mathsf{unfold}^1(\mathbf{t}) = \mathbf{t}$ $\qquad$ $\mathsf{unfold}^1(\mu\mathbf{t}.S) = S[\mu\mathbf{t}.S/\mathbf{t}]$ $\qquad$ $\mathsf{unfold}^1(\mathsf{end}) = \mathsf{end}$

For any recursive type S, $\mathsf{unfold}^n(S)$ is the result of inductively unfolding the top level recursion up to a fixed level of nesting. For example:

$$\mathsf{unfold}^1(?[U].\mu\mathbf{t}.![U'].\mathbf{t}) \quad = \quad ?[U].![U'].\mu\mathbf{t}.![U'].\mathbf{t}$$

$$\mathsf{unfold}^2(?[U].\mu\mathbf{t}.?[U].\mu\mathbf{t}'.![U'].\mathbf{t}') \quad =$$

$$\mathsf{unfold}^1(?[U].?[U].\mu\mathbf{t}'.![U'].\mathbf{t}') \quad =$$

$$?[U].?[U].![U'].\mu\mathbf{t}'.![U'].\mathbf{t}'$$

From the definition we have that $\mathsf{unfold}^1(\mathsf{unfold}^n(S)) = \mathsf{unfold}^n(\mathsf{unfold}^1(S))$, even though normally we apply from the outside. Also, since recursive types are not unfolded until they become guarded, but only n-times, $\mathsf{unfold}^n(S)$ terminates. Moreover, because our recursive types are contractive, there is no need to apply unfolding indefinitely to obtain a guarded type.

Then we proceed to define the contexts corresponding to a nested structure of top-level input actions (where branching is treated like input in the sense that a label is to be received). The rationale is that a supertype is less asynchronous than a subtype, hence may consist of input actions before any outputs that need to be checked first, based on the prefix of the subtype. Thus, the multi-hole *asynchronous contexts* are defined as follows:

Definition 5.3.2 (Asynchronous Contexts).

$$\mathcal{A} ::= \langle\cdot\rangle^{h\in H} \mid ?[U].\mathcal{A} \mid \&[l_i : \mathcal{A}_i]_{i\in I}$$

We write $\mathcal{A}\langle S_h\rangle^{h\in H}$ for the context $\mathcal{A}$ with holes indexed by $h \in H$, where each hole $\langle\cdot\rangle^{h\in H}$ is substituted with S_h. For example, taking $H = \{1,2\}$ and

$$\mathcal{A} = \&[l_1 : ?[U].\langle\cdot\rangle^{1\in H}, l_2 : \langle\cdot\rangle^{2\in H}]$$

we obtain

$$\mathcal{A}\langle ![U'].S_h\rangle^{h\in H} = \&[l_1 : ?[U].![U'].S_1, l_2 : ![U'].S_2]$$

We now introduce the main definition, *asynchronous communication subtyping*, which is a modification of the coinductive method of Definition 4.3.1:

Definition 5.3.3 (Asynchronous Subtyping). A relation $\Re \in \mathcal{T} \times \mathcal{T}$ is an asynchronous type simulation if $(T_1, T_2) \in \Re$ implies the following conditions:

1. If $T_1 = \diamond$, then $T_2 = \diamond$.

2. If $T_1 = \mathsf{unit}$, then $T_2 = \mathsf{unit}$.

3. If $T_1 = U_1 \to T_1'$, then $T_2 = U_2 \to T_2'$ or $T_2 = U_2 \multimap T_2'$ with $(U_2, U_1)^{\circledast} \in \Re$ and $(T_1', T_2')^{\circledast} \in \Re$.

4. If $T_1 = U_1 \multimap T_1'$, then $T_2 = U_2 \multimap T_2'$ with $(U_2, U_1)^{\circledast} \in \Re$ and $(T_1', T_2')^{\circledast} \in \Re$.

5. If $T_1 = \langle S_1\rangle$, then $T_2 = \langle S_2\rangle$ and $(S_1, S_2) \in \Re$ and $(S_2, S_1) \in \Re$.

6. If $T_1 = \mathsf{end}$, then for some n, $\mathsf{unfold}^n(T_2) = \mathsf{end}$.

7. If $T_1 = ![U_1].S_1$, then for some n, $\mathsf{unfold}^n(T_2) = \mathcal{A}\langle ![U_2].S_{2h}\rangle^{h\in H}$ with $(U_1, U_2)^{\circledast} \in \Re$ and $(S_1, \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \Re$.

8. If $T_1 = ?[U_1].S_1$, then for some n, $\mathsf{unfold}^n(T_2) = ?[U_2].S_2$, $(U_2, U_1)^{\circledast} \in \Re$ and $(S_1, S_2) \in \Re$.

9. If $T_1 = \oplus[l_i : S_{1i}]_{i\in I}$, then for some n, $\mathsf{unfold}^n(T_2) = \mathcal{A}\langle \oplus[l_j : S_{2jh}]_{j\in J_h}\rangle^{h\in H}$ and $\forall h \in H . I \subseteq J_h$ and $\forall i \in I.(S_{1i}, \mathcal{A}\langle S_{2ih}\rangle^{h\in H}) \in \Re$.

10. If $T_1 = \&[l_i : S_{1i}]_{i\in I}$, then for some n, $\mathsf{unfold}^n(T_2) = \&[l_j : S_{2j}]_{j\in J}$, $J \subseteq I$ and $\forall j \in J.(S_{1j}, S_{2j}) \in \Re$.

11. If $T_1 = \mu\mathbf{t}.S$, then $(\mathsf{unfold}^1(T_1), T_2) \in \Re$.

The coinductive subtyping $T_1 \leqslant_c T_2$ (read: T_1 is an *asynchronous subtype* of T_2) is defined when there exists a type simulation $\Re$ with $(T_1, T_2) \in \Re$. Formally, $\leqslant_c$ is the largest type simulation, defined as the union of all type simulations.

Most cases are similar to the ones in Definition 4.3.1, but in order to facilitate the asynchronous rules the unfolding of the supertype is performed at each case for some level n. We focus on the new rules: in (7), an output prefix of T_1 can be simulated when T_2 can be unfolded to obtain a type that has an output hidden under an asynchronous context $\mathcal{A}$, which by definition consists of only inputs and branchings. Therefore, U_1 is compared to U_2, the first available top-level output. As before, we adapt the variance based on the shape of U_i. Then the continuation S_1 of T_1 is compared with the asynchronous context $\mathcal{A}\langle S_{2h}\rangle^{h \in H}$ where the output(s) have been removed, since they were matched with the output prefix of T_1. For the input in (8), we do not use any context, since the input must appear as the first action after unfolding. No action can appear before the desired input at the supertype: if there is a branching (which is a form of input, with labels as values) it is not comparable, and if there is an output or selection then T_2 cannot be a supertype of the input-prefixed type T_1.

In (9), selection is defined similarly to output and a label appearing in T_1 must be included in the top level selections of the asynchronous context derived from T_2. Note that in the supertype, each hole in the context may use a different indexing set I_h, but the set I of the subtype is smaller than all these sets. Dually to selection, in (10), branching is defined like input and any labelled branch of (the unfolding of) T_2 must be supported in T_1. Finally (11) forces T_1 to be unfolded until it becomes a guarded type.

5.3.1 Some Examples of $\leqslant_c$

We show four small but representative examples which highlight key points of our subtyping relation. The first example shows that permuting outputs in advance of inputs in an infinite type preserves subtyping. The second example demonstrates that in some subtypings, a finite number of extra outputs can appear in the subtype, and dually, a finite number of extra inputs can appear in the supertype; this is acceptable when the total outputs remain infinite without losing type compatibility, and similarly for inputs. The third example demonstrates a case where n-level unfolding is required. The fourth example which is atypical exposes a class of subtypings that induce infinite simulation relations, due to asynchronous subtyping.

Three typical examples Consider the types given previously:

$$S_1 = ![U_1].?[U_2].\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$$

$$S_2 = \mu\mathbf{t}.?[U_2].![U_1].\mathbf{t}$$

It is easy to verify that $S_1 \leqslant_c S_2$ by checking that the following relation is a type simulation:

$$\begin{aligned}\mathfrak{R} = \{ & (S_1,S_2), (U_1,U_1), (?[U_2].\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}, ?[U_2].S_2),\\ & (U_2,U_2), (\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}, S_2) \}\end{aligned}$$

It is also straightforward to show that for the following types:

$$S_3 = ![U_1].\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$$

$$S_4 = ?[U_2].\mu\mathbf{t}.?[U_2].![U_1].\mathbf{t}$$

It holds that $S_3 \leqslant_c S_4$ using the following simulation:

$$\begin{aligned}\mathfrak{R} = \{ & (S_3,S_4), (U_1,U_1), (\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}, ?[U_2].S_4),\\ & (![U_1].?[U_2].\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}, ?[U_2].S_4), (?[U_2].\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}, ?[U_2].?[U_2].S_4),\\ & (U_2,U_2) \}\end{aligned}$$

Also easily we can demonstrate that for the following types:

$$S_5 = \mu\mathbf{t}.![U].?[U].\&[l_1 : \mathbf{t}, l_2 : \mathbf{t}]$$

$$S_6 = \mu\mathbf{t}_1.?[U].\mu\mathbf{t}_2.\&[l_1 :![U].\mathbf{t}_1, l_2 :![U].\mathbf{t}_1]$$

We have that $S_5 \leqslant_c S_6$ with the following simulation:

$$\begin{aligned}\mathfrak{R} = \{ & (S_5,S_6), (U,U), (\mathsf{unfold}^1(S_5),S_6), (\mathsf{unfold}^1(S_5),\mathsf{unfold}^2(S_6)),\\ & (?[U].\&[l_1 : S_5, l_2 : S_5], ?[U].\&[l_1 : S_6, l_2 : S_6]), (\&[l_1 : S_5, l_2 : S_5], \&[l_1 : S_6, l_2 : S_6])\}\end{aligned}$$

In fact, since as we prove in the next section $\leqslant_c$ is transitive, we can also find a simulation $\mathfrak{R}'$ such that $(\mu\mathbf{t}.![U_1].?[U].\&[l_1 : \mathbf{t}, l_2 : \mathbf{t}], \mu\mathbf{t}_1.?[U].\mu\mathbf{t}_2.\&[l_1 :![U_2].\mathbf{t}_1, l_2 :![U_3].\mathbf{t}_1]) \in \mathfrak{R}'$ whenever $(U_1,U_2)^{\circledast} \in \mathfrak{R}'$ and $(U_1,U_3)^{\circledast} \in \mathfrak{R}'$. For this the simulation will support the intermediate results $(\mu\mathbf{t}.![U_1].?[U].\&[l_1 : \mathbf{t}, l_2 : \mathbf{t}], \mu\mathbf{t}_1.?[U].\mu\mathbf{t}_2.\&[l_1 :![U_1].\mathbf{t}_1, l_2 :![U_1].\mathbf{t}_1]) \in \mathfrak{R}'$ and $(\mu\mathbf{t}_1.?[U].\mu\mathbf{t}_2.\&[l_1 : ![U_1].\mathbf{t}_1, l_2 :![U_1].\mathbf{t}_1], \mu\mathbf{t}_1.?[U].\mu\mathbf{t}_2.\&[l_1 :![U_2].\mathbf{t}_1, l_2 :![U_3].\mathbf{t}_1]) \in \mathfrak{R}'$.

A more controversial example Consider the types:

$$\begin{aligned} S_7 &= \mu\mathbf{t}.![U_1].\mathbf{t} \\ S_8 &= \mu\mathbf{t}.![U_1].?[U_2].\mathbf{t} \end{aligned}$$

Perhaps surprisingly, it holds that $S_7 \leqslant_c S_8$, as evidenced by the following simulation:

$$\begin{aligned} \mathfrak{R} = \{ & (U_1, U_1), \\ & (S_7, S_8), \\ & (![U_1].S_7, S_8), \\ & (S_7, ?[U_2].S_8), \\ & (![U_1].S_7, ?[U_2].S_8), \\ & (S_7, ?[U_2].?[U_2].S_8), \\ & (![U_1].S_7, ?[U_2].?[U_2].S_8), \\ & \quad\vdots \\ & (S_7, ?[U_2]^{\mathbb{N}}.S_8), \\ & (![U_1].S_7, ?[U_2]^{\mathbb{N}}.S_8)\} \\ = & \{(U_1, U_1)\} \\ \cup & \{(S_7, ?[U_2]^n.S_8),\ (![U_1].S_7, ?[U_2]^n.S_8) \mid n \in 0\ldots\mathbb{N}\} \end{aligned}$$

where $?[U_2]^n.S_8$ is the type S_8 prefixed with a sequence of n input actions $?[U_2]$. Effectively, the subtype is sending all the infinite outputs in advance, and never receives any values.

The above example seems slightly pathological: operationally if a process running a session with the subtype $\mu\mathbf{t}.![U_1].\mathbf{t}$ takes the place of one typed with the supertype $\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$, its buffer might grow indefinitely. To see why, assume that the process with session $\mu\mathbf{t}.![U_1].\mathbf{t}$ interacts, as expected, with a process that runs the session with type $\overline{S_8} = \mu\mathbf{t}.?[U_1].![U_2].\mathbf{t}$. Clearly, the values of type U_2 are received in the buffer but not in the program, therefore the buffer increases in size indefinitely, eventually causing a buffer overflow. However, type safety is not violated since no value of unexpected type is ever received within a term.

There are many similar examples, where the common denominator in all is the presence, within a recursion at the subtype, of a greater proportion of output actions (including selection) compared to the supertype. For instance, $\mu\mathbf{t}.![U_1].![U_1].?[U_2].\mathbf{t} \leqslant_c \mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$ also holds and can be shown with an infinite simulation relation.

5.3.2 The Relation $\leqslant_c$ is a Preorder

We conclude this section with the main theorem, stating that $\leqslant_c$ is a preorder. In inductively defined subtyping systems, commonly presented as a set of deduction rules, transitivity is a property by definition [35, 76]. In a coinductive setting, transitivity cannot be assumed, and not every simulation is guaranteed to contain the necessary hypotheses; however, we can prove that $\leqslant_c$ is transitive by careful construction of supporting simulations, containing the necessary components up to unfolding and context manipulation.

If $\leqslant_c$ was not transitive, there would not be type safety. The typical explanation is that, if there exists $U_1 \leqslant_c U_2$ and $U_2 \leqslant_c U_3$ such that $U_1 \nleqslant_c U_3$, then from two consecutive applications of subsumption (such as with rule (Sub) of the previous Chapter) we may provide a value of type U_1 when U_3 is expected, which is unsafe when $U_1 \nleqslant_c U_3$. For a detailed exposition to the issues arising from the use of coinductive definitions in subtyping, see Chapter 21 of [76].

The standard method of relational composition [39] is not enough for proving the transitivity of $\leqslant_c$. Moreover, the asynchronous coinductive subtyping in our previous work [65] does not work with higher-order and contravariant components in types. The difficulty is that, given $S_1 \leqslant_c S_2$ and $S_2 \leqslant_c S_3$, we need to find a subtyping relation that includes enough elements to justify $S_1 \leqslant_c S_3$ directly. However, due to the use of nested n-times unfolding with manipulation of asynchronous contexts, we only obtain information which cannot be straightforwardly combined with the hypotheses from $S_2 \leqslant_c S_3$.

Our objective is to discover how to obtain the missing elements, and to achieve it we gradually formalise a set of extensions on simulations, essentially monotonous functions from simulations to simulations, and then utilise them to prove the main result, Theorem 5.3.15, stating that $\leqslant_c$ is a preorder.

Overview of Proof Specifically, we perform the following steps:

- We prove as standard that unfolding S_1 or S_2 or both in $S_1 \leqslant_c S_2$ preserves the subtyping. We formalise the *unfolding extension* of a simulation to include such n-times unfoldings.
- We define a class of *single-step permutation contexts* representing an input/branching prefixed type. Then we formalise rules for moving an output/selection appearing within such a context (that is, immediately after the initial input/branching), to the position ahead of it. This represents the finest granularity of permutation since it is not transitive.
- The *contextual extension* of a simulation is defined, which is a supporting construction. It

is necessary in order to obtain the subtypings that arise when removing an output/selection from a single-step permutation context, thus changing its original structure.

- The *asynchronous extension* of degree n is defined by applying n consecutive single-step permutations on the subtypes in a simulation relation, and up to contexts $\mathcal{A}$ (that is, also deep within the structure of types). Both the contextual and the unfolding extensions are necessary to prove that this relation is also a simulation.

- Multi-step permutations that can extract an output/selection from deep within a context $\mathcal{A}$, placing it ahead of all actions (that is, prefixing $\mathcal{A}$), are shown to be included in the asynchronous extension of degree $\mathbb{N}$. This is effectively a proof that the transitive application of nested single-step permutations is included in the asynchronous extension.

- The *transitivity connection* of two simulations is then defined, utilising a composition of asynchronous extensions for the given simulations. The proof that the transitivity connection is a simulation implies that $\leqslant_c$ is transitive.

- The relation $\leqslant_c$ is shown to be a preorder: reflexivity is easy to obtain using straightforward techniques, and transitivity is proved directly by utilising the result for transitivity connections.

Lemma 5.3.4. *If* $S_1 \leqslant_c S_2$ *then* $\mathsf{unfold}^n(S_1) \leqslant_c S_2$.

Proof Let $\mathfrak{R}$ be a type simulation such that $S_1 \,\mathfrak{R}\, S_2$. Let

$$\mathcal{U}_1^n(\mathfrak{R}) = \bigcup_{i \in 1..n} \left\{ (\mathsf{unfold}^i(S_1'), S_2') \mid (S_1', S_2') \in \mathfrak{R} \right\} \cup \mathfrak{R}$$

Clearly $(\mathsf{unfold}^n(S_1), S_2) \in \mathcal{U}_1^n(\mathfrak{R})$, but is has to be shown that $\mathcal{U}_1^n(\mathfrak{R})$ is a type simulation. For this we need to demonstrate that for any $(T_1, T_2) \in \mathcal{U}_1^n(\mathfrak{R})$ the rules of simulation (Definition 5.3.3) hold. Since $\mathfrak{R} \subseteq \mathcal{U}_1^n(\mathfrak{R})$ and $\mathfrak{R}$ is a simulation, we only need to examine the cases for $(\mathsf{unfold}^i(S_1'), S_2') \in \mathcal{U}_1^n(\mathfrak{R}) \setminus \mathfrak{R}$, that is, for the new elements for which $(S_1', S_2') \in \mathfrak{R}$ holds by our construction of $\mathcal{U}_1^n(\mathfrak{R})$.

In the following we write $\leqslant_c^{(11)}$ to mean case (11) of Definition 5.3.3.

Case $\mathsf{unfold}^i(S_1') = \mathsf{end}$. Then $S_1' = \mu\mathbf{t}_1 \ldots \mu\mathbf{t}_z.\mathsf{end}$ for $0 \leq z \leq i$. We have, by assumption, that $(S_1', S_2') \in \mathfrak{R}$, therefore after applying $\leqslant_c^{(11)}$ z times we get $(\mathsf{end}, S_2') \in \mathfrak{R}$, and by the rules of simulation $\mathsf{unfold}^m(S_2') = \mathsf{end}$, for some m, as required.

Case $\mathsf{unfold}^i(S_1') = ![U_1].S_{10}$. Without loss of generality let $S_1' = \mu\mathbf{t}_1 \ldots \mu\mathbf{t}_z.![U_1].S_{10}'$ with $0 \le z \le i$. We have $(S_1', S_2') \in \Re$ and after z uses of $\leqslant_c^{(11)}$ we obtain $(\mathsf{unfold}^z(S_1'), S_2') \in \Re$ which can be written, based on the shape of S_1', as $(![U_1].S_{10}'', S_2') \in \Re$. The type S_{10}'' is derived from S_{10}' after the type variable substitutions induced by the z unfoldings on S_1'. By the rules of simulation, from $(![U_1].S_{10}'', S_2') \in \Re$ we obtain $\mathsf{unfold}^m(S_2') = \mathcal{A}\langle ![U_2].S_{2h}\rangle^{h\in H}$ and $(U_1, U_2)^{\circledast} \in \Re$ and $(S_{10}'', \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \Re$. From the shape of S_1' given previously we have that $\mathsf{unfold}^i(S_1') = ![U_1].\mathsf{unfold}^{i-z}(S_{10}'')$ and by our assumptions $S_{10} = \mathsf{unfold}^{i-z}(S_{10}'')$. By the construction of $\mathcal{U}_\mathbf{l}^n(\Re)$ and $(S_{10}'', \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \Re$ we have that $(\mathsf{unfold}^{i-z}(S_{10}''), \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \mathcal{U}_\mathbf{l}^n(\Re)$. Therefore we have $(S_{10}, \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \mathcal{U}_\mathbf{l}^n(\Re)$. From the definition of $\mathcal{U}_\mathbf{l}^n(\Re)$ which includes $\Re$ the above provide us with $(U_1, U_2)^{\circledast} \in \mathcal{U}_\mathbf{l}^n(\Re)$ and $(S_{10}, \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \mathcal{U}_\mathbf{l}^n(\Re)$, as required.

Case $\mathsf{unfold}^i(S_1') = \mu\mathbf{t}_1 \ldots \mu\mathbf{t}_z.S_{10}$. Then without loss of generality $S_1' = \mu\mathbf{t}_1' \ldots \mu\mathbf{t}_i'.\mu\mathbf{t}_1 \ldots \mu\mathbf{t}_z.S_{10}'$. The type S_{10} is derived from S_{10}' after the type variable substitutions induced by the i unfoldings on S_1'. Since $(S_1', S_2') \in \Re$, after i applications of $\leqslant_c^{(11)}$ we obtain $(\mathsf{unfold}^i(S_1'), S_2') \in \Re$ and hence $(\mathsf{unfold}^1(\mathsf{unfold}^i(S_1')), S_2') \in \Re$ which is the required result since $\Re \subseteq \mathcal{U}_\mathbf{l}^n(\Re)$.

Other cases are similar. □

Lemma 5.3.5. *If* $S_1 \leqslant_c S_2$ *then* $S_1 \leqslant_c \mathsf{unfold}^n(S_2)$.

Proof Let $\Re$ be a type simulation such that $S_1 \,\Re\, S_2$. Let

$$\mathcal{U}_\mathbf{r}^n(\Re) = \bigcup_{i\in 1..n} \{(S_1', \mathsf{unfold}^i(S_2')) \mid (S_1', S_2') \in \Re\} \cup \Re$$

The proof follows a pattern similar to the previous lemma. Clearly $(S_1, \mathsf{unfold}^n(S_2)) \in \mathcal{U}_\mathbf{r}^n(\Re)$, but is has to be shown that $\mathcal{U}_\mathbf{r}^n(\Re)$ is a type simulation. For this we need to demonstrate that for any $(T_1, T_2) \in \mathcal{U}_\mathbf{r}^n(\Re)$ the rules of simulation (Definition 5.3.3) hold. Since $\Re \subseteq \mathcal{U}_\mathbf{r}^n(\Re)$ and $\Re$ is a simulation, we only need to examine the cases for $(S_1, \mathsf{unfold}^m(S_2)) \in \mathcal{U}_\mathbf{r}^n(\Re) \setminus \Re$ with $m \le n$, that is, for the new elements for which $(S_1, S_2) \in \Re$ holds by the construction of $\mathcal{U}_\mathbf{r}^n(\Re)$.

Interesting cases are:

Case $S_1 = \mathsf{end}$. Then $(S_1, S_{20}) \in \Re$ and $S_2 = \mathsf{unfold}^m(S_{20})$ and $\mathsf{unfold}^z(S_{20}) = \mathsf{end}$. If $z \le m$ then $\mathsf{unfold}^m(S_{20}) = \mathsf{end}$ as required. If $z > m$ then $\mathsf{unfold}^{z-m}(S_2) = \mathsf{end}$ as required.

Case $S_1 = ![U_1].S_1'$. Then $(S_1, S_{20}) \in \Re$ and $S_2 = \mathsf{unfold}^m(S_{20})$ and $\mathsf{unfold}^z(S_{20}) = \mathcal{A}\langle ![U_2].S_{2h}\rangle^{h\in H}$ and $(U_1, U_2)^{\circledast} \in \Re$ and $(S_1', \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \Re$.

If $z \leq m$ then, using the definition of unfold

$$S_2 = \mathsf{unfold}^{m-z}(\mathcal{A}\langle ![U_2].S_{2h}\rangle^{h\in H}) = \mathcal{A}\langle ![U_2].\mathsf{unfold}^{m-z}(S_{2h})\rangle^{h\in H}$$

We have $(U_1, U_2)^{\circledast} \in \mathfrak{R}$, then we need $(S_1', \mathcal{A}\langle \mathsf{unfold}^{m-z}(S_{2h})\rangle^{h\in H}) \in \mathcal{U}_{\mathbf{r}}^{n}(\mathfrak{R})$. From $(S_1', \mathcal{A}\langle S_{2h}\rangle^{h\in H}) \in \mathfrak{R}$ we obtain $(S_1', \mathsf{unfold}^{m-z}(\mathcal{A}\langle S_{2h}\rangle^{h\in H})) \in \mathcal{U}_{\mathbf{r}}^{n}(\mathfrak{R})$, and then from the definition of unfold we obtain $\mathsf{unfold}^{m-z}(\mathcal{A}\langle S_{2h}\rangle^{h\in H}) = \mathcal{A}\langle \mathsf{unfold}^{m-z}(S_{2h})\rangle^{h\in H}$, as required. If $z > m$ then $\mathsf{unfold}^{z-m}(S_2) = \mathcal{A}\langle ![U_2].S_{2h}\rangle^{h\in H}$ and the supporting elements are in $\mathfrak{R}$, as required.

Other cases are similar. □

Definition 5.3.6 (Unfolding Extension). Given a simulation $\mathfrak{R}$, the *unfolding extension* of $\mathfrak{R}$ is defined as follows:

$$\mathcal{U}^{n}(\mathfrak{R}) = \mathcal{U}_{\mathbf{l}}^{n}(\mathfrak{R}) \cup \mathcal{U}_{\mathbf{r}}^{n}(\mathfrak{R})$$

Proposition 5.3.1. *If $\mathfrak{R} \subseteq \leqslant_c$ then $\mathcal{U}^{n}(\mathfrak{R}) \subseteq \leqslant_c$. That is, for any simulation $\mathfrak{R}$, the unfolding extension $\mathcal{U}^{n}(\mathfrak{R})$ is a type simulation.*

Proof Trivial as $\mathcal{U}^{n}(\mathfrak{R})$ is defined as the union of two simulations. □

We now define the single-step permutation transformations for top-level actions, which enable us to obtain more asynchronous subtypes, as this is needed further on when, given a simulation, we obtain more asynchronous simulations utilising single and multi-step permutations. There are two components, permutation contexts $\mathcal{C}$ and permutation rules $\gg$, defined as follows:

Definition 5.3.7 (Single-step Permutation).

Permutation Contexts

$$\mathcal{C} ::= ?[U].\langle\cdot\rangle^{h\in H} \mid \&[l_i : \langle\cdot\rangle^{h\in H}]_{i\in I}$$

Permutation Rules

$$\begin{array}{rcll} S & \gg & S & \\ \mathcal{C}\langle ![U].S_h\rangle^{h\in H} & \gg & ![U].\mathcal{C}\langle S_h\rangle^{h\in H} & \\ \mathcal{C}\langle \oplus[l_i : S_{ih}]_{i\in I_h}\rangle^{h\in H} & \gg & \oplus[l_i : \mathcal{C}\langle S_{ih}\rangle^{h\in H}]_{i\in I} & \forall h\in H\,.\, I \subseteq I_h \end{array}$$

Definition 5.3.8 (Contextual Extension). Given a simulation $\mathfrak{R}$, the *contextual extension* of $\mathfrak{R}$ is defined as follows:

$$\begin{aligned}\mathcal{CE}(\mathfrak{R}) &= \{(?[U_1].S_1, ?[U_2].S_2) \mid (U_2,U_1)^{\circledast} \in \mathfrak{R} \wedge (S_1,S_2) \in \mathfrak{R}\} \\ &\cup \left\{(\&[l_i : S_{1i}]_{i\in I}, \&[l_j : S_{2j}]_{j\in J}) \mid J \subseteq I \wedge \forall j \in J\,.\,(S_{1j},S_{2j}) \in \mathfrak{R}\right\} \\ &\cup \mathfrak{R}\end{aligned}$$

Lemma 5.3.9. *If $\mathfrak{R} \subseteq \leqslant_c$ then $\mathcal{CE}(\mathfrak{R}) \subseteq \leqslant_c$. That is, for any simulation $\mathfrak{R}$, the contextual extension $\mathcal{CE}(\mathfrak{R})$ is a type simulation.*

Proof Trivial since the generated pairs in $\mathcal{CE}(\mathfrak{R})$ are exactly those justified by the conditions in Definition 5.3.3, cases (8) and (10), with the required assumptions provided in $\mathfrak{R}$. We do not need to examine the $\mathfrak{R}$ subcomponent as it is a simulation by assumption. □

Next we define the *asynchronous extension* of a simulation, with degree n. The degree represents the number of single-step permutations, applied successively to all the components of the given simulation, up to asynchronous contexts $\mathcal{A}$.

Definition 5.3.10 (Asynchronous Extension). Given a simulation $\mathfrak{R}$, the *asynchronous extension* of $\mathfrak{R}$ with degree n is defined as follows:

$$\begin{aligned}\alpha^0(\mathfrak{R}) &= \mathfrak{R} \\ \alpha^n(\mathfrak{R}) &= \mathcal{CE}(\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R}))) \\ &\quad \cup \left\{(\mathcal{A}\langle S'_{1h}\rangle^{h\in H}, S_2) \mid (\mathcal{A}\langle S_{1h}\rangle^{h\in H}, S_2) \in \alpha^{n-1}(\mathfrak{R}) \wedge \forall h \in H\,.\,S_{1h} \gg S'_{1h}\right\} \quad (n \geq 1)\end{aligned}$$

Lemma 5.3.11. *If $\mathfrak{R} \subseteq \leqslant_c$ then $\alpha^n(\mathfrak{R}) \subseteq \leqslant_c$. That is, for any simulation $\mathfrak{R}$ and degree $n \in \mathbb{N}$, the asynchronous extension $\alpha^n(\mathfrak{R})$ is a type simulation.*

Proof We proceed by induction on the degree n. The base case of $n = 0$ holds because $\mathfrak{R}$ is a simulation by assumption. We then prove the inductive case for any $n \geq 1$.

By the inductive hypothesis $\alpha^{n-1}(\mathfrak{R}) \subseteq \leqslant_c$, then by Proposition 5.3.1 we have $\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R})) \subseteq \leqslant_c$, and by Lemma 5.3.9 we obtain $\mathcal{CE}(\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R}))) \subseteq \leqslant_c$. Therefore, it is not necessary to examine pairs in this subset of $\alpha^n(\mathfrak{R})$. Then, it remains to examine an arbitrary pair $(\mathcal{A}\langle S'_{1k}\rangle^{k\in K}, S_2) \in \alpha^n(\mathfrak{R})$ such that $(\mathcal{A}\langle S_{1k}\rangle^{k\in K}, S_2) \in \alpha^{n-1}(\mathfrak{R})$ with $\forall k \in K\,.\,S_{1k} \gg S'_{1k}$. We proceed by taking cases on the shape of the context $\mathcal{A}$.

Case $\mathcal{A} = \langle\cdot\rangle^{k\in K}$. Then let $S_1 = \mathcal{A}\langle S_{1k}\rangle^{k\in K}$, and $S'_1 = \mathcal{A}\langle S'_{1k}\rangle^{k\in K}$. We have $S_1 \gg S'_1$, and proceed by examination of the permutation applied.

Subcase $S_1 = S_1'$. Trivial.

Subcase $S_1 = \mathcal{C}\langle ![U].S_{1k}\rangle^{k\in K}$ *and* $S_1' = ![U].\mathcal{C}\langle S_{1k}\rangle^{k\in K}$. We proceed with cases on $\mathcal{C}$.

If $\mathcal{C} = ?[U_1].\langle\cdot\rangle^{k\in K}$, then

$$S_1 = \mathcal{C}\langle ![U].S_{1k}\rangle^{k\in K} = ?[U_1].![U].S_{1k}$$

and

$$S_1' = ![U].\mathcal{C}\langle S_{1k}\rangle^{k\in K} = ![U].?[U_1].S_{1k}$$

$$\begin{aligned}(S_1,S_2)\in\alpha^{n-1}(\mathfrak{R}) \quad &\Rightarrow \quad \mathsf{unfold}^n(S_2) = ?[U_2].S_2'\\ &\wedge \quad (U_2,U_1)^{\circledast}\in\alpha^{n-1}(\mathfrak{R})\\ &\wedge \quad (![U].S_{1k},S_2')\in\alpha^{n-1}(\mathfrak{R})\end{aligned}$$

$$\begin{aligned}(![U].S_{1k},S_2')\in\alpha^{n-1}(\mathfrak{R}) \quad &\Rightarrow \quad \mathsf{unfold}^m(S_2') = \mathcal{A}_1\langle ![U'].S_{2h}\rangle^{h\in H}\\ &\wedge \quad (U,U')^{\circledast}\in\alpha^{n-1}(\mathfrak{R})\\ &\wedge \quad (S_{1k},\mathcal{A}_1\langle S_{2h}\rangle^{h\in H})\in\alpha^{n-1}(\mathfrak{R})\end{aligned}$$

From the definition of n-times unfolding we obtain

$$\mathsf{unfold}^{n+m}(S_2) = ?[U_2].\mathcal{A}_1\langle ![U'].S_{2h}\rangle^{h\in H} = \mathcal{A}_2\langle ![U'].S_{2h}\rangle^{h\in H}$$

with

$$\mathcal{A}_2 = ?[U_2].\mathcal{A}_1$$

Now we proceed to justify the inclusion $(S_1',S_2)\in\alpha^n(\mathfrak{R})$. We have $(![U].?[U_1].S_{1k},S_2)\in\alpha^n(\mathfrak{R})$. Also $\mathsf{unfold}^{n+m}(S_2) = \mathcal{A}_2\langle ![U'].S_{2h}\rangle^{h\in H}$ with $(U,U')^{\circledast}\in\alpha^{n-1}(\mathfrak{R})(\subseteq\alpha^n(\mathfrak{R}))$. We then need to show that $(?[U_1].S_{1k},\mathcal{A}_2\langle S_{2h}\rangle^{h\in H})\in\alpha^n(\mathfrak{R})$ which can be written $(?[U_1].S_{1k},?[U_2].\mathcal{A}_1\langle S_{2h}\rangle^{h\in H})\in\alpha^n(\mathfrak{R})$. We have $(U_2,U_1)^{\circledast}\in\alpha^{n-1}(\mathfrak{R})$ and $(S_{1k},\mathcal{A}_1\langle S_{2h}\rangle^{h\in H})\in\alpha^{n-1}(\mathfrak{R})$, hence $(?[U_1].S_{1k},\mathcal{A}_2\langle S_{2h}\rangle^{h\in H})\in\mathcal{CE}(\alpha^{n-1}(\mathfrak{R}))\subseteq\alpha^n(\mathfrak{R})$ as required.

If $\mathcal{C} = \&[l_i:\langle\cdot\rangle^{k\in K}]_{i\in I}$, then

$$S_1 = \mathcal{C}\langle ![U].S_{1k}\rangle^{k\in K} = \&[l_i:![U].S_{1i}]_{i\in I}$$

and

$$S_1' = ![U].\mathcal{C}\langle S_{1k}\rangle^{k\in K} = ![U].\&[l_i : S_{1i}]_{i\in I}$$

$$\begin{array}{rcl}
(S_1,S_2) \in \alpha^{n-1}(\mathfrak{R}) & \Rightarrow & \mathsf{unfold}^n(S_2) = \&[l_j : S_{2j}]_{j\in J} \\
& \wedge & J \subseteq I \\
& \wedge & \forall j \in J\,.\,(![U].S_{1j}, S_{2j}) \in \alpha^{n-1}(\mathfrak{R}) \\
& \Rightarrow & \forall j \in J\,.\,\mathsf{unfold}^{m_j}(S_{2j}) = \mathcal{A}_j\langle ![U'].S'_{2jh}\rangle^{h\in H_j} \\
& \wedge & (U,U')^{\circledast} \in \alpha^{n-1}(\mathfrak{R}) \\
& \wedge & \forall j \in J\,.\,(S_{1j}, \mathcal{A}_j\langle S'_{2jh}\rangle^{h\in H_j}) \in \alpha^{n-1}(\mathfrak{R})
\end{array}$$

Let $m_{\mathsf{max}} = max_{j\in J}(m_j)$. Then from the unfolding construction of $\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R}))$ we obtain

$$\forall j \in J\,.\,(S_{1j}, \mathsf{unfold}^{m_{\mathsf{max}}-m_j}(\mathcal{A}_j\langle S'_{2jh}\rangle^{h\in H_j})) \in \mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R}))$$

From the definition of n-times unfolding we obtain

$$\begin{array}{rcl}
\mathsf{unfold}^{n+m_{\mathsf{max}}}(S_2) & = & \&[l_j : \mathsf{unfold}^{m_{\mathsf{max}}-m_j}(\mathcal{A}_j\langle ![U'].S'_{2jh}\rangle^{h\in H_j})]_{j\in J} \\
& = & \mathcal{A}'\langle ![U'].S'_{2jh}\rangle^{h\in H}
\end{array}$$

with

$$\mathcal{A}' = \&[l_j : \mathsf{unfold}^{m_{\mathsf{max}}-m_j}(\mathcal{A}_j)]_{j\in J} \qquad \text{and} \qquad H = \uplus_{j\in J}(H_j)$$

Now we proceed to justify the inclusion $(![U].\&[l_i : S_{1i}]_{i\in I}, S_2) \in \alpha^n(\mathfrak{R})$. We have $\mathsf{unfold}^{n+m_{\mathsf{max}}}(S_2) = \mathcal{A}'\langle ![U'].S'_{2jh}\rangle^{h\in H}$, and $(U,U')^{\circledast} \in \alpha^{n-1}(\mathfrak{R})$. We then need to show that $(\&[l_i : S_{1i}]_{i\in I}, \mathcal{A}'\langle S'_{2jh}\rangle^{h\in H}) \in \alpha^n(\mathfrak{R})$. Since $J \subseteq I$ and $\forall j \in J\,.\,(S_{1j}, \mathsf{unfold}^{m_{\mathsf{max}}-m_j}(\mathcal{A}_j\langle S'_{2jh}\rangle^{h\in H})) \in \mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R}))$, we have that $(\&[l_i : S_{1i}]_{i\in I}, \&[l_j : \mathsf{unfold}^{m_{\mathsf{max}}-m_j}(\mathcal{A}_j\langle S'_{2jh}\rangle^{h\in H})]_{j\in J}) \in \mathcal{CE}(\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R})))$, as required.

Subcase $S_1 = \mathcal{C}\langle \oplus[l_i : S_{1ik}]_{i\in I_k}\rangle^{k\in K}$ *and* $S_1' = \oplus[l_i : \mathcal{C}\langle S_{1ik}\rangle^{k\in K}]_{i\in I}$ *with* $\forall k \in K\,.\,I \subseteq I_k$. We proceed with cases on $\mathcal{C}$.

If $\mathcal{C} = ?[U_1].\langle\cdot\rangle^{k\in K}$, then

$$S_1 = \mathcal{C}\langle \oplus[l_i : S_{1ik}]_{i\in I_k}\rangle^{k\in K} = ?[U_1].\oplus[l_i : S_{1i}]_{i\in I}$$

and

$$S_1' = \oplus[l_i : \mathcal{C}\langle S_{1ik}\rangle^{k\in K}]_{i\in I} = \oplus[l_i : ?[U_1].S_{1i}]_{i\in I}$$

$$\begin{array}{rcl}(S_1,S_2)\in\alpha^{n-1}(\mathfrak{R}) & \Rightarrow & \mathsf{unfold}^n(S_2) = ?[U_2].S_2' \\ & \wedge & (U_2,U_1)^{\circledast}\in\alpha^{n-1}(\mathfrak{R}) \\ & \wedge & (\oplus[l_i : S_{1i}]_{i\in I}, S_2')\in\alpha^{n-1}(\mathfrak{R})\end{array}$$

$$\begin{array}{rcl}(\oplus[l_i : S_{1i}]_{i\in I}, S_2')\in\alpha^{n-1}(\mathfrak{R}) & \Rightarrow & \mathsf{unfold}^m(S_2') = \mathcal{A}_1\langle\oplus[l_j : S_{2jh}]_{j\in J_h}\rangle^{h\in H} \\ & \wedge & \forall h\in H\,.\,I\subseteq J_h \\ & \wedge & \forall i\in I\,.\,(S_{1i},\mathcal{A}_1\langle S_{2ih}\rangle^{h\in H})\in\alpha^{n-1}(\mathfrak{R})\end{array}$$

From the definition of n-times unfolding we obtain

$$\mathsf{unfold}^{n+m}(S_2) = ?[U_2].\mathcal{A}_1\langle\oplus[l_j : S_{2jh}]_{j\in J_h}\rangle^{h\in H} = \mathcal{A}_2\langle\oplus[l_j : S_{2jh}]_{j\in J_h}\rangle^{h\in H}$$

with

$$\mathcal{A}_2 = ?[U_2].\mathcal{A}_1$$

Now we proceed to justify the inclusion $(S_1', S_2)\in\alpha^n(\mathfrak{R})$. We have $(\oplus[l_i : ?[U_1].S_{1i}]_{i\in I}, S_2)\in\alpha^n(\mathfrak{R})$. Also $\mathsf{unfold}^{n+m}(S_2) = \mathcal{A}_2\langle\oplus[l_j : S_{2jh}]_{j\in J_h}\rangle^{h\in H}$ with $\forall h\in H\,.\,I\subseteq J_h$. We then need to show that $\forall i\in I\,.\,(?[U_1].S_{1i},\mathcal{A}_2\langle S_{2ih}\rangle^{h\in H})\in\alpha^n(\mathfrak{R})$ which can be written $(?[U_1].S_{1i}, ?[U_2].\mathcal{A}_1\langle S_{2ih}\rangle^{h\in H})\in\alpha^n(\mathfrak{R})$. We have $(U_2,U_1)^{\circledast}\in\alpha^{n-1}(\mathfrak{R})$ and $(S_{1i},\mathcal{A}_1\langle S_{2ih}\rangle^{h\in H})\in\alpha^{n-1}(\mathfrak{R})$, hence $(?[U_1].S_{1i},\mathcal{A}_2\langle S_{2ih}\rangle^{h\in H})\in\mathcal{CE}(\alpha^{n-1}(\mathfrak{R}))$ as required.

If $\mathcal{C} = \&[l_i : \langle\cdot\rangle^{k\in K}]_{i\in I}$, then

$$S_1 = \mathcal{C}\langle\oplus[l_i : S_{1ik}]_{i\in I_k}\rangle^{k\in K} = \&[l'_j : \oplus[l_i : S_{1ij}]_{i\in I_j}]_{j\in J}$$

and

$$S_1' = \oplus[l_i : \mathcal{C}\langle S_{1ik}\rangle^{k\in K}]_{i\in I} = \oplus[l_i : \&[l'_j : S_{1ij}]_{j\in J}]_{i\in I} \qquad \forall j\in J\,.\,I\subseteq I_j$$

$$\begin{array}{rcl}(S_1,S_2)\in\alpha^{n-1}(\mathfrak{R}) & \Rightarrow & \mathsf{unfold}^n(S_2) = \&[l'_z : S_{2z}]_{z\in Z} \\ & \wedge & Z\subseteq J \\ & \wedge & \forall z\in Z\,.\,(\oplus[l_i : S_{1iz}]_{i\in I_z}, S_{2z})\in\alpha^{n-1}(\mathfrak{R}) \\ & \Rightarrow & \forall z\in z\,.\,\mathsf{unfold}^{m_z}(S_{2z}) = \mathcal{A}_z\langle\oplus[l_j : S_{2zjh}]_{j\in J_h}\rangle^{h\in H_z} \\ & \wedge & \forall z\in z\,.\,\forall h\in H_z\,.\,I_z\subseteq J_h \\ & \wedge & \forall z\in z\,.\,\forall i\in I_z\,.\,(S_{1iz},\mathcal{A}_z\langle S_{2zih}\rangle^{h\in H_z})\in\alpha^{n-1}(\mathfrak{R})\end{array}$$

Let $m_{\mathsf{max}} = max_{z\in Z}(m_z)$. Then from the unfolding construction of $\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R}))$ we obtain

$$\forall z\in Z\,.\,\forall i\in I_z\,.\,(S_{1iz},\mathsf{unfold}^{m_{\mathsf{max}}-m_z}(\mathcal{A}_z\langle S_{2zih}\rangle^{h\in H_z}))\in\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R}))$$

From the definition of n-times unfolding we obtain

$$\begin{aligned}\mathsf{unfold}^{n+m_{\mathsf{max}}}(S_2) &= \&[l'_z : \mathcal{A}_z\langle\oplus[l_j : S_{2zjh}]_{j\in J_h}\rangle^{h\in H_z}]_{z\in Z} \\ &= \mathcal{A}'\langle\oplus[l_j : S_{2zjh}]_{j\in J_h}\rangle^{h\in H}\end{aligned}$$

with

$$\mathcal{A}' = \&[l'_z : \mathsf{unfold}^{m_{\mathsf{max}}-m_z}(\mathcal{A}_z)]_{z\in Z} \qquad \text{and} \qquad H = \uplus_{z\in Z}(H_z)$$

Now we proceed to justify the inclusion $(\oplus[l_i : \&[l'_j : S_{1ij}]_{j\in J}]_{i\in I}, S_2) \in \alpha^n(\mathfrak{R})$. We have $\mathsf{unfold}^{n+m_{\mathsf{max}}}(S_2) = \mathcal{A}'\langle\oplus[l_j : S_{2zjh}]_{j\in J_h}\rangle^{h\in H}$ and from $I \subseteq I_{z\in Z\subseteq J} \subseteq J_{h\in H_z}$ we obtain $\forall h \in H\,.\, I \subseteq J_h$. We then need to show that $\forall i \in I\,.\,(\&[l'_j : S_{1ij}]_{j\in J}, \&[l'_z : \mathsf{unfold}^{n+m_{\mathsf{max}}}(\mathcal{A}_z\langle S_{2zih}\rangle^{h\in H_z})]_{z\in Z}) \in \alpha^n(\mathfrak{R})$. These pairs are in $\mathcal{CE}(\mathcal{U}^{\mathbb{N}}(\alpha^{n-1}(\mathfrak{R})))$ by construction, as required.

Case $\mathcal{A} = ?[U].\mathcal{A}'$. From the shape of $\mathcal{A}$ we have $(?[U].\mathcal{A}'\langle S_{1k}\rangle^{k\in K}, S_2) \in \alpha^{n-1}(\mathfrak{R})$. By the rules of simulation, $\mathsf{unfold}^m(S_2) = ?[U'].S'_2$ and $(U', U)^{\circledast} \in \alpha^{n-1}(\mathfrak{R})$ and $(\mathcal{A}'\langle S_{1k}\rangle^{k\in K}, S'_2) \in \alpha^{n-1}(\mathfrak{R})$. By the construction of $\alpha^n(\mathfrak{R})$ we have $(\mathcal{A}'\langle S'_{1k}\rangle^{k\in K}, S'_2) \in \alpha^n(\mathfrak{R})$. It is now straightforward to show that $(?[U].\mathcal{A}'\langle S'_{1k}\rangle^{k\in K}, S_2)$ is justified by the rules of simulation and the above hypotheses.

Case $\mathcal{A} = \&[l_i : \mathcal{A}_i]_{i\in I}$. From the shape of $\mathcal{A}$ we have $(\&[l_i : \mathcal{A}_i\langle S_{1k}\rangle^{k\in K}]_{i\in I}, S_2) \in \alpha^{n-1}(\mathfrak{R})$. By the rules of simulation, $\mathsf{unfold}^m(S_2) = \&[l_j : S_{2j}]_{j\in J}$ and $J \subseteq I$ and $\forall j \in J\,.\,(\mathcal{A}_j\langle S_{1k}\rangle^{k\in K}, S_{2j}) \in \alpha^{n-1}(\mathfrak{R})$. By the construction of $\alpha^n(\mathfrak{R})$ we have $\forall j \in J\,.\,(\mathcal{A}_j\langle S'_{1k}\rangle^{k\in K}, S_{2j}) \in \alpha^n(\mathfrak{R})$. As before it is now trivial to justify $(\&[l_i : \mathcal{A}_i\langle S'_{1k}\rangle^{k\in K}]_{i\in I}, S_2) \in \alpha^n(\mathfrak{R})$. □

Corollary 5.3.12 (Multi-step Permutation).

1. *If* $(\mathcal{A}\langle ![U_1].S_{1k}\rangle^{k\in K}, S_2) \in \alpha^{\mathbb{N}}(\mathfrak{R})$ *then* $(![U_1].\mathcal{A}\langle S_{1k}\rangle^{k\in K}, S_2) \in \alpha^{\mathbb{N}}(\mathfrak{R})$

2. *If* $(\mathcal{A}\langle\oplus[l_i : S_{1ih}]_{i\in I_h}\rangle^{h\in H}, S_2) \in \alpha^{\mathbb{N}}(\mathfrak{R})$ *and* $\forall h \in H\,.\, I \subseteq I_h$, *then* $(\oplus[l_i : \mathcal{A}\langle S_{1ih}\rangle^{h\in H}]_{i\in I}, S_2) \in \alpha^{\mathbb{N}}(\mathfrak{R})$.

Proof Every context $\mathcal{A}$ can be written as a (possibly empty) nested structure of $\mathcal{C}$ contexts, such that $\mathcal{A} = \mathcal{C}\langle \mathcal{C}_h\langle \mathcal{C}_{hk}\langle\ldots\rangle^{\cdots}\rangle^{h\in H}\rangle^{k\in K}$. Every level of asynchronous permutation in $\alpha^{\mathbb{N}}(\mathfrak{R})$ generates pairs by applying a transformation on the innermost $\mathcal{C}$ contexts of all matching types; in this way it reduces the depth of the innermost contexts for the generated type pairs, which are matched at the next level. At every level, the penultimate contexts become last. By induction on the maximum depth of the nested $\mathcal{C}$-context representation of any $\mathcal{A}$, we can obtain the result formally. □

Proposition 5.3.2. *If* $(S_1, S_2) \in \alpha^{\mathbb{N}}(\mathfrak{R})$ *then* $(\mathsf{unfold}^n(S_1), S_2) \in \alpha^{\mathbb{N}}(\mathfrak{R})$

Proof Easy to obtain: since $\gg$ allows the identity permutation, then for all m, $\alpha^m(\mathfrak{R})$ will include $\mathcal{CE}(\mathcal{U}^{\mathbb{N}}(\alpha^{m-1}(\mathfrak{R}))) \supseteq \mathcal{U}^{\mathbb{N}}(\alpha^{m-1}(\mathfrak{R}))$ even when there are no more effective permutations to apply on any type, and up to all contexts. Suppose $(S_1, S_2) \in \alpha^z(\mathfrak{R})$, take $m = z+n+1$, and we obtain the result $(\mathsf{unfold}^n(S_1), S_2) \in \alpha^m(\mathfrak{R})$. $\square$

Transitivity Connection

Next is the main definition of this section, the *transitivity connection* of two relations. It is defined as the relational composition (taking the union of both directions, needed due to the presence of contravariant components) of the asynchronous extensions of the given simulations, respectively. We then prove that the transitivity connection (of simulations) is a simulation, which is, effectively, a proof of the transitivity of $\leqslant_c$.

Definition 5.3.13 (Transitivity Connection). For type simulations $\mathfrak{R}_1$ and $\mathfrak{R}_2$, the *transitivity connection* $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ is defined as follows:

$$\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2) = \alpha^{\mathbb{N}}(\mathfrak{R}_1) \cdot \alpha^{\mathbb{N}}(\mathfrak{R}_2) \cup \alpha^{\mathbb{N}}(\mathfrak{R}_2) \cdot \alpha^{\mathbb{N}}(\mathfrak{R}_1)$$

Lemma 5.3.14. *If $\mathfrak{R}_{i\in\{1,2\}} \subseteq \leqslant_c$ then $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2) \subseteq \leqslant_c$. That is, for any two simulations $\mathfrak{R}_1$ and $\mathfrak{R}_2$, the transitivity connection $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ is a type simulation.*

Proof We examine an arbitrary $(T_1, T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1) \cdot \alpha^{\mathbb{N}}(\mathfrak{R}_2) \subseteq \mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$, taking cases on the shape of T_1. The remaining cases, for membership in $\alpha^{\mathbb{N}}(\mathfrak{R}_2) \cdot \alpha^{\mathbb{N}}(\mathfrak{R}_1)$, are symmetric.

Case $T_1 = ![U_1].S_1$. Then $(T_1, T_2) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1)$ and $(T_2, T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2)$.

$$\begin{array}{rcl}
(![U_1].S_1, T_2) \in \alpha^{\mathbb{N}}(\Re_1) & \Rightarrow & \mathsf{unfold}^n(T_2) = \mathcal{A}_1\langle ![U_2].S_{2h}\rangle^{h\in H} \\
& \wedge & (U_1, U_2)^{\circledast} \in \alpha^{\mathbb{N}}(\Re_1) \\
& \wedge & (S_1, \mathcal{A}_1\langle S_{2h}\rangle^{h\in H}) \in \alpha^{\mathbb{N}}(\Re_1) \\[1em]
(T_2, T_3) \in \alpha^{\mathbb{N}}(\Re_2) & \Rightarrow & (\mathsf{unfold}^n(T_2), T_3) \in \alpha^{\mathbb{N}}(\Re_2) \\
& \Leftrightarrow & (\mathcal{A}_1\langle ![U_2].S_{2h}\rangle^{h\in H}, T_3) \in \alpha^{\mathbb{N}}(\Re_2) \\[1em]
\text{Corollary 5.3.12} & \Rightarrow & (![U_2].\mathcal{A}_1\langle S_{2h}\rangle^{h\in H}, T_3) \in \alpha^{\mathbb{N}}(\Re_2) \\
& \Rightarrow & \mathsf{unfold}^m(T_3) = \mathcal{A}_2\langle ![U_3].S_{3k}\rangle^{k\in K} \\
& \wedge & (U_2, U_3)^{\circledast} \in \alpha^{\mathbb{N}}(\Re_2) \\
& \wedge & (\mathcal{A}_1\langle S_{2h}\rangle^{h\in H}, \mathcal{A}_2\langle S_{3k}\rangle^{k\in K}) \in \alpha^{\mathbb{N}}(\Re_2) \\[1em]
U_{i\in\{1,2\}} = S'_i & \Rightarrow & (U_1, U_3)^{\circledast} = (U_3, U_1) \in \alpha^{\mathbb{N}}(\Re_2)\cdot\alpha^{\mathbb{N}}(\Re_1) \\
& \wedge & (S_1, \mathcal{A}_2\langle S_{3k}\rangle^{k\in K}) \in \alpha^{\mathbb{N}}(\Re_1)\cdot\alpha^{\mathbb{N}}(\Re_2) \\[1em]
U_{i\in\{1,2\}} \neq S & \Rightarrow & (U_1, U_3)^{\circledast} = (U_1, U_3) \in \alpha^{\mathbb{N}}(\Re_1)\cdot\alpha^{\mathbb{N}}(\Re_2) \\
& \wedge & (S_1, \mathcal{A}_2\langle S_{3k}\rangle^{k\in K}) \in \alpha^{\mathbb{N}}(\Re_1)\cdot\alpha^{\mathbb{N}}(\Re_2)
\end{array}$$

Hence (T_1, T_3) is justified in $\mathbf{trc}(\Re_1, \Re_2)$.

Case $T_1 = ?[U_1].S_1$. Then $(T_1, T_2) \in \alpha^{\mathbb{N}}(\Re_1)$ and $(T_2, T_3) \in \alpha^{\mathbb{N}}(\Re_2)$.

$$\begin{array}{rcl}
(?[U_1].S_1, T_2) \in \alpha^{\mathbb{N}}(\Re_1) & \Rightarrow & \mathsf{unfold}^n(T_2) = ?[U_2].S_2 \\
& \wedge & (U_2, U_1)^{\circledast} \in \alpha^{\mathbb{N}}(\Re_1) \\
& \wedge & (S_1, S_2) \in \alpha^{\mathbb{N}}(\Re_1) \\[1em]
(T_2, T_3) \in \alpha^{\mathbb{N}}(\Re_2) & \Rightarrow & (\mathsf{unfold}^n(T_2), T_3) \in \alpha^{\mathbb{N}}(\Re_2) \\
& \Leftrightarrow & (?[U_2].S_2, T_3) \in \alpha^{\mathbb{N}}(\Re_2) \\
& \Rightarrow & \mathsf{unfold}^m(T_3) = ?[U_3].S_3 \\
& \wedge & (U_3, U_2)^{\circledast} \in \alpha^{\mathbb{N}}(\Re_2) \\
& \wedge & (S_2, S_3) \in \alpha^{\mathbb{N}}(\Re_2) \\[1em]
U_{i\in\{1,2\}} = S'_i & \Rightarrow & (U_3, U_1)^{\circledast} = (U_1, U_3) \in \alpha^{\mathbb{N}}(\Re_1)\cdot\alpha^{\mathbb{N}}(\Re_2) \\
& \wedge & (S_1, S_3) \in \alpha^{\mathbb{N}}(\Re_1)\cdot\alpha^{\mathbb{N}}(\Re_2) \\[1em]
U_{i\in\{1,2\}} \neq S & \Rightarrow & (U_3, U_1)^{\circledast} = (U_3, U_1) \in \alpha^{\mathbb{N}}(\Re_2)\cdot\alpha^{\mathbb{N}}(\Re_1) \\
& \wedge & (S_1, S_3) \in \alpha^{\mathbb{N}}(\Re_1)\cdot\alpha^{\mathbb{N}}(\Re_2)
\end{array}$$

Hence, as before, (T_1,T_3) is justified in $\mathbf{trc}(\mathfrak{R}_1,\mathfrak{R}_2)$.

Case $T_1 = \oplus[l_i : S_{1i}]_{i\in I}$. Then $(T_1,T_2) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1)$ and $(T_2,T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2)$.

$$\begin{array}{rcl}
(\oplus[l_i : S_{1i}]_{i\in I}, T_2) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1) & \Rightarrow & \mathsf{unfold}^n(T_2) = \mathcal{A}_1\langle \oplus[l_j : S_{2jh}]_{j\in J_h}\rangle^{h\in H} \\
& \wedge & \forall h \in H \,.\, I \subseteq J_h \\
& \wedge & \forall i \in I \,.\, (S_{1i}, \mathcal{A}_1\langle S_{2ih}\rangle^{h\in H}) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1) \\
\\
(T_2,T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2) & \Rightarrow & (\mathsf{unfold}^n(T_2), T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2) \\
& \Leftrightarrow & (\mathcal{A}_1\langle \oplus[l_j : S_{2jh}]_{j\in J_h}\rangle^{h\in H}, T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2) \\
\\
\text{Corollary 5.3.12 with } I \subseteq J_{h\in H} & \Rightarrow & (\oplus[l_i : \mathcal{A}_1\langle S_{2ih}\rangle^{h\in H}]_{i\in I}, T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2) \\
& \Rightarrow & \mathsf{unfold}^m(T_3) = \mathcal{A}_2\langle \oplus[l_z : S_{2zk}]_{z\in Z_k}\rangle^{k\in K} \\
& \wedge & \forall k \in K \,.\, I \subseteq Z_k \\
& \wedge & \forall i \in I \,.\, (\mathcal{A}_1\langle S_{2ih}\rangle^{h\in H}, \mathcal{A}_2\langle S_{3ik}\rangle^{k\in K}) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2) \\
& \Rightarrow & \forall i \in I \,.\, (S_{1i}, \mathcal{A}_2\langle S_{3ik}\rangle^{k\in K}) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1)\cdot\alpha^{\mathbb{N}}(\mathfrak{R}_2)
\end{array}$$

Hence (T_1,T_3) is justified in $\mathbf{trc}(\mathfrak{R}_1,\mathfrak{R}_2)$.

Case $T_1 = \mu\mathbf{t}.S_1$. Then $(T_1,T_2) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1)$ and $(T_2,T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_2)$.

$$\begin{array}{rcl}
(\mu\mathbf{t}.S_1, T_2) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1) & \Rightarrow & (\mathsf{unfold}^1(\mu\mathbf{t}.S_1), T_2) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1) \\
& \Rightarrow & (\mathsf{unfold}^1(\mu\mathbf{t}.S_1), T_3) \in \alpha^{\mathbb{N}}(\mathfrak{R}_1)\cdot\alpha^{\mathbb{N}}(\mathfrak{R}_2)
\end{array}$$

Thus, (T_1,T_3) is justified in $\mathbf{trc}(\mathfrak{R}_1,\mathfrak{R}_2)$.

Other cases are similar, and in fact simpler, because they make no use of asynchronous contexts and permutations. □

Theorem 5.3.15 (**$\leqslant_c$ is a Preorder**). *The relation $\leqslant_c$ is reflexive and transitive.*

Proof For reflexivity it is easy to prove that $\{(T,T) \mid T \in \mathcal{T}\} \subseteq \leqslant_c$. For transitivity, we have that whenever $(T_1,T_2) \in \mathfrak{R}_1$ and $(T_2,T_3) \in \mathfrak{R}_2$, then $(T_1,T_3) \in \mathbf{trc}(\mathfrak{R}_1,\mathfrak{R}_2)$, and $\mathbf{trc}(\mathfrak{R}_1,\mathfrak{R}_2) \subseteq \leqslant_c$ by Lemma 5.3.14. □

5.4 Asynchronous Higher-Order Session Typing

The typing system extends the one for the language of Chapter 4, replacing a few rules with more general versions; for the basic system see Figures 4.8 and 4.9, starting on page 81. New formulations are needed for the integration of typing at the level of session queues, and for ensuring that the asynchronous calculus is sound.

Queue Types Due to the presence of labels in session queues, we need to extend the types to facilitate all buffer components, as follows:

$$\tau ::= U \mid l$$

Therefore, every label induces a singleton type identified with the label value.

Session Remainder Type soundness is established by also typing the queues created during the execution of a well-typed initial program. We track the movement of linear functions and channels to and from the queue to ensure that linearity is preserved, and we check that endpoints continue to have dual types up to asynchronous subtyping after each use. To analyse the intermediate steps precisely, we utilise a *session remainder* $S - \vec{\tau} = S'$ which subtracts the vector $\vec{\tau}$ of the queue types of the values stored in a queue from the complete session type S of the queue, obtaining a remaining session S'. The rules are formalised below: When S' is end, then the session has been completed; otherwise it is not closed yet.

(Empty)
$$\frac{}{S - \varepsilon = S}$$

(Get)
$$\frac{S - \vec{\tau} = S'}{?[U].S - U\vec{\tau} = S'}$$

(Put)
$$\frac{S - \vec{\tau} = S'}{![U].S - \vec{\tau} = ![U].S'}$$

(Branch)
$$\frac{S_k - \vec{\tau} = S' \qquad k \in I}{\&[l_i : S_i]_{i \in I} - l_k\vec{\tau} = S'}$$

(Select)
$$\frac{S_i - \vec{\tau} = S'_i \qquad \forall i \in I}{\oplus[l_i : S_i]_{i \in I} - \vec{\tau} = \oplus[l_i : S'_i]_{i \in I}}$$

Figure 5.4: Session Remainder

(Empty) is a base rule. (Get) takes an input prefixed session type $?[U].S$ and subtracts the type U at the head of the queue, then returns the remainder S' of the rest of the session S minus the tail $\vec{\tau}$ of the queue type. (Put) disregards the output action type of the session and calculates the remainder S' of $S - \vec{\tau}$, which is returned prefixed with the original output giving $![U].\vec{\tau}$. Therefore the output is not consumed. (Branch) is similar with (Get), but it only records the remainder of the k-th branch with respect to a stored label l_k. Dually, (Select) records the remainder of all selection paths.

Typing System for Terms with Session Queues We first extend the session environment as follows:

$$\Delta ::= \Sigma \mid \Delta, s : \vec{\tau} \mid \Delta, s : (S, \vec{\tau})$$

The typing judgement is also extended with

$$\Gamma; \Lambda; \Sigma \vdash l : l$$

which is used for typing any labels appearing in a session queue. Δ contains usage information for queues $(s : \vec{\tau})$ in a term, so that the cumulative result can be compared with the expected session type; for this we use the pairing $(s : (S, \vec{\tau}))$ that combines the usage of a channel and the sequence of types already on its queue. We identify $(S, \vec{\tau})$ and $(\vec{\tau}, S)$.

We define a composition operation $\odot$ on Δ-environments, used to obtain the paired usages for channels and queues:

$$\Delta_1 \odot \Delta_2 = \{s : (\Delta_1(s), \Delta_2(s)) \mid s \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \cup \Delta_1 \backslash \mathsf{dom}(\Delta_2) \cup \Delta_2 \backslash \mathsf{dom}(\Delta_1)$$

The typing rules for runtime are listed in Figure 5.5. (Label) types a label in a queue, while (Queue) forms a sequence of the types of the values in a queue: we ensure the disjointness of session environments of values, and apply a weakening of ended session types (Σ_0) for closure under the structure rules. (New$_s$) is the main rule for typing the two endpoint queues of a session. Types S_1 and S_2 can be given to the queues s and $\bar{s}$ when the session remainders S_1' and S_2' of $S_1 - \vec{\tau}_1$ and $S_2 - \vec{\tau}_2$ are dual session types *up to asynchronous subtyping*; more precisely, S_1' must be a subtype of the dual of S_2', written $S_1' \leqslant_c \overline{S_2'}$. Since the session is compatible, we can restrict s. (Par) composes processes, including queues, and records the session usage by $\odot$; this rule subsumes (Par) for programs. Note that, as this is a runtime typing system, the set of linear variables is empty.

(Label)
$$\frac{}{\Gamma;\emptyset;\emptyset \vdash l : l}$$

(Queue) if $\tau_i = U \to T$ then $\Sigma_i = \emptyset$
$$\frac{\Gamma;\emptyset;\Sigma_i \vdash h_i : \tau_i \quad i \in 1..n \quad \Sigma_0 = \{\vec{s} : \vec{\mathsf{end}}\}}{\Gamma;\emptyset;(\Sigma_0,..,\Sigma_n) \odot s:\tau_1..\tau_n \vdash s:h_1..h_n : \diamond}$$

(New$_s$)
$$\frac{\Gamma;\emptyset;\Delta, s:(S_1,\vec{\tau}_1), \bar{s}:(S_2,\vec{\tau}_2) \vdash P:\diamond \quad S_i - \vec{\tau}_i = S'_i \quad i \in 1,2 \quad S'_1 \leqslant_c \overline{S'_2}}{\Gamma;\emptyset;\Delta \vdash (\nu s)P:\diamond}$$

(New)
$$\frac{\Gamma,\, a:\langle S\rangle;\Lambda;\Delta \vdash P:\diamond}{\Gamma;\Lambda;\Delta \vdash (\nu a:\langle S\rangle)P:\diamond}$$

(Par)
$$\frac{\Gamma;\Lambda_{1,2};\Delta_{1,2} \vdash P_{1,2}:\diamond}{\Gamma;\Lambda_1,\Lambda_2;\Delta_1 \odot \Delta_2 \vdash P_1 \mid P_2:\diamond}$$

Figure 5.5: Runtime Typing for Asynchronous Higher-order π-calculus

5.4.1 Typing the Optimised Mobile Business Protocol

Using the program and runtime typing systems, we can now type the hotel booking example of § 5.2.2, in the presence of asynchronous optimisation for higher-order mobility. `Agent` and standard `Client` can be typed, by using the rules in Figures 4.8 and 4.9, as follows:

$$S_{\texttt{Agent}} = ![\texttt{int}].\&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\texttt{Agent}}, \mathsf{local} : S'_{\texttt{Agent}}]$$

$$\text{where} \quad S'_{\texttt{Agent}} = ?[\texttt{string}].?[\texttt{string}].![\texttt{double}].?[\texttt{int}].\mathsf{end}$$

$$\text{and} \quad S_{\texttt{client}} = \overline{S_{\texttt{Agent}}}$$

We then type `MClient` and obtain:

$$S_{\texttt{MClient}} = \oplus[\mathsf{move} : ![\mathsf{unit} \multimap \diamond].![\texttt{string}].![\texttt{string}].![\texttt{int}].?[\texttt{int}].?[\texttt{double}].\mathsf{end}]$$

Applying Definition 5.3.3 we verify that $S_{\texttt{MClient}} \leqslant_c \overline{S_{\texttt{Agent}}}$ (and $S_{\texttt{MClient}} \leqslant_c S_{\texttt{Client}}$). Then using typing rules (Conn,ConnDual) we can type both `MClient` and `Agent` with $a : \langle\overline{S_{\texttt{Agent}}}\rangle \in \Gamma$, after applying (Sub) on the premises of (ConnDual) typing the body of `MClient`.

We now demonstrate runtime typing; after three reduction steps of `MClient` | `Agent` we can have this configuration:

$$(\nu s)(\bar{s} \triangleright \{\mathsf{move} : \bar{s}?(code).(run\ code \mid \ldots), \mathsf{local} : \ldots\} \mid s:\texttt{rtt} \mid \bar{s}:\mathsf{move} \cdot \ulcorner s!\langle\texttt{ritz}\rangle \ldots \urcorner)$$

with $\bar{s}$ as the `Agent`'s queue. Both queues contain values including the linear higher-order code sent by MClient (which became **0** after this output). Using (Queue, Label) from Figure 5.5, we type $\bar{s}: \mathsf{move} \cdot \ulcorner s!\langle \mathtt{ritz} \rangle \ldots \urcorner$ with session environment

$$\{s : S'_{\mathtt{MClient}}, \bar{s} : \mathsf{move} \cdot \mathsf{unit} \multimap \diamond\}$$

where $S'_{\mathtt{MClient}}$ comes from typing the HO code containing s, and:

$$S'_{\mathtt{MClient}} = ![\mathtt{string}].![\mathtt{string}].![\mathtt{int}].?[\mathtt{int}].?[\mathtt{double}].\mathsf{end}$$

and similarly we type $s : \mathtt{rtt}$ with:

$$\{s : \mathtt{int}\}$$

.

The Agent $\bar{s} \triangleright \{\mathsf{move} : \ldots, \mathsf{local} : \ldots\}$ is typed with (Bra) under session environment:

$$\{\bar{s} : \&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\mathtt{Agent}}, \mathsf{local} : S'_{\mathtt{Agent}}]\}$$

The above session environments can be synthesised using $\odot$ to obtain:

$$\{s : (S'_{\mathtt{MClient}}, \mathtt{int}), \bar{s} : (\&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\mathtt{Agent}}, \mathsf{local} : S'_{\mathtt{Agent}}], \mathsf{move} \cdot \mathsf{unit} \multimap \diamond)\}$$

Now we use the rules in Figure 5.4 to calculate the session remainder of each queue:

$$S'_{\mathtt{MClient}} - \mathtt{int} = ![\mathtt{string}].![\mathtt{string}].![\mathtt{int}].?[\mathtt{double}].\mathsf{end}$$
$$\&[\mathsf{move} : ?[\mathsf{unit} \multimap \diamond].S'_{\mathtt{Agent}}, \mathsf{local} : S'_{\mathtt{Agent}}] - \mathsf{move} \cdot \mathsf{unit} \multimap \diamond = S'_{\mathtt{Agent}}$$

and we have:

$$![\mathtt{string}].![\mathtt{string}].![\mathtt{int}].?[\mathtt{double}].\mathsf{end} \leqslant_c \overline{S'_{\mathtt{Agent}}}$$

Finally, we can apply (New$_s$) and complete the derivation.

5.5 Type Soundness and Communication Safety

This section studies the key properties of our typing system. First, we show that typed processes enjoy subject reduction and communication safety.

We begin by introducing *balanced environments* which specify the conditions for composing environments of runtime processes. Our definition extends the one in [39] to accommodate for the presence of buffers, using session remainders.

Definition 5.5.1 (Balanced Δ)**.** $\mathsf{balanced}(\Delta)$ holds if whenever $\{s:(S_1,\vec{\tau}_1),\bar{s}:(S_2,\vec{\tau}_2)\}\subseteq\Delta$ with $S_1-\vec{\tau}_1=S_1'$ and $S_2-\vec{\tau}_2=S_2'$, then $S_1'\leqslant_c\overline{S_2'}$.

The definition is based on (New_s) in the runtime typing system (Figure 5.5): intuitively, all subprocesses generated from an initial typable program should conform to the balanced condition. We next define the ordering between the session environments which abstractly represents an interaction at session channels.

Definition 5.5.2 (Δ Ordering)**.** Recall $\odot$ defined in § 5.4. We define $\Delta\sqsubseteq_s\Delta'$ as follows:

$$s:?[U].S\odot s:U\vec{\tau}\;\sqsubseteq_s\;s:S\odot s:\vec{\tau}\qquad s:\&[l_i:S_i]_{i\in I}\odot s:l_k\vec{\tau}\;\sqsubseteq_s\;s:S_k\odot s:\vec{\tau}\quad k\in I$$

$$s:![U].S\odot\bar{s}:\vec{\tau}\;\sqsubseteq_s\;s:S\odot\bar{s}:\vec{\tau}U\qquad s:\oplus[l_i:S_i]_{i\in I}\odot\bar{s}:\vec{\tau}\;\sqsubseteq_s\;s:S_k\odot\bar{s}:\vec{\tau}l_k\quad k\in I$$

$$s:\mu\mathbf{t}.S\odot s':\vec{\tau}\;\sqsubseteq_s\;s:S'\odot s':\vec{\tau}'\quad\text{if}\quad s:S[\mu\mathbf{t}.S/\mathbf{t}]\odot s':\vec{\tau}\;\sqsubseteq_s\;s:S'\odot s':\vec{\tau}'$$

$$\Delta\odot\Delta_1\sqsubseteq_s\Delta\odot\Delta_2\quad\text{if}\quad\Delta_1\sqsubseteq_s\Delta_2\text{ and }\Delta\odot\Delta_1\text{ defined}$$

Note that if $\Delta_1\sqsubseteq_s\Delta_2$ and $\Delta\odot\Delta_1$ is defined, then $\Delta\odot\Delta_2$ is defined; and if $\mathsf{balanced}(\Delta)$ and $\Delta\sqsubseteq_s\Delta'$ then $\mathsf{balanced}(\Delta')$. Then we have:

Theorem 5.5.3 (Type Soundness)**.**

1. *Suppose* $\Gamma;\Lambda;\Delta\vdash P:\diamond$. *Then* $P\equiv P'$ *implies* $\Gamma;\Lambda;\Delta\vdash P':\diamond$.

2. *Suppose* $\Gamma;\emptyset;\Delta\vdash P:T$ *with* $\mathsf{balanced}(\Delta)$. *Then* $P\longrightarrow P'$ *implies* $\Gamma;\emptyset;\Delta'\vdash P':T$ *and either* $\Delta=\Delta'$ *or* $\Delta\sqsubseteq_s\Delta'$.

First we prove a number of supporting results; the proof of Type Soundness begins on page 135.

Lemma 5.5.4. *If* Σ_1,Σ_2 *defined and* $\Sigma_1'\leqslant_c\Sigma_1$ *and* $\Sigma_2'\leqslant_c\Sigma_2$ *then* Σ_1',Σ_2' *defined and* $\Sigma_1',\Sigma_2'\leqslant_c\Sigma_1,\Sigma_2$.

Proof. Trivial by the definition of $\leqslant_c$ on environments and the fact that it does not change the domain of an environment. □

Lemma 5.5.5. *If* $\Gamma;\Lambda,x;\Sigma\vdash P:T$ *then* x *is free in* P.

Proof. Since there is no weakening for the Λ set, the only way to introduce x in Λ,x is by applying the axiom (LVar) with subject x. But whenever a linear variable is bound, it is removed from the linear set of the conclusion; see (Abs) and (Rec). Hence x appears in P and is not bound. □

Lemma 5.5.6. *If $\Gamma;\Lambda;\Sigma,k:S \vdash P:T$ and k is not free in P then $S = \mathsf{end}$.*

Proof. Since there is no weakening for the Σ environment, and k is not free in P, the only way to introduce a mapping for k in $\Sigma,k:S$ is by applying the axiom (Nil). But then $S = \mathsf{end}$. □

Lemma 5.5.7 (Environment Properties). *1. If $\Delta \odot s:\vec{\tau}_1$ defined then $\Delta \odot s:\vec{\tau}_2$ defined for any $\vec{\tau}_1$ and $\vec{\tau}_2$.*

2. If $\Delta \odot s:S_1$ defined then $\Delta \odot s:S_2$ defined for any S_1 and S_2.

3. If Δ,Δ' defined then $\Delta \odot \Delta'$ defined and $\Delta,\Delta' = \Delta \odot \Delta'$.

4. $\Delta \odot \Delta' = \Delta' \odot \Delta$ and $(\Delta_1 \odot \Delta_2) \odot \Delta_3 = \Delta_1 \odot (\Delta_2 \odot \Delta_3)$.

5. If $\Delta_1 \odot \Delta_2$ defined and $\Delta_2 \sqsubseteq_s \Delta_3$ then $\Delta_1 \odot \Delta_3$ defined.

6. If $\mathsf{balanced}(\Delta)$ and $\Delta \sqsubseteq_s \Delta'$ then $\mathsf{balanced}(\Delta')$.

Proof. Straightforward from the definitions of $\mathsf{balanced}(\Delta)$, $\odot$ and $\sqsubseteq_s$. □

Lemma 5.5.8. *If $\Gamma;\Lambda;\Sigma \odot \Delta \vdash P:T$ and $\Sigma \leqslant_c \Sigma'$ then $\Gamma;\Lambda;\Sigma' \odot \Delta \vdash P:T$*

Proof. Outline: For each $s{:}S \in \Sigma$ with $s{:}S' \in \Sigma'$, and with $P \equiv (\nu\vec{a}{:}\langle\vec{S}\rangle)(\nu\vec{s})(P_1 \mid \ldots \mid P_n)$, we take cases on the free occurrence of s in some P_i. If P_i is not a queue process then by (Sub) we obtain $s{:}S'$ in the session environment of the subderivation for P_i. If P_i is a queue process then it is typed using (Queue) and we can apply (Sub) as before on the premises. Then using (New), (New$_S$) and (Par) we obtain the required judgement. □

Lemma 5.5.9 (Queue Subsumption). *If $\Gamma;\Lambda;\Delta \odot s{:}\vec{\tau}_1 U_1 \vec{\tau}_2 \vdash P:T$ and $U_1 \leqslant_c U_2$ then $\Gamma;\Lambda;\Delta \odot s{:}\vec{\tau}_1 U_2 \vec{\tau}_2 \vdash P:T$*

Proof. Last rule applied is (Queue); in the premises we can apply (Sub) on the typing judgement of the value that corresponds to the U_1 typing, then apply (Queue) using the new premise with U_2. □

We have the standard Weakening (Lemma 4.4.2) and Strengthening (Lemma 4.4.3) for Γ environments, and the restricted form of Strengthening (Lemma 4.4.4) for Σ.

The Substitution Lemma (4.4.10) remains the same from Chapter 4, noting that we only need to define substitution for terms that do not contain runtime elements, hence the original Lemma using Σ environments (and not Δ) is sufficient.

Proof of Theorem 5.5.3 (Type Soundness)

Part (1). *Subject congruence is standard, except for the case of garbage collection. The latter is easy: first use the restricted weakening environment Σ_0 of rule* (Queue) *to obtain, after $\odot$-composition, the balanced usage pairs $(\mathsf{end}, \varepsilon)$ for the dual ended queues; then by* (New$_S$) *the ended session can be restricted.*

Part (2). *For this part we proceed as standard by taking cases on the last reduction rule applied. For all cases we assume:*

$$\Gamma; \emptyset; \Delta \vdash P : T \quad (\star) \qquad \mathsf{balanced}(\Delta) \qquad P \longrightarrow P'$$

Case (beta) $\quad P = (\lambda(x{:}U).Q)V \qquad P' = Q\{V/x\}$

From $(\star)$ we have that the judgement for P has as last rule(s) a (possibly empty) sequence of applications of (Sub)*, and then* (App)*. We then have by the judgement $(\star)$ before* (Sub) *and the premises of* (App) *that:*

$$\Gamma; \emptyset; \Sigma_1, \Sigma_2 \vdash P : T' \quad (\mathit{1}) \qquad \Gamma; \emptyset; \Sigma_1 \vdash \lambda(x{:}U).Q : U' \multimap T' \quad (\mathit{2}) \qquad \Gamma; \emptyset; \Sigma_2 \vdash V : U' \quad (\mathit{3})$$

$$\Sigma_1, \Sigma_2 \leqslant_c \Delta \quad (\mathit{4}) \qquad U' \leqslant_c^{\circledast} U \quad (\mathit{5}) \qquad T' \leqslant_c^{\circledast} T \quad (\mathit{6})$$

$$\textit{By } (\dagger) \textit{ if } U' = U_0 \rightarrow H_0 \textit{ then } \Sigma_2 = \emptyset \quad (\mathit{7})$$

Note that the function may have also have the smaller type $U \multimap T''$ with $T'' \leqslant_c T'$.

(a) *$U = H$. To obtain (2) we have (after a possibly empty sequence of* (Sub)*), an application of* (Abs) *with:*

$$\Gamma, x{:}U; \Lambda; \Sigma_1' \vdash Q : T'' \quad (\mathit{8}) \qquad T'' \leqslant_c^{\circledast} T' \quad (\mathit{9}) \qquad \Sigma_1' \leqslant_c \Sigma_1 \quad (\mathit{10})$$

By the sidecondition of (Abs) *we have that if $U = U_1 \multimap T_1$ then $\Lambda = \{x\}$ else $\Lambda = \emptyset$, since $\Lambda \setminus x = \emptyset$ in (2). By Lemma 5.5.4 and (10) and since Σ_1, Σ_2 is defined we have that Σ_1', Σ_2 is defined and:*

$$\Sigma_1', \Sigma_2 \leqslant_c \Sigma_1, \Sigma_2 \quad (\mathit{11})$$

(a-1) *$x \in \mathsf{fv}(Q)$. By (8) and (3) and (5) and (7), using Lemma 4.4.10(1), we obtain:*

$$\Gamma; \emptyset; \Sigma_1', \Sigma_2 \vdash Q\{V/x\} : T'' \quad (\mathit{12})$$

Finally using (Sub) *with (11) and (4) for the session environment and (9) and (6) for* *the result type, we obtain:*

$$\Gamma;\emptyset;\Delta \vdash Q\{V/x\} : T$$

(a-2) *$x \notin \mathsf{fv}(Q)$. If $U = U_1 \multimap T_1$ then by* (Abs) *$x \in \Lambda$, but then by Lemma 5.5.5 $x \in \mathsf{fv}(Q)$ which contradicts the assumption. Otherwise, if $U = U_1 \to T_1$ or $U = \langle S\rangle$ we have $\Sigma_2 = \emptyset$. We have $P' = Q$ and by Lemma 6.5.1 and (8) we obtain:*

$$\Gamma;\Lambda;\Sigma_1' \vdash Q : T'' \quad (\mathit{13})$$

Then $\Sigma_1' \leqslant_c \Delta$ and with an application of (Sub) *we obtain the result as before.*

(b) *$U = S$. Then $V = s$. From (3), and (2) with* (Abs$_S$)*, following similar steps as before, we obtain:*

$$\Gamma;\emptyset;\Sigma_2 \vdash s : S' \quad (\mathit{14}) \qquad\qquad \Gamma;\emptyset;\Sigma_1', x : S \vdash Q : T'' \quad (\mathit{15})$$

By (5) and $\circledast$ we have $S \leqslant_c S'$, then by Lemma 4.4.10(2) with (14) and (15) we obtain:

$$\Gamma;\emptyset;\Sigma_1', s : S \vdash Q\{V/x\} : T'' \quad (\mathit{16})$$

We have that $\{s : S'\} \leqslant_c \Sigma_2$ in (14) and hence $\{s : S\} \leqslant_c \Sigma_2$, then using also (10) with Lemma 5.5.4 as before we obtain $\Sigma_1', s : S \leqslant_c \Sigma_1, \Sigma_2$. Then using (4) and (9) and (6) with (Sub) *on (16) we obtain*

$$\Gamma;\emptyset;\Delta \vdash Q\{V/x\} : T$$

□

Case (send) $\qquad P = s!\langle V\rangle.Q \mid \overline{s}:\vec{h} \qquad P' = Q \mid \overline{s}:\vec{h}\cdot V$

The last rule applied was (Par) *for runtime. From this we have:*

$$\Gamma;\emptyset;\Sigma_1 \vdash s!\langle V\rangle.Q : \diamond \quad (\mathit{1}) \qquad \Gamma;\emptyset;\Sigma_2 \odot \overline{s}:\vec{\tau} \vdash \overline{s}:\vec{h} : \diamond \quad (\mathit{2}) \qquad \Delta = \Sigma_1 \odot \Sigma_2 \odot \overline{s}:\vec{\tau} \quad (\mathit{3})$$

After a possible application of (Sub) *on (1), by* (Send) *and its premises:*

$$\Gamma;\emptyset;\Sigma_1' \vdash s!\langle V\rangle.Q : \diamond \quad (\mathit{4}) \qquad \Sigma_1' = (\Sigma_{11}, \Sigma_{12}) \setminus \{s{:}S\}, s{:}![U].S \quad (\mathit{5}) \qquad \Gamma;\emptyset;\Sigma_{11} \vdash Q : \diamond \quad (\mathit{6})$$

$$\Gamma;\emptyset;\Sigma_{12} \vdash V : \diamond \quad (\mathit{7}) \qquad s{:}S \in \Sigma_{1i} \quad i \in 1,2 \quad (\mathit{8}) \qquad \mathit{if}\ U = U_1 \to T_1\ \mathit{then}\ \Sigma_{12} = \emptyset \quad (\mathit{9})$$

$$\Sigma_1' \leqslant_c \Sigma_1 \quad (\mathit{10})$$

Using (8) with $\Sigma_{11},\Sigma_{12} = \Sigma'_{11},\Sigma'_{12},s:S$*, we get, using (3), (10) and Lemma 5.5.8:*

$$\begin{array}{ll}
\Sigma'_1 \odot \Sigma_2 \odot \bar{s}:\vec{\tau} & \textit{is defined} \\
(\Sigma_{11},\Sigma_{12})\setminus\{s:S\}, s:![U].S \odot \Sigma_2 \odot \bar{s}:\vec{\tau} & \textit{is defined} \\
(\Sigma'_{11},\Sigma'_{12},s:S)\setminus\{s:S\}, s:![U].S \odot \Sigma_2 \odot \bar{s}:\vec{\tau} & \textit{is defined} \\
(\Sigma'_{11},\Sigma'_{12},s:![U].S) \odot \Sigma_2 \odot \bar{s}:\vec{\tau} & \textit{is defined}
\end{array}$$

Then using Lemma 5.5.7(3) on the above, followed by Lemma 5.5.7(1–2), we obtain:

$$\Sigma'_{11} \odot \Sigma'_{12} \odot s:S \odot \Sigma_2 \odot \bar{s}:\vec{\tau}U \qquad \textit{is defined} \quad (\textbf{\textit{11}})$$

In (2) the last rule applied was (Queue) *and combining the premises and adding (7) we obtain, by a new application of* (Queue) *(noting also (9) which is needed):*

$$\Gamma;\emptyset;\Sigma_2 \odot \bar{s}:\vec{\tau}U \vdash \bar{s}:\vec{h}\cdot V : \diamond \quad (\textbf{\textit{12}})$$

where the session environment is defined by (11). Then using (6), (12), and (Par)*, we obtain:*

$$\Gamma;\emptyset;\Sigma_{11} \odot \Sigma_2 \odot \Sigma_{12} \odot \bar{s}:\vec{\tau}U \vdash P' : \diamond \quad (\textbf{\textit{13}})$$

By Lemma 5.5.7(3) we have:

$$\Sigma_{11} \odot \Sigma_{12} = \Sigma_{11},\Sigma_{12} = \Sigma'_{11},\Sigma'_{12},s:S = \Sigma'_{11} \odot \Sigma'_{12} \odot s:S$$

and (13) becomes:

$$\Gamma;\emptyset;\Sigma'_{11} \odot \Sigma'_{12} \odot s:S \odot \Sigma_2 \odot \bar{s}:\vec{\tau}U \vdash P' : \diamond \quad (\textbf{\textit{14}})$$

By Lemma 5.5.8 since $\Sigma'_{11} \odot \Sigma'_{12} \leqslant_c \Sigma_1 \setminus s$*, we have from (14) that:*

$$\Gamma;\emptyset;(\Sigma_1 \setminus s) \odot s:S \odot \Sigma_2 \odot \bar{s}:\vec{\tau}U \vdash P' : \diamond \quad (\textbf{\textit{15}})$$

From (10) we have $s:![U].S \in \Sigma'_1$ *and by (5) there is* $s:S' \in \Sigma_1$ *with* $![U].S \leqslant_c S'$*. By the definition of simulation* $\mathsf{unfold}^n(S') = \mathcal{A}\langle ![U'].S'_h\rangle^{h\in H}$ *and* $U \leqslant_c^{\circledast} U'$ *and* $S \leqslant_c \mathcal{A}\langle S'_h\rangle^{h\in H}$*. Finally using Lemma 5.5.8 and Lemma 5.5.9 on (15) we can obtain:*

$$\Gamma;\emptyset;\Delta' \vdash P' : \diamond$$

where $\Delta' = (\Sigma_1 \setminus s) \odot s : \mathcal{A}\langle S'_h \rangle^{h \in H} \odot \Sigma_2 \odot \bar{s} : \vec{\tau}U'$ *and it holds that* $\Delta \sqsubseteq_s \Delta'$. □

Case (recv) *is very similar to* (beta)*; the rest are easy to obtain.* □

We now formalise communication-safety (which subsumes the usual type-safety). First, an *s-queue* is a queue process $s : \vec{h}$. An *s-input* is a process of the shape $s?(x).P$ or $s \rhd \{l_i : P_i\}_{i \in I}$. An *s-output* is a process $s!\langle V \rangle.P$ or $s \lhd l.P$. Then, an *s-process* is an *s-queue*, *s-input* or *s-output*. Finally, an *s-redex* is a parallel composition of either an *s-input* and non-empty *s-queue*, or an *s-output* and $\bar{s}$*-queue*.

Definition 5.5.10 (Error Process)**.** We say P is an *error* if $P \equiv (\nu\vec{a})(\nu\vec{s})(Q \mid R)$ where Q is one of the following: (a) a $|$-composition of two *s-processes* that does not form either an *s-redex* or an *s-input* and an empty *s-queue*; (b) an *s-redex* consisting an *s-input* and *s-queue* such that $Q = s?(x).Q' \mid s : l_k\vec{h}$ or $Q = s \rhd \{l_i : P_i\}_{i \in I} \mid s : V\vec{h}$; (c) an *s-process* for $s \in \vec{s}$ with $\bar{s}$ not free in R or Q; (d) a prefixed process or application containing an *s-queue*.

The above says that a process is an error if (a) it breaks the linearity of s by having e.g. two *s-inputs* in parallel; (b) there is communication-mismatch; (c) there is no corresponding opponent process for a session; or (d) it encloses a queue under prefix, thus making it unavailable. As a corollary of Theorem 5.5.3, we achieve the following general communication-safety theorem, subsuming the case that P is an initial program.

Theorem 5.5.11 (Communication Safety)**.** *If* $\Gamma; \Lambda; \Delta \vdash P : \diamond$ *with* $\mathsf{balanced}(\Delta)$*, then* P *never reduces into an error.*

Proof The proof is very similar to the one for Type Safety in the Synchronous HOπ^{s}, stated as Theorem 4.4.14. The strategy is to show, with case analysis, that no error process is typable. This is easily demonstrated by contradiction. □

5.6 Concluding Remarks

The proof of the transitivity in this work requires a more complex construction of the transitive closure $\mathbf{trc}(\mathfrak{R}_1, \mathfrak{R}_2)$ (Definition 5.3.13) than the one in [65] due to the higher-order constructs. In spite of the richness of the type structures, we proposed a more compact runtime typing and proved communication safety in the presence of higher-order code, which is not presented in [65]. Moreover, our new typing system extends naturally the previous linear typing system of Chapter 4 demonstrating a smooth integration of two kinds of type-directed optimisation.

The subtyping system of [39] does not provide any form of asynchronous permutation, thus does not need the nested n-times unfolding (Definition 5.3.1). Our transitivity proof and the algorithmic subtyping are significantly more involved than in [39] due to the incorporation with n-time unfolding, permutation, and higher-order functions.

Our treatment of runtime typing, specifically our method for typing session queues and the use of *session remainders*, is more compact than previous asynchronous session works [49, 9, 10] where they use the method of *rolling-back* messages – the head type of a queue typing *moves* to the prefix of the session type of a process using the queue, and then compatibility is checked on the constructed types. Our method is simpler, as we remove type elements appearing in a queue from its typing. On the other hand, our queue typing is more similar to that of the functional language in [40], where smaller types are obtained after *matching* with buffer values. Our method works with queue types rather than with values directly, hence it can be extended smoothly to handle asynchronous optimisation, which is not treated in [40]. For example we allow a type consisting an output followed by an input action to be reduced with a type corresponding to the input, leaving the output prefix intact. Using a more delicate composition between values and queue typing, our system enables linear mobile code to be stored in the queues.

An analysis of asynchronous session action permutations, encompassing an asynchronous "acceptance" relation which accommodates for output actions performed in advance, appears in an unpublished manuscript [69]. The authors suggest that their algorithm is terminating. However, if their system admits $\mu\mathbf{t}.![U_1].\mathbf{t}$ as a subtype of $\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$, which as we show on page 116 induces an infinite simulation, then it is unclear how it avoids divergence without any special provision.

6 Sessions and Objects

Overview *Here we introduce asynchronous session typing into the Abadi and Cardelli imperative object calculus. Our system shows how process–oriented programs can be organised in object units with the structuring benefits that arise from that such as dynamic dispatch, object subtyping, and self-recursion. The asynchronous subtyping follows an iso-recursive approach.*

6.1 Introduction

This Chapter addresses the question of how to design and implement type-safe concurrent programs communicating via message-passing, in a natural programming style based on objects. We formalise a small object calculus enabling structured concurrent programming of typed bidirectional protocols via *sessions*. Our language is typed using both object types and session types. The formalism extends the first-order imperative object calculus of Abadi and Cardelli [2], adding queue-based primitives for asynchronous communication where the senders send messages without being blocked (but preserving their order). As before, each session consists of two queues, or *end-points*, used in a symmetric way with an output on one corresponding to an input on the other. Queues are considered linear values, which forbids general aliasing, but we allow controlled encapsulation within *linear objects*, obtaining a powerful and uniform programming paradigm. The resulting language is small but very expressive, due to the high degree of amalgamation of object and structured concurrency primitives.

Contributions

While many theoretical works exist, no formalism has successfully integrated all of the main features appearing in object-oriented languages without classes: imperative constructs, thread spawning, mutable state, recursion, structural object types, and subtyping. Our choice to augment session-based communication in an extension of the object calculus distills essential features of higher-order and object behaviours, offering a theoretical tool to analyse delicate type safety conditions.

Compared to class-based calculi, the object-calculus approach to sessions has significant advantages. First, objects are more primitive, in the sense that different class-based and trait-based systems can be encoded [2, 22]. In particular, our work can be used to extend languages with structural object types such as OCaml and also the mainstream class-based languages Java and C# which use nominal types, as well as languages with classes and traits such as Scala [71]. Secondly, higher-order structures are naturally expressed as objects, providing a powerful programming style [22] which manifests also in class-based languages that support anonymous classes defined within expressions. Thus, the object calculus provides a versatile and expressive basis for object-oriented languages including, but not limited to, those that are class-based.

Our main contributions are summarised as follows:

- Linear objects allow higher-order mobile code to encapsulate active sessions, while mobility of queue end-points realises higher-order session communication. The feature of linear higher-order code was not allowed in previous class-based systems [31, 26], in which it would enable anonymous classes to use endpoints appearing free in their definition.

- We define a new formulation for recursive sessions, naturally integrating session-based choices with method invocations. This iso-recursive approach to sessions has not appeared previously in the literature, and it proves natural in the object calculus which follows the same approach for recursive methods.

- We formulate an iso-recursive version of asynchronous subtyping, capturing the essential properties in a tractable framework.

- The typing system of our calculus guarantees type soundness and communication safety in the presence of asynchronous session subtyping.

6.2 The Session Objects Calculus

The **session**ς-calculus is based on the imperative object calculus **imp**ς [2]. Objects can be invoked, updated, and cloned; terms can reduce in sequence and also concurrently; session primitives enable structured interactions based on asynchronous message passing. Concurrency, method arguments, linear objects, and sessions are additions to the original object calculus.

Reduction is defined on configurations consisting an extended runtime syntax paired with a heap where object names map to objects and queue names map to queues.

6.2.1 Syntax

$a, b, c \ ::=$	term
u	identifier
$[\tau \,|\, l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}]$	object
$[\tau \,|\, l_i{=}\lambda(y_i)\,b_i{}^{i\in I}]$	linear object
$w.l{\leftarrow}u$	method invocation
$w.l \Leftarrow \varsigma(x)\,\lambda(y)\,b$	method update
$\mathsf{clone}(w)$	cloning
$\mathsf{let}\ x{=}a\ \mathsf{in}\ b$	sequencial evaluation
$\mathsf{spawn}\ a$	concurrent evaluation
$\mathsf{let}\ (x,y) = \mathsf{new\,session}\langle\kappa_1,\kappa_2\rangle\ \mathsf{in}\ b$	session creation
$u!v$	output
$u?$	input
$w.l \lhd u$	selection
$u \rhd w$	branching
$\mathsf{close}(u)$	session closing
$u, v, w \ ::=$	identifier
x, y, z	variable
o	object name
s	queue name
$\bar{s}$	dual of s, with $\bar{\bar{s}} = s$

Figure 6.1: Syntax

The syntax of the **session**ς-calculus is given in Figure 6.1, with terms ranged over a, b, c. We have identifiers (u, v, w) which can be variables (x, y, z), object names (o), or queue endpoints $(s, \bar{s})$. In the remaining, we use the convention that u is used for queues, w for objects, and v for both kinds of value. The *dual* of a queue endpoint s is denoted $\bar{s}$, and represents the other endpoint of the same session. The operation is self-inverse hence $\bar{\bar{s}} = s$.

An object $[\tau \,|\, l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}]$ is a label-indexed collection of methods. The annotation τ is the type of the object. A method $\varsigma(x)\,\lambda(y)\,b$ has body b, self variable x (ς-bound), and queue argument y (λ-bound). The self variable allows methods to be mutually recursive. Contrary to regular objects which are unrestricted, a linear object $[\tau \,|\, l_i{=}\lambda(y_i)\,b_i{}^{i\in I}]$ will be used exactly once and should not be aliased; consequently, its methods have no access to self, and only take a queue argument. Later typing ensures that a linear object has at least one method.

An invocation of a method l with argument u is written $w.l{\leftarrow}u$. This is a departure from **imp**$\boldsymbol{\varsigma}$, where there is no argument in methods. In a standard way, a method update $w.l \Leftarrow \varsigma(x)\,\lambda(y)\,b$ modifies a heap at w so that l maps to the new method. Note that this operation only applies to unrestricted objects – linear objects are immutable. To perform a shallow copy of the object mapped to by w in the heap, we write $\mathsf{clone}(w)$, which returns a new name pointing to a copy of the object. Sequencing is expressed with a let-expression, and $\mathsf{spawn}\ a$ launches a thread with body a.

The term $\mathsf{let}\,(x,y) = \mathsf{new\,session}\langle\kappa_1,\kappa_2\rangle\ \mathsf{in}\ b$ creates a new session, resulting in the instantiation, within the session body b, of x and y to a fresh pair of endpoint queues. The annotations κ_1 and κ_2 correspond to the session type of each endpoint. To emit a value v over x we write $x!\,v$; to obtain the value, we perform an input on y, as in $\mathsf{let}\ z = y?\ \mathsf{in}\ b'$. An output on x places a value at the end of the queue of y; an input on x removes the first value on the queue of x and returns it, or blocks if the queue is empty. There cannot be race conditions: each endpoint has output capability on the dual's queue and input capability on its own.

It is desirable to allow alternative behaviours to emerge as choices, within parts of a single session. To preserve determinism, when an internal choice (selection) is made on an endpoint, there will be a corresponding external choice (branching) on its dual; this choice is determined by a method name (a label). With $w_1.l \lhd x$ a choice of branch l is performed on x, implemented as an invocation $w_1.l{\leftarrow}x$. This choice is communicated to the dual endpoint, which receives it using $y \rhd w_2$, and realises it as $w_2.l{\leftarrow}y$.

This unified approach reduces the normally verbose syntax of session branching to a single method invocation on each endpoint. Also, it allows us to implement session recursion directly at the object level: nested session choices are implemented with nested method calls, and both endpoints are used recursively.

Sessions have to be closed at both ends for the interaction to be complete, using $\mathsf{close}(u)$.

Bound variables are x_i, y_i in $[\tau \,|\, l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i \in I}]$, y_i in $[\tau \,|\, l_i{=}\lambda(y_i)\,b_i{}^{i \in I}]$, x, y in $w.l \Leftarrow \varsigma(x)\,\lambda(y)\,b$ and $\mathsf{let}\,(x,y) = \mathsf{new\,session}\langle\kappa_1,\kappa_2\rangle\ \mathsf{in}\ b$, and x in $\mathsf{let}\ x = a\ \mathsf{in}\ b$. The notions of free variables, alpha equivalence and substitution are standard. $\mathsf{fv}(a)$ denotes the set of free variables in a.

Abbreviations We denote the empty object with $[\,]$. The notation $a;\,b$ means $\mathsf{let}\ x = a\ \mathsf{in}\ b$ with $x \notin \mathsf{fv}(b)$. Parentheses are used to distinguish terms like $(\mathsf{spawn}\ a);\,b$ and $\mathsf{spawn}\ (a;\,b)$.

We also use the following abbreviations. Below x is fresh:

$$\begin{array}{rcl} u \triangleright [\ldots] & \equiv & \mathsf{let}\, x = [\ldots] \;\mathsf{in}\; u \triangleright x \\ [\tau \,|\, l = \_] \triangleleft u & \equiv & \mathsf{let}\, x = [\tau \,|\, l = \_] \;\mathsf{in}\; x.l \triangleleft u \\ w.l \triangleleft (u_1, u_2) & \equiv & \mathsf{let}\, x = w.l \triangleleft u_1 \;\mathsf{in}\; x.l \triangleleft u_2 \end{array}$$

6.2.2 Configurations

Reduction is defined on configurations of an extended language embedded in the definition of evaluation contexts:

$$E ::= E_s \mid E \,|\, b \mid b \,|\, E \qquad E_s ::= \langle \cdot \rangle \mid \mathsf{let}\, x = E_s \;\mathsf{in}\; b$$

Sequential contexts E_s ensure a call-by-value evaluation order for terms structured using (possibly nested) let-expressions. General reduction contexts E extend sequential contexts with parallel composition. Although the user syntax in Figure 6.1 does not include a production $a \,|\, b$, new threads are created with spawn, hence the extension. Evaluation order in parallel-composed contexts is non-deterministic: the hole can occur in any component. The separation of sequential and concurrent contexts enforces that threads are top-level, i.e., $\mathsf{let}\, x = (E \,|\, a) \;\mathsf{in}\; b$ is not well-formed.

Heaps are defined as follows:

$$\begin{array}{rcl} B & ::= & \emptyset \\ & | & B \cdot s \mapsto \vec{r}_1 \cdot \bar{s} \mapsto \vec{r}_2 \\ & | & B \cdot o \mapsto [\tau \,|\, l_i = \varsigma(x_i)\, \lambda(y_i)\, b_i{}^{i \in I}] \\ & | & B \cdot o \mapsto [\tau \,|\, l_i = \lambda(y_i)\, b_i{}^{i \in I}] \\ \\ r & ::= & v \mid l \mid \mathsf{end} \end{array}$$

Endpoint queues appear in pairs of duals. Each queue is a vector of runtime values drawn from the set of identifiers, labels, and the special value end. Labels enable selection and branching, where a method name is communicated; the value end is used to mark the end of a session and is especially useful for de-allocation of queues.

6.2.3 Reduction

The reduction rules are formalised in Figure 6.2. To denote that a closed configuration consisting an evaluation context E with heap B reduces to a new configuration with context E' and heap B'

(R-CONTEXT)
$$\frac{a,B \rightarrow a',B'}{E\langle a\rangle,B \rightarrow E\langle a'\rangle,B'}$$

(R-OBJECT)
$$\frac{c \in \{\, [\tau \mid l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}],\ [\tau \mid l_i{=}\lambda(y_i)\,b_i{}^{i\in I}]\,\} \qquad o \notin \mathsf{dom}(B)}{c,B \rightarrow o,B\cdot o \mapsto c}$$

(R-SPAWN)
$$E\langle \mathsf{spawn}\ a\rangle,B \rightarrow E\langle[]\rangle \mid a,B$$

(R-CLONE)
$$\frac{B(o) = [\tau \mid l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}] \qquad o' \notin \mathsf{dom}(B)}{\mathsf{clone}(o),B \rightarrow o',B\cdot o' \mapsto B(o)}$$

(R-LET)
$$\mathsf{let}\ x{=}v\ \mathsf{in}\ b,B \rightarrow b\{^{v}/_{x}\},B$$

(R-INVOKE)
$$\frac{B(o) = [\tau \mid l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}]}{o.l_k{\leftarrow}s,B \rightarrow b_k\{^{o}/_{x_k}\}\{^{s}/_{y_k}\},B}$$

(R-LINVOKE)
$$\frac{B(o) = [\tau \mid l_i{=}\lambda(y_i)\,b_i{}^{i\in I}]}{o.l_k{\leftarrow}s,B \rightarrow b_k\{^{s}/_{y_k}\},(B\setminus o)}$$

(R-UPDATE)
$$\frac{B \equiv B',o \mapsto [\tau \mid l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}]}{o.l_k \Leftarrow \varsigma(x)\,\lambda(y)\,b,B \rightarrow o,B'\cdot o \mapsto [\tau \mid l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I\setminus k}, l_k{=}\varsigma(x)\,\lambda(y)\,b]}$$

(R-SESSION)
$$\frac{s,\bar{s} \notin \mathsf{dom}(B)}{\mathsf{let}\ (x,y) = \mathsf{new\,session}\langle\kappa_1,\kappa_2\rangle\ \mathsf{in}\ b,B \rightarrow b\{^{s}/_{x}\}\{^{\bar{s}}/_{y}\},B\cdot s \mapsto \varepsilon\cdot \bar{s} \mapsto \varepsilon}$$

(R-PUT)
$$s!\,v,B\cdot\bar{s} \mapsto \vec{r} \rightarrow [\,],B\cdot\bar{s} \mapsto \vec{r}v$$

(R-GET)
$$s?,B\cdot s \mapsto v\vec{r} \rightarrow v,B\cdot s \mapsto \vec{r}$$

(R-SELECT)
$$o.l_k \triangleleft s,B\cdot\bar{s} \mapsto \vec{r} \rightarrow o.l_k{\leftarrow}s,B\cdot\bar{s} \mapsto \vec{r}l_k$$

(R-BRANCH)
$$s \triangleright o,B\cdot s \mapsto l_k\vec{r} \rightarrow o.l_k{\leftarrow}s,B\cdot s \mapsto \vec{r}$$

(R-END)
$$\mathsf{close}(s),B\cdot\bar{s} \mapsto \vec{r} \rightarrow [\,],B\cdot\bar{s} \mapsto \vec{r}\,\mathsf{end}$$

(R-QCLEAN)
$$a,B\cdot s \mapsto \mathsf{end}\cdot\bar{s} \mapsto \mathsf{end} \rightarrow a,B$$

(R-TCLEANL)
$$o \mid E,B \rightarrow E,B$$

(R-TCLEANR)
$$E \mid o,B \rightarrow E,B$$

Figure 6.2: Reduction

we write:

$$E,B \rightarrow E',B'$$

With rule (R-OBJECT), when an object is encountered, a new heap mapping for it is created and returned. Rule (R-CLONE) takes an existing object and adds a copy under a fresh name. In rule (R-SPAWN) the thread body a is taken out of its original context and placed in parallel, with the empty object $[]$ placed as the result in the original context.

Rule (R-INVOKE) for unrestricted objects is standard, substituting the object for the self variable and the queue for the formal argument within a copy the method body; rule (R-LINVOKE) for linear objects is similar, however there is no self argument, and importantly, the object is removed from the heap – this is safe because typing guarantees it will only be used once. Rule (R-UPDATE) is as expected, replacing the old method with the new one within the object's method set.

In (R-SESSION) fresh dual queues s and $\bar{s}$ are created and substituted for the variables x and y within the scope of session body b. The queues are added to the heap initialised to ε, the empty vector. Queues facilitate asynchronous session communication: a sending action is never blocked (non-blocking) and two messages sent on the same queue arrive in the sending order (message order preservation per session).

Rule (R-PUT) for output on a queue s places the value v (an object or queue name) at the end of the dual queue $\bar{s}$; its result is the empty object. Rule (R-GET) for input on s is applicable if there is at least one value in the same queue, in which case the value is removed from the queue and returned.

In rule (R-SELECT) for selection $o.l_k \triangleleft s$ the method name l_k is enqueued on $\bar{s}$, the dual of the argument s, similarly to output; the selection then becomes an invocation $o.l_k \Leftarrow s$, i.e., the same method is invoked locally with argument s. Rule (R-BRANCH) is used to receive a selection, using $s \triangleright o$. It requires the first value of s to be a method name l_k, which is dequeued and used in the resulting invocation $o.l_k \Leftarrow s$.

Rule (R-END) is similar to output, but in this case the special value end is appended to the dual queue. Rule (R-QCLEAN) deallocates both s and its dual $\bar{s}$ when both have end as the only remaining value, because this signifies that the session is complete at both endpoints. Finished threads are removed from a parallel composition by (R-TCLEANL,R).

6.2.4 Example: Instant Messenger

In Figure 6.3 we show an example which utilises most features of the calculus, demonstrating the subtle interplay between objects, sessions, and recursion: choices are naturally implemented as recursive objects; linear objects enable complex interleavings, encoding multi-parameter methods under a concise declarative programming style.

We chose to write the client component of a simple “instant messenger” requiring message passing, choice, mutual recursion, higher-order sessions, and thread spawning. We only show how to implement the object for sending messages and files; a usable program would also need a server, running concurrently, for receiving them.

```
1   let $client =$
2     $[\tau_1 \,|\, arg = \varsigma(x)\,\lambda(y)$
3       $[\tau_2 \,|\, arg = \lambda(z)$
4         $y \triangleright [\tau_3 \,|\, exit = \lambda(y_1)\,\mathsf{close}(y_1)\,;[\tau_4 \,|\, exit = \lambda(y_2)\,\mathsf{close}(y_2)] \triangleleft z\,,$
5           $msg = \lambda(y_1)$ let $m = y_1?$
6             in $[\tau_5 \,|\, msg = \lambda(y_2)\, y_2!m; x.arg \triangleleft (y_1, y_2)] \triangleleft z\,,$
7           $file = \lambda(y_1)$ let $fn = y_1?$
8             in let $(x_1, x_2) = \mathsf{new\,session}\langle \kappa_1, \kappa_2 \rangle$
9               in spawn let $f = \mathsf{get\_file}(fn)$
10                        in $(x_1!fn; x_1!f; \mathsf{close}(x_1))$;
11                $[\tau_6 \,|\, file = \lambda(y_2)$
12                        $y_2!x_2; x.arg \triangleleft (y_1, y_2)\,] \triangleleft z\,]\,]\,]$
13  in $client.arg \triangleleft (s_{\mathsf{usr}}, s_{\mathsf{im}})$
```

Figure 6.3: Example: Instant Messenger

In line 1, we define the *client* object which is instantiated, in line 13, with the queues s_{usr} and s_{im}. The first queue is used to control the object (e.g. through a session with a user interface), and the second is used for instant messaging with a suitable server object, which we assume is separately defined.

In line 2 we start the definition of a shared object with a method *arg* which will take s_{usr} for the y argument. Invoking *arg* will result in the object starting in line 3, which is linear and will take s_{im} for its z argument.

Then, in line 4, we branch on queue y, offering the choices (i.e. method names) *exit*, *msg* and *file*. The first is used to end the two sessions: first y_1, which will be the same as y, is closed; then, using a nested object, we propagate the user choice by performing a selection of *exit* also on z ending the session. The method *msg* is for sending a message; the method *file* is for sending a file. If *msg* is chosen, a message is received from the user queue y_1, then in line 6 a selection of *msg* is performed on the messaging queue z, relaying the message, and then invoking the outer object with arguments y_1 and y_2 (which will have the values s_{usr} and s_{im}) implementing a recursive protocol. Method *file*, starting in line 7, receives a filename from the user queue, then in line 8 creates a new pair of endpoints for the file transfer.

In lines 9 and 10 a new thread is spawned in which the file f is obtained using the filename *fn*, and then *fn* and f are sent over the first of the new endpoints, x_1, which is then closed. (We assumed a system function 'get_file.') After spawning, in lines 11-12, an object is sequence composed, and selection of *file* is made on z, the endpoint x_2 is sent over z to the server object so that the file can be received, and finally a recursive invocation is performed repeating the client protocol.

For the file transfer we created a new session running on a different thread, and we communicated one of the new endpoints over the existing session, to save our client from blocking. Finally, note that the code can be written in a modular way, separating the objects, by abstracting over the free queues.

6.3 Typing

Our session types can be thought of as process types encompassing typed input and output, sequencing, and recursive label-indexed branching (external choice) and selection (internal choice). The language of types can express rich interaction patterns, and a deterministic behaviour that arises from linear use of dually typed queues.

After introducing the syntax of types, we define type duality and the subtyping relation, and then describe the typing system. An integration of objects and sessions requires delicate conditions on both types and typing, which are justified by examples at the end. We also give session types and typing for the instant messenger example of § 6.2.4.

6.3.1 Types

The types are defined in Figure 6.4. Notice that we use a different set of metavariables than the one in the previous chapters, to avoid confusion since there are subtle differences. Value types range over τ, and can be complete session types, ranging over κ, or object types. An object type records the label-indexed argument and result types, σ_i and τ_i, for each method l_i: the argument is of session type and the result of value type. Linear object types are distinguished by the lin prefix.

τ ::=	value
κ	complete session
$[l_i : (\sigma_i)\, \tau_i{}^{i \in I}]$	object
$\mathsf{lin}[l_i : (\sigma_i)\, \tau_i{}^{i \in I}]$	linear object $(\star)$
σ ::=	session
π	partial session
κ	complete session
π ::=	partial session
ε	empty
$![\tau]$	output
$?[\tau]$	input
$\oplus(X)[l_i : \pi_i{}^{i \in I} \uplus l_j : \pi_j; X_j{}^{j \in J}]$	internal choice $(\star)$
$\&(X)[l_i : \pi_i{}^{i \in I} \uplus l_j : \pi_j; X_j{}^{j \in J}]$	external choice $(\star)$
$\pi;\pi$	sequence
κ ::=	complete session
end	ended session
X	type variable
$\oplus(X)[l_i : \kappa_i{}^{i \in I}]$	internal choice $(\star)$
$\&(X)[l_i : \kappa_i{}^{i \in I}]$	external choice $(\star)$
$\pi;\kappa$	complete sequence

$$(\star) \quad I \supset \emptyset$$

Figure 6.4: Types

Session types, ranging over σ, are partitioned into partial, ranging over π, and complete. Partial session types are building blocks for larger protocols. An output of a value of type τ is written $![\tau]$, and similarly input is $?[\tau]$. In the process calculi we studied there are no partial session types, but in the object calculus we chose to include them, because this is a simple way to allow methods to implement a part of a session without full knowledge of the protocol; this feature facilitates a greater degree of modularity, which is very relevant for the objectives of an object-oriented language.

Internal choice (selection) is represented by a set of label-indexed session types, and is prefixed with $\oplus$ followed by a bound type variable X which is used for tail-recursion within the branches. This type is equivalent to $\mu X.\oplus[l_i : {\pi_i}^{i\in I} \uplus l_j : \pi_j;{X_j}^{j\in J}]$. A pleasant consequence of our combined recursive structures is that, due to the presence of labels between the definition and the type variables, the types are always contractive. The branches are here shown partitioned into two sets, with $\uplus$ meaning that labels are distinct in each, and moreover we are ensuring that there should be at least one branch in which no type variable occurs. This separates the partial and complete internal choices, which is essential for the soundness of the typing system, see Example 6.3.5 (8). External choice (branching) is similar and uses the & prefix.

The sequence composition of partial session types is written $\pi;\pi'$, and ε denotes the empty sequence.

The grammar for complete session types adds the terminals end, marking the end of a session, and type variables X which are used as in partial types. Choice constructs are complete if and only if all branches are complete. A complete session type can only be the last component in a sequence. Finally, we assume that all bound type variables in a session type are chosen to be different.

Equality and Duality We use three axioms for type equality. First we regard ε as the unit of sequence composition:

$$\sigma;\varepsilon = \varepsilon;\sigma = \sigma$$

Next, the type end distributes within partial branches of a choice construct:

$$\oplus(X)[l_i : {\pi_i}^{i\in I} \uplus l_j : {\kappa_j}^{j\in J}];\mathsf{end} \;=\; \oplus(X)[l_i : \pi_i;\mathsf{end}^{i\in I} \uplus l_j : {\kappa_j}^{j\in J}]$$

$$\&(X)[l_i : {\pi_i}^{i\in I} \uplus l_j : {\kappa_j}^{j\in J}];\mathsf{end} \;=\; \&(X)[l_i : \pi_i;\mathsf{end}^{i\in I} \uplus l_j : {\kappa_j}^{j\in J}]$$

This type of equality should not be generalised to arbitrary composition, for example between $\oplus(X)[l_i : \pi_i{}^{i\in I} \uplus l_j : \kappa_j{}^{j\in J}];\sigma$ and $\oplus(X)[l_i : \pi_i;\sigma^{i\in I} \uplus l_j : \kappa_j{}^{j\in J}]$, because in the first type σ is done once, but in the second type it may be done more than once depending on recursion.

As standard, duality interchanges input and output, branching and selection, and distributes over sequencing and into branches. It does not distribute within the value type of input and output. Type variables, and the types ε and end, remain unaffected.

Formally, duality is defined by the following rules:

$$\overline{\varepsilon} = \varepsilon \qquad \overline{![\tau]} = ?[\tau] \qquad \overline{?[\tau]} = ![\tau]$$

$$\overline{\oplus(X)[l_i : \sigma_i{}^{i\in I}]} = \&(X)[l_i : \overline{\sigma}_i{}^{i\in I}] \qquad \overline{\&(X)[l_i : \sigma_i{}^{i\in I}]} = \oplus(X)[l_i : \overline{\sigma}_i{}^{i\in I}]$$

$$\overline{\pi;\sigma} = \overline{\pi};\overline{\sigma} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \overline{X} = X \qquad \overline{\overline{\sigma}} = \sigma$$

6.3.2 Subtypes and Asynchronous Subtyping

The rules for subtyping are given in Figure 6.5. The subtyping rule for objects follows [2]: a larger object type is a subtype of a smaller one, with method argument and result types invariant. Linear objects are treated in the same way, however, it is crucial that a linear object type must consist at least one method – otherwise the object would not be usable linearly since no method could be invoked. Rule (SUB-LINEARISE) formalises that an unrestricted object can safely be used as linear, similarly to the rule for functions in [63]; the converse, promoting a linear object to unrestricted type, is unsafe. These conditions are justified in Example 6.3.5 (1) and (2).

In session subtyping, output is covariant to the value type, and input is contravariant. Note that the input-output subtyping differs from the one for the π-calculus in [77]. This is because our relation means "if a queue is assigned σ_1, then it also satisfies σ_2" while the one in [77] says "if you must do at least σ_2, then you are allowed to do σ_1." In other words, we anticipate a contravariant dual process on each session which, by the interchanging of constructors induced by duality, becomes covariance on our side.

Our definition of session subtyping is more uniform with objects, as can be seen from the rules (SUB-SELECT) and (SUB-BRANCH). Selection and branching can be thought of as remote method invocation (over a session), where the first chooses a method, and the second must support it; thus if one can invoke an object it can also invoke one with larger interface, and dually an object that can support a protocol can also fulfil a smaller one. Hence selection with a larger indexing set is a supertype, and branching is dual. These rules are the same as those in [19].

(SUB-REFL)
$$\frac{}{\zeta \preccurlyeq \zeta}$$

(SUB-TRANS)
$$\frac{\zeta_1 \preccurlyeq \zeta_2 \qquad \zeta_2 \preccurlyeq \zeta_3}{\zeta_1 \preccurlyeq \zeta_3}$$

$$\zeta ::= \tau \mid \pi$$

(SUB-OBJ)
$$\frac{J \subseteq I}{[l_i : (\sigma_i)\,\tau_i{}^{i\in I}] \preccurlyeq [l_j : (\sigma_j)\,\tau_j{}^{j\in J}]}$$

(SUB-LOBJ)
$$\frac{\emptyset \subset J \subseteq I}{\mathsf{lin}[l_i : (\sigma_i)\,\tau_i{}^{i\in I}] \preccurlyeq \mathsf{lin}[l_j : (\sigma_j)\,\tau_j{}^{j\in J}]}$$

(SUB-LINEARISE)
$$\frac{\emptyset \subset I}{[l_i : (\sigma_i)\,\tau_i{}^{i\in I}] \preccurlyeq \mathsf{lin}[l_i : (\sigma_i)\,\tau_i{}^{i\in I}]}$$

(SUB-PUT)
$$\frac{\tau_1 \preccurlyeq \tau_2}{![\tau_1] \preccurlyeq ![\tau_2]}$$

(SUB-GET)
$$\frac{\tau_1 \preccurlyeq \tau_2}{?[\tau_2] \preccurlyeq ?[\tau_1]}$$

(SUB-SEQ)
$$\frac{\pi_1 \preccurlyeq \pi_2 \qquad \sigma_1 \preccurlyeq \sigma_2}{\pi_1;\sigma_1 \preccurlyeq \pi_2;\sigma_2}$$

(SUB-SELECT)
$$\frac{\forall i \in I.\ \sigma_i \preccurlyeq \sigma_i' \qquad \emptyset \subset I \subseteq J}{\oplus(X)[l_i : \sigma_i{}^{i\in I}] \preccurlyeq \oplus(X)[l_j : \sigma_j'{}^{j\in J}]}$$

(SUB-BRANCH)
$$\frac{\forall i \in I.\ \sigma_i \preccurlyeq \sigma_i' \qquad \emptyset \subset I \subseteq J}{\&(X)[l_j : \sigma_j{}^{j\in J}] \preccurlyeq \&(X)[l_i : \sigma_i'{}^{i\in I}]}$$

(SUB-ASYNCPUT)
$$\frac{\forall h \in H\,.\,X \notin \mathsf{ftv}(\sigma_h)}{![\tau];\mathcal{B}_X\langle\sigma_h\rangle^{h\in H} \preccurlyeq \mathcal{B}_X\langle ![\tau];\sigma_h\rangle^{h\in H}}$$

(SUB-ASYNCSEL)
$$\frac{\forall h \in H\,.\forall i \in I\,.X,Y \notin \mathsf{ftv}(\sigma_{ih})}{\oplus(Y)[l_i : \mathcal{B}_X\langle\sigma_{ih}\rangle^{h\in H\ i\in I}] \preccurlyeq \mathcal{B}_X\langle\oplus(Y)[l_i : \sigma_{ih}{}^{i\in I}]\rangle^{h\in H}}$$

Asynchronous Contexts: $\mathcal{B}_X \ ::= ?[\tau];\langle\cdot\rangle^{h\in H} \ \mid\ \&(X)[l_i : \langle\cdot\rangle^{h\in H\ i\in I}]$

Figure 6.5: Subtyping

Asynchronous Subtyping A consequence of our asynchronous semantics for message passing is that, as in the $\mathrm{HO}\pi^{\mathbf{as}}$, it is harmless to perform an output or selection "in advance" of any sequence of inputs and branchings. However, the relative order of outputs (and inputs) should not be permuted. First, at the bottom of Figure 6.5 we define a class of asynchronous multi-hole contexts, written $\mathcal{B}_X$, where X is a parameterisation specifying the bound type variable that may appear if we have a recursive branch context (last production). We write $\mathcal{B}_X\langle\sigma_h\rangle^{h\in H}$ for the context where the holes indexed by $h \in H$ are replaced by the corresponding types σ_h.

We write $\mathsf{ftv}(\sigma)$ for the free type variables of a session type, omitting the formal definition which is straightforward. We present two rules: the first is (SUB-ASYNCPUT), which permits an output to appear outside of the asynchronous context, which in turn can be an input or branching; the second, (SUB-ASYNCSEL), allows a selection to appear outside of the context (and enclose it). In both rules, there is a necessary limitation, which is that the bound type variable of the context $\mathcal{B}_X$ (if there is one), and also the type variable induced by the selection (Y in the last rule), cannot appear free in the continuations σ_h and σ_{ih}, respectively for each rule. Practically, this means that the types must not behave recursively, although branching, in general, facilitates recursion. The rationale for this condition is as follows: consider the first rule, (SUB-ASYNCPUT), then if the context is recursive, as for example in $\mathcal{B}_X\langle ![\tau];\sigma_h\rangle^{h\in H} = \&(X)[l : ![\tau];X]$, the output $![\tau]$ cannot be extracted, because inside the context the action is repetitive, but outside it will only occur once, since by the rule we would obtain $![\tau];\&(X)[l : X]$. Similarly for selection in rule (SUB-ASYNCSEL), with the additional constraint forbidding recursion on Y, as this would also induce an incorrect transformation. For example, if we could apply (SUB-ASYNCSEL) with $\&(X)[l_1 : \oplus(Y)[l_2 : Y]]$ as the supertype, we would obtain a subtype $\oplus(Y)[l_2 : \&(X)[l_1 : Y]]$, in which the internal branch is now performed repeatedly, when in the original type it was only performed once before the recursive selection.

However, we still obtain an enlarged typability, and the restriction on recursion is applied only for the type variables of the transformed types, which means that any recursion defined in a larger, enclosing scope, can still take place. For example we have that:

$$\&(X_1)[l_3 : \oplus(Y)[l_2 : \&(X)[l_1 : X_1]]] \preccurlyeq \&(X_1)[l_3 : \&(X)[l_1 : \oplus(Y)[l_2 : X_1]]]$$

using first (SUB-BRANCH) followed by (SUB-ASYNCSEL). This transformation is sound as both subcomponents are repetitive in the subtype and in the supertype, using the same type variable X_1.

As another positive example, the type $\sigma = ![\tau];?[\tau];\mathsf{end}$ has its dual $?[\tau];![\tau];\mathsf{end}$ as a supertype;

therefore two endpoints that are both assigned the same type σ can comprise a valid session. See Example 6.3.5 (3) for a concrete example.

6.3.3 Judgements and Environments

The typing judgements take the shape:

$$\Gamma;\Lambda;\Sigma \vdash a : \tau \qquad \text{and} \qquad \Gamma;\Lambda;\Sigma \vdash a_1 \,|\, \ldots \,|\, a_n : \diamond$$

where Γ, Λ, Σ are unordered environments giving the types for identifiers of unrestricted objects, linear objects, and queues, respectively. The inductive definition is:

$$\begin{array}{lcl} \Gamma & ::= & \emptyset \mid \Gamma, u : [l_i : (\sigma_i)\, \tau_i{}^{i \in I}] \\ \Lambda & ::= & \emptyset \mid \Lambda, u : \mathsf{lin}[l_i : (\sigma_i)\, \tau_i{}^{i \in I}] \\ \Sigma & ::= & \emptyset \mid \Sigma, u : \sigma \end{array}$$

We write $\mathsf{dom}(\Gamma)$ for the domain of Γ and similarly for Λ and Σ. For a judgement to be *well-formed*, we need that no identifier occurs in the domain of more than one environment, i.e., $\mathsf{dom}(\Gamma) \uplus \mathsf{dom}(\Lambda) \uplus \mathsf{dom}(\Sigma)$ must be defined. Also, all types assigned in environments must be closed, i.e., there should not occur free (session) type variables. Parallel compositions are typed with the process type $\diamond$.

Environments Λ and Σ are linear: every identifier is associated with a single usage, and therefore weakening (used when discarding an identifier) and contraction (used when an identifier is copied) are not allowed. We write $\Lambda_1 \uplus \Lambda_2$ for the environment that is the set union of Λ_1 and Λ_2, defined when the domains of Λ_1 and Λ_2 are disjoint. Similarly for $\Sigma_1 \uplus \Sigma_2$. The sequence composition $\Sigma_1 \fatsemi \Sigma_2$ is an extension of sequencing from session types to session environments. It is the partial non-commutative operation defined by:

$$\Sigma_1 \fatsemi \Sigma_2 = \Sigma_1 \uplus \Sigma_2 \quad \text{if } \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2) = \emptyset$$

$$(\Sigma_1, u : \pi) \fatsemi (\Sigma_2, u : \sigma) = (\Sigma_1 \fatsemi \Sigma_2), u : \pi;\sigma$$

Successful composition of environments using $(\uplus)$ and $(\fatsemi)$ prevents contraction: in the first case, there should be nothing to contract; in the second, multiple usages of an identifier are understood as one sequential usage, not as copying.

Subtyping is also extended to session environments:

$$\emptyset \preccurlyeq \emptyset \qquad \Sigma, u : \sigma \preccurlyeq \Sigma, u : \sigma' \quad \text{if } \sigma \preccurlyeq \sigma'$$

6.3.4 Typing Rules

The typing rules are defined in Figure 6.6. We assume that for a rule to be applicable, the environments in the consequence are defined.

Values Rule (OBJVAL) is the axiom for typing the identifiers of unrestricted objects. Rule (CLONE) is for typing the cloning of shared objects. Rule (LOBJVAL) is for linear objects, and rule (QUEUEVAL) is for queues. In (LOBJVAL) and (QUEUEVAL), the linear environments Λ and Σ do not contain irrelevant mappings, as this would amount to weakening. Consequently, for each linear value in the environments of a judgement, one of these axioms has been applied exactly once within one of the subderivations in its premises.

Concurrency Rule (SPAWN) is for typing thread bodies. There are two conditions: the first is that queues appearing free in the thread body a must be assigned complete session types; the second is that the result of a must be an unrestricted object, that is, a subtype of $[\,]$. Rule (PAR) types parallel-composed terms, requiring that linear elements are not shared between threads, and that the result of each thread is of shared type. These conditions are explained in Example 6.3.5 (4).

Sequencing Rule (LET) is for local definitions and sequencing. In the conclusion, no linear object in a occurs in the continuation b, and the session usages of a are sequence-composed with those of b. This follows from the call-by-value reduction order. Depending on the type τ of a, the mapping $x : \tau$ is added to the correct environment using a shorthand notation; if τ is linear, usual conditions will apply. Note that the type of a can be a complete session κ but not a partial session π, otherwise type safety can be violated; see Example 6.3.5 (5).

Subtyping Rule (SUBSUME) introduces subtyping for both session environments and types.

Objects Rule (OBJECT) is for unrestricted objects. Methods are typed with the object type for the self variable, and the queue must be used as mandated by the argument type. The object can be copied, therefore there should not be any use of free linear objects or queues within its methods,

(OBJVAL) $(\tau \equiv [\ldots])$
$$\frac{}{\Gamma, u : \tau; \emptyset; \emptyset \vdash u : \tau}$$

(CLONE) $(\tau \equiv [\ldots])$
$$\frac{}{\Gamma, u : \tau; \emptyset; \emptyset \vdash \mathsf{clone}(u) : \tau}$$

(LOBJVAL) $(\tau \equiv \mathsf{lin}[\ldots])$
$$\frac{}{\Gamma; \{u : \tau\}; \emptyset \vdash u : \tau}$$

(QUEUEVAL)
$$\frac{}{\Gamma; \emptyset; \{u : \kappa\} \vdash u : \kappa}$$

(SPAWN)
$$\frac{\Gamma; \Lambda; \{u_i : \kappa_i\}^{i \in I} \vdash a : [\,]}{\Gamma; \Lambda; \{u_i : \kappa_i\}^{i \in I} \vdash \mathsf{spawn}\ a : [\,]}$$

(PAR)
$$\frac{\forall i \in \{1..n\}\ .\ \Gamma; \Lambda_i; \Sigma_i \vdash a_i : [\,]}{\Gamma; \biguplus \Lambda_i; \biguplus \Sigma_i \vdash a_1 \,|\, \ldots \,|\, a_n : \diamond}$$

(LET)
$$\frac{\Gamma; \Lambda_1; \Sigma_1 \vdash a : \tau \qquad (\Gamma; \Lambda_2; \Sigma_2) \uplus \{x : \tau\} \vdash b : \tau'}{\Gamma; \Lambda_1 \uplus \Lambda_2; \Sigma_1 \fatsemi \Sigma_2 \vdash \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau'}$$

$$(\Gamma; \Lambda; \Sigma) \uplus \{u : \tau\} = \begin{cases} \Gamma, u : \tau; \Lambda; \Sigma & \text{if } \tau \equiv [\ldots], \\ \Gamma; \Lambda, u : \tau; \Sigma & \text{if } \tau \equiv \mathsf{lin}[\ldots], \\ \Gamma; \Lambda; \Sigma, u : \tau & \text{if } \tau \equiv \kappa. \end{cases}$$

(SUBSUME)
$$\frac{\Gamma; \Lambda; \Sigma \vdash a : \tau \qquad \Sigma \preccurlyeq \Sigma' \qquad \tau \preccurlyeq \tau'}{\Gamma; \Lambda; \Sigma' \vdash a : \tau'}$$

(OBJECT) $(\tau \equiv [l_i : (\sigma_i)\, \tau_i{}^{i \in I}])$
$$\frac{\Gamma, x_i : \tau; \emptyset; \{y_i : \sigma_i\} \vdash b_i : \tau_i \qquad \forall i \in I}{\Gamma; \emptyset; \emptyset \vdash [\tau \,|\, l_i = \varsigma(x_i)\, \lambda(y_i)\, b_i{}^{i \in I}] : \tau}$$

(LOBJECT) $(\tau \equiv \mathsf{lin}[l_i : (\sigma_i)\, \tau_i{}^{i \in I}], I \supset \emptyset)$
$$\frac{\Gamma; \Lambda; \left\{u_j : \kappa_j\right\}^{j \in J}, y_i : \sigma_i \vdash b_i : \tau_i \qquad \forall i \in I}{\Gamma; \Lambda; \left\{u_j : \kappa_j\right\}^{j \in J} \vdash [\tau \,|\, l_i = \lambda(y_i)\, b_i{}^{i \in I}] : \tau}$$

(INVOKE)
$$\frac{\Gamma; \Lambda; \Sigma \vdash w : \mathsf{lin}[l_i : (\sigma_i)\, \tau_i{}^{i \in I}] \qquad k \in I}{\Gamma; \Lambda; \Sigma, u : \sigma_k \vdash w.l_k \leftarrow u : \tau_k}$$

(UPDATE) $(\tau \equiv [l_i : (\sigma_i)\, \tau_i{}^{i \in I}])$
$$\frac{\Gamma; \emptyset; \emptyset \vdash w : \tau \qquad \Gamma, x : \tau; \emptyset; \{y : \sigma_k\} \vdash b : \tau_k \qquad k \in I}{\Gamma; \emptyset; \emptyset \vdash w.l_k \Leftarrow \varsigma(x)\, \lambda(y)\, b : \tau}$$

(SESSION)
$$\frac{\Gamma; \Lambda; \Sigma, x : \kappa_1, y : \kappa_2 \vdash b : \tau \qquad \kappa_1 \preccurlyeq \overline{\kappa_2}}{\Gamma; \Lambda; \Sigma \vdash \mathsf{let}\ (x, y) = \mathsf{new\,session}\langle \kappa_1, \kappa_2 \rangle\ \mathsf{in}\ b : \tau}$$

(SEQUNIT)
$$\frac{\Gamma; \Lambda; \Sigma \vdash a : \tau}{\Gamma; \Lambda; \Sigma, u : \varepsilon \vdash a : \tau}$$

(END)
$$\frac{}{\Gamma; \emptyset; \{u : \mathsf{end}\} \vdash \mathsf{close}(u) : [\,]}$$

(GET)
$$\frac{}{\Gamma; \emptyset; \{u : ?[\tau]\} \vdash u? : \tau}$$

(PUT)
$$\frac{\Gamma; \Lambda; \Sigma \vdash v : \tau}{\Gamma; \Lambda; \{u : ![\tau]\} \fatsemi \Sigma \vdash u!v : [\,]}$$

(SELECT) $(\sigma \equiv \oplus(X)[l_i : \sigma_i{}^{i \in I}])$
$$\frac{\Gamma; \Lambda; \Sigma, u : \sigma_k\{\!\{\sigma/X\}\!\} \vdash w.l_k \leftarrow u : \tau \qquad k \in I}{\Gamma; \Lambda; \Sigma, u : \sigma \vdash w.l_k \lhd u : \tau}$$

(BRANCH) $(\sigma \equiv \&(X)[l_i : \sigma_i{}^{i \in I}])$
$$\frac{\Gamma; \Lambda; \Sigma \vdash w : \mathsf{lin}[l_i : (\sigma_i\{\!\{\sigma/X\}\!\})\, [\ldots]_i{}^{i \in I}]}{\Gamma; \Lambda; \Sigma, u : \sigma \vdash u \rhd w : [\,]}$$

Figure 6.6: Typing

expressed by the empty linear environments. Linear objects are typed with (LOBJECT); there is no self in methods, however other linear objects and queues can be used. The restrictions are that: first, all free sessions must be complete; second, all methods must use the same linear objects and implement the same session type for free queues. The first condition allows a linear object to change thread; the second ensures that since a linear object is invoked exactly once, it will produce the same "side-effect" irrespective of which method is invoked.

Rule (INVOKE) types method invocation. Shared objects are linearised first with (SUBSUME). The corresponding usage for the actual queue argument is added in the final environments. Rule (UPDATE) is standard; the new method body is typed in the same way as a method in (OBJECT). See Example 6.3.5 (6). Note that linear objects cannot be updated or cloned.

Sessions Rule (SESSION) for new sessions has two constraints: first, the queues must be used completely and then closed, producing complete session types; second, the session types must agree with the annotations and these must be dual, ensuring compatibility of interactions. Subtyping may need to be used to achieve syntactic duality. Rule (SEQUNIT) is a limited form of weakening useful for type preservation: when one queue has been used and closed it may not appear free, hence will not be in Σ, but due to asynchrony the dual queue may still be present, albeit with empty remaining type (modulo buffered values); this rule recovers duality of both endpoints. In (END) closing a queue u produces the singleton Σ-environment $\{u : \mathsf{end}\}$ and gives the empty object type as result, to match reduction.

Rule (GET) for input records an input session part with the carried value type appearing as the result. Rule (PUT) for output $u!v$ first types the sent value v assigning to it a type τ with linear usage Λ and Σ, then in the conclusion an output session part is composed to the left (meaning it happens first) of Σ. When v is unrestricted $\Lambda = \Sigma = \emptyset$; when v is a linear object $\Lambda = \{v : \tau\}$ and $\Sigma = \emptyset$; when v is a queue $\Lambda = \emptyset$ and $\Sigma = \{v : \tau\}$ with $\tau = \kappa$. The rule accepts $s!s$, i.e., a queue that emits itself, giving $\{s : ![\kappa];\kappa\}$, which is correct because the rule composes the output type and the actual type of the value, which is used after that output.

Rule (SELECT) is used for a selection $w.l_k \triangleleft u$. In the premise we type the invocation $w.l_k \leftarrow u$ which gives a session environment $\Sigma, u : \sigma_k\{\!\{\sigma/X\}\!\}$. The notation $\sigma_k\{\!\{\sigma/X\}\!\}$ is a capture-avoiding type substitution in which the type σ replaces any instance of the type variable X in σ_k. The type σ is equal to $\oplus(X)[l_i : \sigma_i{}^{i\in I}]$ with $k \in I$, therefore, $\sigma_k\{\!\{\sigma/X\}\!\}$ is an *unfolding* of the k-th component of σ. Then, the select construct $w.l_k \triangleleft u$ types u with the *folding* σ of the unfolded types $\{\sigma_i\{\!\{\sigma/X\}\!\}^{i\in I}\}$ under selection. See Example 6.3.5 (7).

Rule (BRANCH) works similarly, but the constraints are slightly stronger: first, all mappings of label to argument types from the object type (which may be subsumed to lesser methods) are used in the fold; secondly, all methods used in the branching (but not any methods hidden by subsumption) must return a shared value of some object type, although these types can be different, because any of them can be chosen during reduction and we discard the actual value and assign the empty object type to the branch construct $u \triangleright w$. This is a design choice. Alternatively we could allow the type of w in the premise to be:

$$\mathsf{lin}[l_i : (\sigma_i\{\!\{\sigma/X\}\!\})\, \tau_i{}^{i \in I}]$$

allowing the type of $u \triangleright w$ in the conclusion to be τ with the additional constraint that $\forall i \in I \,.\, \tau_i \preccurlyeq \tau$. This would allow us to use the returned value within the program, however, it would make it more difficult to use an object in branching, because a subset of methods satisfying the additional constraint would have to be chosen. In the current formulation, even methods with arbitrary return types, as long as they are shared, can participate in the branching. In fact, both versions of the rule can be added to the system for maximum typability, but for simplicity we prefer to keep the most useful one for sessions in our present formulation.

Our approach is iso-recursive [35], which is a natural choice since we have coordinated object-based recursion on both endpoints of a session, and we can perform an implicit type folding at the points where recursion is decided, i.e., upon selection and branching.

6.3.5 Examples: Justification of Types and Typing Rules

In this subsection we justify the key conditions on types and typing rules with examples, demonstrating the subtle interplay between recursion, linearity of objects, sessions and concurrency.

1. Allowing the empty indexing set in the syntax of linear object types $\mathsf{lin}[l_i : (\sigma_i)\, \tau_i{}^{i \in I}]$, i.e., allowing empty linear objects, is problematic. For example consider the following:

   $$\begin{array}{ll} & \mathsf{let}\, x = [\,\tau_1 \,|\, l = \lambda(y)\, s_1 !\, v;\, \mathsf{close}(s_1)\,]\ \mathsf{in}\ s!x \\ | & \mathsf{let}\, x = [\,\tau_2 \,|\, l = \lambda(y)\, \mathsf{let}\, z = y?\ \mathsf{in}\ b\,]\ \mathsf{in}\ x.l \leftarrow \bar{s} \end{array}$$

   Assume we allowed a linear object to subsume the type $\mathsf{lin}[]$. Then the output $s!x$ would be typable with $\{s : ![\mathsf{lin}[]]; \mathsf{end}\}$. Take $\tau_2 \equiv \mathsf{lin}[\,l : (\,?[\mathsf{lin}[]]; \mathsf{end}\,)\, \tau_3\,]$. Then the invocation $x.l \leftarrow \bar{s}$ is typable. In the second object, the input $y?$ would give a value of type $\mathsf{lin}[]$ for z.

Since z has no method to invoke, it must either be re-sent or returned in b. Therefore, the enclosed queue s_1 will never be used, which is clearly undesirable.

2. We justify the key object subtyping rule, (SUB-LINEARISE). It is always safe to use a shared object linearly. However, the inverse is unsafe, for example it would allow:

$$\mathsf{let}\, x = [\,\tau \,|\, l{=}\lambda(y)\, s_1\,]\ \mathsf{in}\ []$$

which discards s_1, and:

$$\mathsf{let}\, x = [\,\tau \,|\, l{=}\lambda(y)\, s_1\,]\ \mathsf{in}\ x.l{\leftarrow}s;\, x.l{\leftarrow}s$$

in which s_1 is copied.

3. Using (SUB-ASYNCPUT), the term

$$s\,!\,v_1;\, s?;\, \mathsf{close}(s) \ |\ \bar{s}\,!\,v_2;\, \bar{s}?;\, \mathsf{close}(\bar{s})$$

is typable. One of the queue usages can be subsumed, say for s, obtaining:

$$\{s : ?[\tau_2];\, ![\tau_1];\mathsf{end},\ \bar{s} : ![\tau_2];\, ?[\tau_1];\mathsf{end}\}$$

which assigns syntactically dual types to the endpoints.

4. If we allowed partial session use in thread bodies, we would accept:

$$\mathsf{spawn}\ (s\,!\,v);\, \mathsf{close}(s) \ |\ \bar{s}?;\, \mathsf{close}(\bar{s})$$

which reduces (without showing the heap) to:

$$\mathsf{close}(s) \ |\ \bar{s}?;\, \mathsf{close}(\bar{s}) \ |\ s\,!\,v$$

where the order of the output and the session ending is non-deterministic, violating communication safety: it constitutes copying of s since in the result it cannot be composed to a sequential usage.

If we allowed a thread body to evaluate to a linear object, then we would accept:

$$\mathsf{spawn}\ [\tau \,|\, l = \lambda(x)\, s?;\, \mathsf{close}(s)] \mid \bar{s}!\, v;\, \mathsf{close}(\bar{s}), \quad B$$

which reduces to:

$$[\,] \mid \bar{s}!\, v;\, \mathsf{close}(\bar{s}) \mid o, \quad B \cdot o \mapsto [\tau \,|\, l = \lambda(x)\, s?;\, \mathsf{close}(s)]$$

in which s has been effectively discarded, because no term has access to o.

5. If we allowed aliasing of queues in (LET), by permitting a local definition to have a partial session type, we would type $\mathsf{let}\ x = s\ \mathsf{in}\ s!\, v_1; x!\, v_2$, which is unsafe as we would obtain a final Σ-environment $\{s : ![\tau_2]; ![\tau_1]\}$ when the true usage is $\{s : ![\tau_1]; ![\tau_2]\}$.

6. In (OBJECT) and (LOBJECT), the formal method argument can implement part of a session, for example:

$$\mathsf{let}\ z = [\tau \,|\, l = \varsigma(x)\, \lambda(y)\, y!\, v]\ \mathsf{in}\ (z.l \leftarrow s;\, z.l \leftarrow s)$$

can be typed with $\tau = [\,l : (![\tau_v])\,[\,]\,]$ and results in Σ-environment $\{s : ![\tau_v]; ![\tau_v]\}$. The advantage is modularity: an object does not need knowledge of the complete type of the session to which it will be applied (s in the example above). Also, a method with partial type argument is guaranteed not to send or spawn the session, as these require complete types.

7. As a simple example of recursive sessions typing using rule (SELECT) for internal choice, consider:

$$\mathsf{let}\ z = [\tau \,|\, l = \varsigma(x)\, \lambda(y)\, y!\, v;\, x.l \triangleleft y]\ \mathsf{in}\ z.l \triangleleft s$$

with $\tau \equiv [\,l : (\sigma)\,[\,]\,]$ and $\sigma = ![\tau_v]; \oplus(X)[l : ![\tau_v]; X]$. Omitting binders, the body of method l is $y!\, v; x.l \triangleleft y$ which needs to be typed with Σ-environment:

$$\{y : \sigma\} = \{y : ![\tau_v]\} \,;\, \{y : \oplus(X)[l : ![\tau_v]; X]\}$$

Self x has type τ therefore the invocation $x.l \leftarrow y$ assigns σ to y which then folds to $\oplus(X)[l : ![\tau_v]; X]$ in $x.l \triangleleft y$, producing the required typing. Outside of the object, in $z.l \triangleleft s$, the same process can type the selection, folding σ and giving, as desired, $\{s : \oplus(X)[l : ![\tau_v]; X]\}$ for the whole example.

8. If for a choice $\sigma = \oplus(X)[\,l_1 : X,\, l_2 : X\,]$ we have $\sigma \in \pi$ and $\sigma \in \kappa$ then the following term is

typed using (LET) and (SPAWN):

$$\begin{aligned}&\mathsf{let}\ x = [\tau \,|\, l_1 = \varsigma(x)\,\lambda(y)\,x.l_2 \triangleleft y\ , l_2 = \varsigma(x)\,\lambda(y)\,x.l_1 \triangleleft y]\\ &\quad \mathsf{in\ spawn}\ (x.l_1 \triangleleft s);\ s\,!\,v\end{aligned}$$

In the second line, the sequence composition $\sigma;\, ![\tau]$ succeeds since $\sigma \in \pi$, and (SPAWN) succeeds since $\sigma \in \kappa$, but the term reduces to an unsafe configuration with an output parallel to the spawned recursion. This justifies our restriction.

6.3.6 Session Types and Typing for the Instant Messenger

We can now give session types to the queues s_{usr} and s_{im} from the example of Figure 6.3:

$$\begin{aligned}\sigma_{\mathsf{usr}} &= \oplus(X)[arg : \&(Y)[exit : \mathsf{end}\,, msg : ![\tau_s];X\,, file : ![\tau_s];X\,]\,]\\ \sigma_{\mathsf{im}} &= \oplus(X)[arg : \oplus(Y)[exit : \mathsf{end}\,, msg : ![\tau_s];X\,, file : ![\sigma_{\mathsf{file}}];X\,]\,]\\ \sigma_{\mathsf{file}} &= ?[\tau_s];\,?[\tau_f];\mathsf{end}\end{aligned}$$

with τ_s being the type of strings and τ_f the type of files. Note that the above session types concisely abstract the structure of interaction and the behaviour of the program as types. Then under environments $\Gamma = \Lambda = \emptyset$ and $\Sigma = \{s_{\mathsf{usr}} : \sigma_{\mathsf{usr}}, s_{\mathsf{im}} : \sigma_{\mathsf{im}}\}$, by using rules (LET), (SELECT), (OBJECT), (LOBJECT), (BRANCH), (GET), (PUT), (END), (INVOKE), (SESSION), (SPAWN), and the axioms, the whole program of Figure 6.3 is typable.

6.4 Typing Runtime Terms

Session typing ensures first, that a queue behaves according to a session type, and secondly, that only dual queues with compatible types can interact in a session. The typing rules enforce linearity in the use of objects and queues obtaining a strong behavioural guarantee of non-interference.

The type soundness of the **session**ς-calculus is established by also typing the heap created during the execution of a well-typed initial program. In particular, we track the movement of linear objects and queues to and from the heap, to ensure that linearity is preserved, and we check that endpoints continue to have dual types after each use. To analyse the intermediate steps precisely, we utilise the following concepts:

- **Session Remainder:** We assign types to queues using *session remainders*, which are a subtraction of the type of the values stored in a queue from the complete session type of the queue. This technique is very similar to the one in [40], however in our approach asynchrony at the type level requires more elaboration.

- **Heap Types:** We assign types to heaps, understanding a heap type as a collection of mappings giving types to all of the heap components. Our formulation must take into account the possible circularities within a heap.

6.4.1 Session Remainder

Before we proceed to heap typing, it is necessary to explain how to obtain typings for a queue endpoint taking into consideration that values waiting to be received may be present at the queue. In Figure 6.7 we formalise the rules, using a new form of judgement taking the shape:

$$\Gamma;\Lambda;\Sigma \vdash \kappa - \vec{r} = \sigma$$

where κ is a complete session type from which we will “subtract” the values $\vec{r}$ of the queue to obtain the *session remainder* σ. By our rules, this remainder will either be ε meaning that the session has been completed and there is nothing more remaining, or a complete remaining session; if a partial session was the remainder, it would mean that the session will not be closed. The environments left of $\vdash$ accumulate the ones produced by typing each value in $\vec{r}$. As before, a judgement is *well-formed* when $\mathsf{dom}(\Gamma) \uplus \mathsf{dom}(\Lambda) \uplus \mathsf{dom}(\Sigma)$ is defined.

$$\text{(Q-ANY)}\quad \dfrac{}{\Gamma;\emptyset;\emptyset \vdash \kappa - \varepsilon = \kappa} \qquad\qquad \text{(Q-END)}\quad \dfrac{}{\Gamma;\emptyset;\emptyset \vdash \mathsf{end} - \mathsf{end} = \varepsilon}$$

$$\text{(Q-GET)}\quad \dfrac{\Gamma;\Lambda_1;\Sigma_1 \vdash v : \tau \qquad \Gamma;\Lambda_2;\Sigma_2 \vdash \kappa - \vec{r} = \sigma}{\Gamma;\Lambda_1 \uplus \Lambda_2;\Sigma_1 \uplus \Sigma_2 \vdash ?[\tau];\kappa - v\vec{r} = \sigma}$$

$$\text{(Q-BRANCH)}\quad (\sigma \equiv \&(X)[l_i : \sigma_i{}^{i\in I}])\quad \dfrac{\Gamma;\Lambda;\Sigma \vdash \sigma_k\{\!\{\sigma/X\}\!\};\sigma' - \vec{r} = \sigma'' \qquad k \in I}{\Gamma;\Lambda;\Sigma \vdash \sigma;\sigma' - l_k\vec{r} = \sigma''}$$

$$\text{(Q-PUT)}\quad \dfrac{\Gamma;\Lambda;\Sigma \vdash \kappa - \vec{r} = \kappa' \quad \vec{r} \neq \vec{r}_1\mathsf{end}}{\Gamma;\Lambda;\Sigma \vdash ![\tau];\kappa - \vec{r} = ![\tau];\kappa'}$$

$$\text{(Q-SELECT)}\quad \dfrac{\Gamma;\Lambda;\Sigma \vdash \sigma_i - \vec{r} = \sigma'_i \quad \vec{r} \neq \vec{r}_1\mathsf{end} \quad \forall i \in I}{\Gamma;\Lambda;\Sigma \vdash \oplus(X)[l_i : \sigma_i{}^{i\in I}];\sigma' - \vec{r} = \oplus(X)[l_i : \sigma'_i{}^{i\in I}];\sigma'}$$

Figure 6.7: Session Remainder

Rule (Q-ANY) defines that any session type agrees with the empty queue ε, producing the full type as remainder. In (Q-END) the type end agrees with the queue consisting the single value end, giving the empty session ε as remainder.

Rule (Q-GET) takes the input prefix of a session $?[\tau];\kappa$ and the first value from the queue $v\vec{r}$, and types (with the usual rules) the value assigning τ, to match the input part. The remaining type κ and queue $\vec{r}$ are subtracted to obtain the final remainder. Disjoint composition in the conclusion's environments ensures that if v is linear then it only appears once in the queue, since either $\Lambda_1 = \{v : \tau\}$ or $\Sigma_1 = \{v : \tau\}$.

Rule (Q-BRANCH) matches a branching type σ at the beginning of a session $\sigma;\kappa$ with a label l_k at the top of the queue $l_k\vec{r}$. The remainder is obtained by the subtraction $\sigma_k\{\!\{\sigma/X\}\!\};\kappa - \vec{r}$ where the label has been used to unfold the k-th component of σ.

Rules for Asynchrony Due to asynchronous subtyping, a process that, according to its type, is due to perform an output or selection, may in fact have values in its input queue, waiting to be received. This is because the dually typed process in the same session, which would normally be expected to wait on in input/branching, may be more asynchronous, and perform output actions in advance. Thus, when calculating the remainder of a session, we need to allow this possibility; this is formalised in the following rules.

Rule (Q-PUT) disregards the output prefix of the session type $![\tau];\kappa$ and calculates the remainder κ' of $\kappa - \vec{r}$, which then appears in the conclusion prefixed with the original output giving $![\tau];\kappa'$. Therefore the output is not consumed, which is correct, since we are subtracting the value types of some queue s from the input/branching components of the type for the same s. This rule relates to the subtyping rule (SUB-ASYNCPUT) of Figure 6.5, which allows a term to perform an output when a sequence of inputs/branchings is expected, depending on the shape of a context $\mathcal{B}_X$ that may have been pushed in the type κ from the outer context of a supertype. If we had not admitted this subtyping rule, the case of output would be subsumed by (Q-ANY), because there would be no subtypes that send values "in advance" and hence, no values left to be received when an output is due. So, for example, $![\tau_1];?[\tau_2];?[\tau_3];\mathsf{end} - v$ where v is typed with τ_2 gives remainder $![\tau_1];?[\tau_3];\mathsf{end}$. We do not allow κ, the rest of the session, to eventually close (as it would if (Q-END) was used in the premise) giving final remainder $![\tau];\varepsilon = ![\tau]$; this would mean that a session can be closed before an output is performed.

Rule (Q-SELECT) has a similar function. Any values in the queue are subtracted from each of the branches inside the selection, resulting in a possibly smaller remainder type. The rule relates to subtyping with (SUB-ASYNCSEL) of Figure 6.5, which allows a term to perform a selection when a sequence of inputs/branchings is expected. Notice that the sequence composed type σ'

may not be a complete type, since the branch could be recursive and complete itself. In this case we can take $\sigma' = \varepsilon$, which leaves us with the selection since ε is the unit of sequence composition, and can be removed.

6.4.2 Heap Typing

We assign types to heaps: a heap typing is a collection of mappings giving types to all of the heap's components, in the form of three typing environments. The new typing judgement is:

$$\Gamma_1;\Lambda_1;\Sigma_1 \vdash B : \langle \Gamma_2;\Lambda_2;\Sigma_2 \rangle$$

where $\langle \Gamma_2;\Lambda_2;\Sigma_2 \rangle$ assign types to the elements in the domain of heap B; we call this the *heap type*. Objects and queues in the codomain of the heap contain references to other objects and queues in the heap, and these assumptions are shown on the left of $\vdash$, as $\Gamma_1;\Lambda_1;\Sigma_1$; we call these *heap assumptions*. A judgement is *well-formed* when $\mathsf{dom}(\Gamma_1) \uplus \mathsf{dom}(\Lambda_1) \uplus \mathsf{dom}(\Sigma_1)$ and $\mathsf{dom}(\Gamma_2) \uplus \mathsf{dom}(\Lambda_2) \uplus \mathsf{dom}(\Sigma_2)$ are defined.

Not every heap typing is meaningful for type safety, where we need that the subset of the heap domain that appears on the left of $\vdash$ is given the same types also in the full heap type. The inconsistency is necessary for intermediate typings, due to circularities preventing a purely inductive definition. Final typings referring to the whole of a heap, used in theorems, will be of the shape:

$$\Gamma_1;\Lambda_1;\Sigma_1 \vdash B : \langle \Gamma_1;\Lambda_1 \uplus \Lambda_2;\Sigma_1 \uplus \Sigma_2 \rangle$$

We call such heap typings *consistent*.

The heap typing rules are shown in Figure 6.8. Rule (B-EMPTY) is for typing the empty heap, requiring no assumptions except for Γ which is arbitrary.

Rule (B-SHARED) types a heap with a mapping $o \mapsto c$ for an unrestricted object c. The object c is typed (with the rules of Figure 6.6) giving empty linear environments and type τ; the rest of the heap is typed giving assumptions Γ_1, Λ_1, Σ_1 which are propagated in the conclusion, along with heap typing Γ, Λ, Σ which becomes extended with $\{o : \tau\}$. The disjoint union corresponds to the fact that each mapping in the heap is unique, that is, B cannot contain another mapping for o.

Rule (B-LINEAR) is for a linear object mapping in the heap; it works similarly to the one for unrestricted objects, however here the assumptions in the conclusion are extended with the linear environments induced from typing the object. The rule requires, by the disjointness of the assumptions in the conclusion, that linear components in c do not also appear inside some

(B-EMPTY)

$$\frac{}{\Gamma;\emptyset;\emptyset \vdash \emptyset : \langle\emptyset;\emptyset;\emptyset\rangle}$$

(B-SHARED) $(c \equiv [\tau' \mid l_i{=}\varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}])$

$$\frac{\Gamma_1;\Lambda_1;\Sigma_1 \vdash B : \langle\Gamma;\Lambda;\Sigma\rangle \qquad \Gamma_1;\emptyset;\emptyset \vdash c : \tau \qquad \tau' \preccurlyeq \tau}{\Gamma_1;\Lambda_1;\Sigma_1 \vdash B\cdot o \mapsto c : \langle\Gamma \uplus \{o:\tau\};\Lambda;\Sigma\rangle}$$

(B-LINEAR) $(c \equiv [\tau' \mid l_i{=}\lambda(y_i)\,b_i{}^{i\in I}])$

$$\frac{\Gamma_1;\Lambda_1;\Sigma_1 \vdash B : \langle\Gamma;\Lambda;\Sigma\rangle \qquad \Gamma_1;\Lambda_2;\Sigma_2 \vdash c : \tau \qquad \tau' \preccurlyeq \tau}{\Gamma_1;\biguplus\Lambda_{1,2};\biguplus\Sigma_{1,2} \vdash B\cdot o \mapsto c : \langle\Gamma;\Lambda \uplus \{o:\tau\};\Sigma\rangle}$$

(B-SESSION)

$$\frac{\begin{array}{c}\Gamma_1;\Lambda_1;\Sigma_1 \vdash B : \langle\Gamma;\Lambda;\Sigma\rangle \\ \Gamma_1;\Lambda_2;\Sigma_2 \vdash \kappa_1 - \vec{r_1} = \sigma_1 \qquad \Gamma_1;\Lambda_3;\Sigma_3 \vdash \kappa_2 - \vec{r_2} = \sigma_2 \qquad \sigma_1 \preccurlyeq \overline{\sigma_2}\end{array}}{\Gamma_1;\biguplus\Lambda_{1,2,3};\biguplus\Sigma_{1,2,3} \vdash B\cdot s \mapsto \vec{r_1}\cdot \bar{s} \mapsto \vec{r_2} : \langle\Gamma;\Lambda;\Sigma \uplus \{s:\kappa_1, \bar{s}:\kappa_2\}\rangle}$$

Figure 6.8: Heap Typing

other object or queue in the heap. Finally, the heap typing is extended as before, but at the Λ-environment.

Rule (B-SESSION) is for typing the two (dual) endpoint queues of a session. Types κ_1 and κ_2 can be given to the queues s and $\bar{s}$ when the session remainders σ_1 and σ_2 of $\kappa_1 - \vec{r_1}$ and $\kappa_2 - \vec{r_2}$ are dual session types; more precisely, σ_1 must be a subtype of the dual of σ_2, written $\sigma_1 \preccurlyeq \overline{\sigma_2}$. This also implies $\sigma_2 \preccurlyeq \overline{\sigma_1}$. The linear assumptions needed for typing the values in $\vec{r_1}$ and $\vec{r_2}$ are added to the heap assumptions, ensuring that everything is used exactly once; the mappings $\{s:\kappa_1, \bar{s}:\kappa_2\}$ are added to the heap type.

6.5 Type Soundness

We first formalise the necessary lemmas for type soundness. In the following, we write E for closed reduction contexts, and refer to runtime types ψ defined as:

$$\psi \ ::= \ \tau \mid \diamond$$

First we have a standard lemma for weakening and strengthening of the assumptions in the unrestricted environment.

Lemma 6.5.1. (Γ Weakening, Strengthening)

(a) If $\Gamma;\Lambda;\Sigma \vdash E : \psi$ *and* $u \notin \mathsf{dom}(\Gamma,\Lambda,\Sigma)$ *and* $\tau \equiv [l_i : (\sigma_i)\,\tau_i{}^{i\in I}]$ *then* $\Gamma, u:\tau;\Lambda;\Sigma \vdash E : \psi$.

(b) *If* $\Gamma;\Lambda;\Sigma \vdash E : \psi$ *and* $u \notin \mathsf{fv}(E)$ *then* $\Gamma \setminus u;\Lambda;\Sigma \vdash E : \psi$.

Proof. By induction on the last typing rule applied. □

Note that weakening and strengthening are not allowed for the linear environments. Next, we formalise the substitution lemma. Notice the difference between (a) unrestricted and (b) linear objects.

Lemma 6.5.2. (Substitution) *For* $o, s \notin \mathsf{fv}(a)$*:*

(a) *If* $\Gamma, x : \tau_1;\Lambda;\Sigma \vdash a : \tau$ *and* $\Gamma;\emptyset;\emptyset \vdash o : \tau_1$ *then* $\Gamma;\Lambda;\Sigma \vdash a\{{}^{o}\!/\!{}_{x}\} : \tau$.

(b) *If* $\Gamma;\Lambda, x : \tau_2;\Sigma \vdash a : \tau$ *and* $\Gamma;\{o : \tau_1\};\emptyset \vdash o : \tau_2$ *then* $\Gamma;\Lambda, o : \tau_1;\Sigma \vdash a\{{}^{o}\!/\!{}_{x}\} : \tau$.

(c) *If* $\Gamma;\Lambda;\Sigma, x : \sigma \vdash a : \tau$ *then* $\Gamma;\Lambda;\Sigma, s : \sigma \vdash a\{{}^{s}\!/\!{}_{x}\} : \tau$.

Proof. By induction on the last typing rule applied. □

The following lemma is used to obtain typings for subterms that are in redex position within a reduction context. For example, this lemma is used for the case of sending a message to a queue as well as the case for spawn.

Lemma 6.5.3. (Subderivation) *If* $\Gamma;\Lambda;\Sigma \vdash E\langle a\rangle : \psi$ *then there exist* $\Lambda_{1,2}, \Sigma_{1,2}, \tau$ *s.t.* $\Lambda = \Lambda_1 \uplus \Lambda_2$ *and* $\Sigma = \Sigma_1 \fatsemi \Sigma_2$ *and* $\Gamma;\Lambda_1;\Sigma_1 \vdash a : \tau$.

Proof. By induction on E. Note that if E is a parallel context then we use the fact $\Sigma_1 \fatsemi \Sigma_2 = \Sigma_1 \uplus \Sigma_2$. □

Using the following lemma, we can obtain new linear environments after a subterm has been replaced within a reduction context.

Lemma 6.5.4. (Context Replacement)
If $\Gamma;\Lambda_1 \uplus \Lambda_2;\Sigma_1 \fatsemi \Sigma_2 \vdash E\langle a\rangle : \psi$ *and* $\Gamma;\Lambda_1;\Sigma_1 \vdash a : \tau$ *and* $\Gamma, \Gamma';\Lambda'_1;\Sigma'_1 \vdash a' : \tau$ *and* $\Lambda'_1 \uplus \Lambda_2$ *defined and* $\Sigma'_1 \fatsemi \Sigma_2$ *defined then* $\Gamma, \Gamma';\Lambda'_1 \uplus \Lambda_2;\Sigma'_1 \fatsemi \Sigma_2 \vdash E\langle a'\rangle : \psi$.

Proof. By induction on E. □

We now proceed to type soundness and communication safety. The essence of our typing is that all linear elements are either used or remain accessible (for later use) through the resulting term, either directly, or indirectly through linear components in the heap. Moreover, sessions must be used respecting types. Hence the following theorem subsumes type soundness and communication safety in the sense of [48]. The conditions are formulated in two clauses: (a) for all reductions except thread creation, and (b) for a reduction that generates a thread.

Theorem 6.5.5. (Subject Reduction)
If $\Gamma_1;\Lambda_1;\Sigma_1 \vdash E : \psi$ *and* $\Gamma_1;\Lambda_2;\Sigma_2 \vdash B : \langle\Gamma_1;\Lambda_1 \uplus \Lambda_2;\Sigma_1 \uplus \Sigma_2\rangle$ *and* $E,B \rightarrow E',B'$ *, using*

(a) any rule except (R-SPAWN)*, then there exist* $\Gamma_2,\Lambda_{3,4},\Sigma_{3,4}$ *s.t.* $\Gamma_2;\Lambda_3;\Sigma_3 \vdash E' : \psi$ *and* $\Gamma_2;\Lambda_4;\Sigma_4 \vdash B' : \langle\Gamma_2;\Lambda_3 \uplus \Lambda_4;\Sigma_3 \uplus \Sigma_4\rangle$.

(b) (R-SPAWN) *with* $E \equiv E_1\langle\mathsf{spawn}\ a\rangle$ *then there exist* $\Lambda_{11,12},\Sigma_{11,12}$ *s.t.* $\Lambda_1 = \Lambda_{11} \uplus \Lambda_{12}$ *and* $\Sigma_1 = \Sigma_{11} \mathbin{;} \Sigma_{12} = \Sigma_{11} \uplus \Sigma_{12}$ *and* $\Gamma_1;\Lambda_{11};\Sigma_{11} \vdash a : []$ *and* $\Gamma_1;\Lambda_{12};\Sigma_{12} \vdash E_1\langle[]\rangle : \psi$ *and* $\Gamma_1;\Lambda_1;\Sigma_1 \vdash E_1\langle[]\rangle \,|\, a : \diamond$.

Proof By induction on the last reduction applied. For the first part of each case, we obtain that the produced term has the same type as the original, using a straightforward combination of Weakening (6.5.1), Substitution (6.5.2), Subderivation (6.5.3), and Context Replacement (6.5.4).

Obtaining a matching heap typing that is consistent is more delicate. For the rules (R-SPAWN), (R-LET), (R-CLONE), (R-INVOKE), (R-UPDATE) and (R-OBJECT) (for the unrestricted case) the heap typing is either unchanged or trivially obtainable, since there are no linear elements moved from the term to the heap or from the heap to the term. Similarly for the rules (R-TCLEANL,R).

For linear objects in (R-OBJECT) we obtain a new heap judgement where any linear components in the body of the object appear, after reduction, on the heap typing assumptions, and the heap type is extended using (B-LINEAR). Dually, for the linear invocation (R-LINVOKE) the heap typing decreases, and the linear assumptions from typing the object body move to the term typing where the method body is copied; for this it is crucial that the linear object is removed from the heap.

The rules for sessions are more interesting. For new sessions in (R-SESSION) a consistent heap typing is obtained since the types for each endpoint are initially dual, and there are no values in the queues, resulting trivially in dual remainders. Rule (R-QCLEAN) is also straightforward. For the case of input in (R-GET) a straightforward inspection of the premises in (Q-GET) gives the remaining session; then, if the received value is linear, its singleton environment obtained by (LOBJVAL) or (QUEUEVAL) is moved to the term typing, using the Substitution Lemma (6.5.2). In (R-SELECT) the selection becomes invocation, resulting in an unfolding on the type of the argument; then, in the remainder typing for the other endpoint, using (Q-BRANCH), the received label will be used for unfolding giving, as desired, dual remaining types for both queues. Rule

(R-BRANCH) is similar: the label causing the unfolding (in the queue typing) is consumed from the queue, but the resulting invocation re-introduces the unfolding therefore leaving the type intact. For (R-END) in $\mathsf{close}(s)$ a direct use of session remainder gives ε for $\bar{s}$ (which is where the end is appended), and the same is obtained using (SEQUNIT) for s (which does not appear anywhere after being closed).

The most delicate case is that of output in (R-PUT), where the sent value can be linear, and the type of the sending endpoint changes; we prove this case below:

Case (R-PUT) $\quad E = E_1\langle s!\,v\rangle \quad B = B_1 \cdot s \mapsto \vec{r}_1 \cdot \bar{s} \mapsto \vec{r}_2$

$\quad E' = E_1\langle [\,]\rangle \quad B' = B_1 \cdot s \mapsto \vec{r}_1 \cdot \bar{s} \mapsto \vec{r}_2 v$

By Lemma 6.5.3 we obtain $\Gamma;\Lambda_{11};\Sigma_{11} \vdash s!\,v : \tau_1$ with $\Lambda_1 = \Lambda_{11} \uplus \Lambda_{22}$ and $\Sigma_1 = \Sigma_{11} \fatsemi \Sigma_{22}$. By (PUT) we have $\Gamma;\Lambda_{11};\Sigma'_{11} \vdash v : \tau$ and $\Sigma_{11} = \{s : ![\tau]\} \fatsemi \Sigma'_{11}$ and $\tau_1 = [\,]$. By (OBJECT) and Lemma 6.5.4 we obtain $\Gamma;\Lambda_{12};\Sigma_{12} \vdash E' : \psi$ as desired. We now check the heap. By (B-SESSION) on the assumption we obtain the premises:

$$\Gamma;\Lambda_{21};\Sigma_{21} \vdash B_1 : \langle \Gamma_1;\Lambda_1 \uplus \Lambda_2;\Sigma'\rangle \tag{6.1}$$

$$\Gamma;\Lambda_{22};\Sigma_{22} \vdash \kappa_1 - \vec{r}_1 = \sigma_1 \tag{6.2}$$

$$\Gamma;\Lambda_{23};\Sigma_{23} \vdash \kappa_2 - \vec{r}_2 = \sigma_2 \tag{6.3}$$

with $\Sigma_1 \uplus \Sigma_2 = \Sigma' \uplus \{s : \kappa_1, \bar{s} : \kappa_2\}$ and $\Lambda_2 = \Lambda_{21} \uplus \Lambda_{22} \uplus \Lambda_{23}$ and $\Sigma_2 = \Sigma_{21} \uplus \Sigma_{22} \uplus \Sigma_{23}$ and $\sigma_1 \preccurlyeq \overline{\sigma_2}$. We have $\Sigma_1(s) = \kappa_1$ and:

$$(\Sigma'_{11} \fatsemi \Sigma_{12})(s) = \kappa'_1 \tag{6.4}$$

with $\kappa_1 = ![\tau];\kappa'_1$ by definition of ($\fatsemi$). By (Q-PUT) $\sigma_1 = ![\tau];\sigma'_1$ and:

$$\Gamma;\Lambda_{22};\Sigma_{22} \vdash \kappa'_1 - \vec{r}_1 = \sigma'_1 \tag{6.5}$$

By $\sigma_1 \preccurlyeq \overline{\sigma_2}$ we have $\sigma_2 \preccurlyeq \overline{\sigma_1}$, therefore $\sigma_2 \preccurlyeq ?[\tau];\overline{\sigma'_1}$. Applying a sequence of (Q-PUT)/(Q-SELECT) as required, followed by (Q-GET) once we reach the input (only outputs/selections can appear before this input in σ_2), we obtain:

$$\Gamma;\Lambda_{11};\Sigma'_{11} \vdash \sigma_2 - v = \sigma'_2 \tag{6.6}$$

with $\sigma'_2 \preccurlyeq \overline{\sigma'_1}$ which is equivalent to $\sigma'_1 \preccurlyeq \overline{\sigma'_2}$. From (6.3) and (6.6) we obtain:

$$\Gamma;\Lambda_{23} \uplus \Lambda_{11};\Sigma_{23} \uplus \Sigma'_{11} \vdash \kappa_2 - \vec{r}_2 v = \sigma'_2 \tag{6.7}$$

where $\Lambda_{23} \uplus \Lambda_{11}$ and $\Sigma_{23} \uplus \Sigma'_{11}$ are defined by the assumptions. Using (B-SESSION) with (6.1) and (6.5) and (6.7) we obtain:

$$\Gamma;\Lambda_{11} \uplus \Lambda_2;\Sigma'_{11} \uplus \Sigma_2 \vdash B' : \langle \Gamma_1;\Lambda_1 \uplus \Lambda_2;\Sigma' \uplus \{s : \kappa'_1, \bar{s} : \kappa_2\}\rangle$$

which must be shown *consistent*. The type of $\bar{s}$ does not change from the original typing (only its remainder changes). For s, we have from (6.4) that $(\Sigma'_{11} \fatsemi \Sigma_{12})(s) = \kappa'_1$ and by the condition in (PUT) that sent queues are complete we have either $\Sigma'_{11} = \{s : \kappa'_1\}$, if $v = s$, or $\Sigma_{12} = \Sigma'_{12}, s : \kappa'_1$, if $v \neq s$. Therefore the heap assumptions agree with the new heap type; the heap judgement is consistent. □

We now formalise communication-safety (which also subsumes type-safety for objects). An *s-input* is a term of the shape $s?$ or $s \triangleright o$. An *s-output* is a term $s!\,v$ or $o.l \triangleleft s$. An *s-close* is a term $\mathsf{close}(s)$. Finally, an *s-method* is a term $o.l \leftarrow s$ and an *s-occurrence* is the term s. Then, an *s-action* is an *s-input*, *s-output*, *s-close*, *s-method*, or *s-occurrence*. For object constructs, we define an *o-invocation* to be a term of the shape $o.l \leftarrow s$, an *o-update* is a term $o.l \Leftarrow \varsigma(x)\,\lambda(y)\,b$, an *o-selection* is a term $o.l \triangleleft s$ and an *o-branch* has the shape $s \triangleright o$. We also define an *o-occurrence* to be the term o. Then an *o-action* is an *o-invocation*, *o-update*, *o-selection*, *o-branch*, or *o-occurrence*. We can now define *error* configurations:

Definition 6.5.6 (Error Configurations). We say that a configuration E,B is an *error* if:

(a) $E = E_1\langle a\rangle$ and $E = E_2\langle b\rangle$ where $E_1 \neq E_2$ and both a and b are *s-action*s.

(b) $E = E_1\langle a\rangle$, a is a *s-input*, and $B = B' \cdot s \mapsto r\vec{r}$ such that (i) $a = s?$ and $r \in \{l, \mathsf{end}\}$, or (ii) $a = s \triangleright o$ and $r \in \{v, \mathsf{end}\}$.

(c) $E = E_1\langle a\rangle$ and $E = E_2\langle b\rangle$ where $E_1 \neq E_2$ and both a and b are *o-action*s and $B(o) = [\tau \,|\, l_i = \lambda(y_i)\,b_i{}^{i\in I}]$.

(d) $E = E_1\langle a\rangle$, and $a = o.l_k \leftarrow s$ or $a = o.l_k \triangleleft s$ or $a = o.l_k \Leftarrow \varsigma(x)\,\lambda(y)\,b$, with $B(o) = [\tau \,|\, l_i = \varsigma(x_i)\,\lambda(y_i)\,b_i{}^{i\in I}]$ or $B(o) = [\tau \,|\, l_i = \lambda(y_i)\,b_i{}^{i\in I}]$ and $k \notin I$.

(e) $E = E_1\langle o.l \Leftarrow \varsigma(x)\,\lambda(y)\,b\rangle$ with $B(o) = [\tau \,|\, l_i = \lambda(y_i)\,b_i{}^{i\in I}]$.

The above says that a configuration is an error if (a) it breaks the linearity of s by having e.g. two *s-inputs* in parallel, or (c) of objects by having two usages of a linear object in parallel; (b) there is communication-mismatch; (d) a method is chosen for some operation but it is not defined in the corresponding object; or (e) a linear object is updated. As a corollary of Theorem 6.5.5, we achieve the following general communication-safety theorem.

Theorem 6.5.7 (**Communication Safety**). *A configuration E, B such that $\Gamma_1;\Lambda_1;\Sigma_1 \vdash E : \psi$ and $\Gamma_1;\Lambda_2;\Sigma_2 \vdash B : \langle \Gamma_1;\Lambda_1 \uplus \Lambda_2;\Sigma_1 \uplus \Sigma_2 \rangle$ never reduces into an* error.

Proof The formalisation of communication safety is defined similarly to Theorem 4.4.14 for HO$\pi^{\mathbf{s}}$ and Theorem 5.5.11 for HO$\pi^{\mathbf{as}}$. The *balanced* condition on environments defined in the previous chapters has a correspondence in this system through the rule (B-SESSION), which therefore ensures that in the configuration under consideration the sessions are used dually. Thus, the strategy for the proof of this theorem in **session**ς is very similar to the previous ones, utilising Subject Reduction (Theorem 6.5.5) and showing by contradiction that error processes cannot arise.

6.6 Notes on Related Work

Object Calculi The untyped imperative object calculus **imp**ς and the first-order system $\mathbf{Ob}_{\mathbf{1}<:}$ of Abadi and Cardelli [2] formed the foundation for the object fragment of our calculus. A notable extension in our work is method arguments. In **imp**ς, methods with arguments can be encoded using the state of an object, but in **session**ς we cannot perform this encoding, because queues are linear and cannot be aliased.

Another related work is the **conc**ς-calculus of Gordon and Hankin [43]. Their work extends the **imp**ς-calculus with parallel composition and mutexes for synchronisation. Their mutexes could be added to our system, although we already achieve a different style of non-interference using the session primitives.

The recent work by Gay *et al.* [41] offers an alternative to our branching sessions: our branching labels coincide with method names; their branching uses instances of *enumerations* corresponding to typestates.

The closest work to ours, in terms of approach to sessions, is that of Drossopoulou, Dezani, and Coppo [33], in which sessions are implemented as methods. In particular, their main session primitive is written $a.l\{b\}$, which behaves as follows: the code of the method $a.l$ is placed in parallel to b, and a fresh channel $\mathbf{k}$ is used for communication between the two threads. Thus, the reduction has the shape $(\nu\mathbf{k})\ (c\{^{\mathbf{k}}/_{\bullet}\}\,|\,b\{^{\mathbf{k}}/_{\bullet}\})$, where c is the code of $a.l$, and $\bullet$ denotes the local placeholder for the channel. Note that this system only allows one session per scope. We can encode their main session primitive in **session**ς as:

$$a.l\{b\} \quad \overset{\text{def}}{=} \quad \mathsf{let}\ z = a\ \mathsf{in}\ \mathsf{let}\ (x,y) = \mathsf{new\,session}\langle\kappa_1,\kappa_2\rangle\ \mathsf{in}\ (\mathsf{spawn}\ z.l \Leftarrow x);\, b$$

with y used in b.

6.7 Concluding Remarks

We presented **session**ς, a small but powerful calculus featuring mutable objects, local definitions, concurrency, and type-safe asynchronous communication primitives. Importantly, our amalgamation of session and object recursion enables complex interaction patterns to be embedded in the structure of objects, creating a natural and concise programming paradigm for sessions.

The calculus enables the programming of elaborate interacting software within a framework of determinism, induced by *linearity* and *type duality*. The ability to utilise higher-order code in the form of shared and linear objects provides powerful programming idioms with a high degree of encapsulation.

The **session**ς-calculus has the *Subject Reduction* property, and *Communication Safety* obtained through careful analysis of intermediate steps and movements of linear values between a running program and the store.

Our session object calculus can serve as a basis for the addition of other features and type analyses, such as recursive self types, parametric polymorphism and variance annotations, to name a few. Our ability to recurse on methods with session argument addresses the question of how to extend a standard object-oriented language uniformly to deal with session and non-session arguments transparently. Moreover, the iso-recursive approach is easier to implement than the equi-recursive which requires coinductive definitions. The asynchronous buffered semantics offer a direct implementation strategy, and the careful removal of used linear values from the heap means that garbage collection is straightforward for our linear structures.

Part III

Conclusion & Future Directions

7 Conclusions, Open Questions and Future Work

Overview *We summarise the contributions of the thesis, and then focus on future work. There are a number of milestones that need to be achieved in order to prepare the foundational session typing theories developed in the previous part, for practical applications. The greatest emphasis is placed on the development of an algorithm for asynchronous coinductive subtyping. Then, we proceed to describe other desirable extensions for both processes and objects, including an evaluation of different methods for the algorithmic typing of objects. We finish by suggesting interesting implementation projects, such as a session typing system for the language Erlang.*

7.1 Summary of Contributions in this Thesis

This thesis presents a theoretical framework for structured communication-oriented programming, in the form of a family of typed programming calculi. In particular, the languages we defined encompass the fundamental constructions of communicating *processes*, *functions* and *objects*, facilitating a wide range of powerful programming idioms. Thus, our work provides a foundation for programming language design integrating multi-paradigm components with session-oriented communications, in a typesafe way.

The Languages

We presented three session-typed calculi:

- $\mathrm{HO}\pi^{\mathrm{s}}$ is a process algebra that extends the $\mathrm{HO}\pi$ (Higher-order π-calculus) with a session typing discipline. This language supports synchronous communication and enables us to represent mobile-code as the communication of functions, which may furthermore contain free instances of typed channels, necessitating a method to control their use. Our system extends and subsumes the existing formalisations for π-calculus, and can therefore serve as a unified theory.

- HO$\pi^{\mathbf{as}}$ is an asynchronous adaptation of HO$\pi^{\mathbf{s}}$ where messages are buffered in queues. Because of the asynchronous semantics we relaxed the conditions of type-safe composition of communicating processes, introducing a notion of *asynchronous subtyping* which is reified through a coinductive subtyping relation between session types corresponding to what is intuitively understood as more asynchronous behaviour. With this system we contributed a theory that takes advantage of the asynchrony induced by the use of buffered messaging, which is the norm in most application domains including the internet in general, and thus offer a refined basis for more flexible session-oriented programming in the presence of code mobility.

- **session**ς is a modification of the Abadi and Cardelli imperative object calculus **imp**ς, and integrates the code organisation of objects into process-oriented programming. In particular the structure of an object coincides with that of the protocols in which it can be used, allowing a concise formalisation where session control flow is merged into the fundamental abilities of objects. Session branching is implemented through associating an endpoint with an object that waits for a choice, or method, and selection corresponds to choosing a method on a branched object. Infinite protocols are realised directly through object self-secursion. Moreover, the creation of sessions is localised, providing an approach more suitable to an object-centric programming style, avoiding synchronised session initiation primitives. Since the language is buffered, we introduce asynchronous subtyping, but following an inductive approach which is more suitable in the context of the object calculus. Finally, the notions of object subtyping and branching subtyping are unified, yielding a natural integration of sessions and objects at the typing level. This object calculus is suitable for understanding and designing languages with structural and also class-based types, providing a unified foundation for a wide range of further development.

Key Ideas and Technical Contributions

The key original technical points are as follows:

- HO$\pi^{\mathbf{s}}$: The treatment of mobile code (sent functions) as *linear* components solved the problems that arise from the uncontrolled usage, and thus from the possible copying or vanishing of processes that implement sessions, which clearly compromises type safety of protocols.

- HO$\pi^{\mathbf{as}}$: The equi-recursive asynchronous subtyping defined constitutes a sound theoretical justification for the reordering message optimisations that can be accepted within type safe session programming. Technically, the coinductive formalisation of subtyping, and the subsequent proof, were more involved than expected and required more elaborate and stratified mathematical constructions.

- **sessionς**: The realisation of session control flow as a direct function of object control flow offers a new intuition on session objects, and in particular a way to utilise objects for the structural and behavioural organisation of process-oriented code. This integration of the object and session process paradigms is original, and can inform the development of practical languages, considering the acknowledged structuring benefits of objects. Moreover, the inductive, iso-recursive approach to asynchronous subtyping, provides a less powerful but simpler and more tractable technique, which is also uniform with the techniques for recursive objects in the original **impς** calculus.

We now discuss in some detail the future work we intend to undertake, focusing on the required results that facilitate practical implementations, under the challenging context of asynchronous subtyping.

7.2 Towards an Algorithm for Coinductive Asynchronous Subtyping

The coinductive method offers an attractive mathematical framework, although in its generality it does not specify *how* to prove (or disprove) a subtyping hypothesis. More specifically, it does not offer a way to obtain a simulation $\mathfrak{R} \subseteq \leqslant_c$, if there is one, that contains a desired hypothesis $(S_1, S_2) \in \mathfrak{R}$; it merely accepts or rejects a solution which is given by us, playing the role of an oracle. The question of how to obtain a (preferably *minimal*) simulation, failing if there is none, is what an *algorithmic subtyping* method is called to answer. In such a system, the coinductive relation $\leqslant_c$ is replaced by a set of rules implementing the algorithmic subtyping $\leqslant$, in judgements of the shape $\Sigma \vdash S_1 \leqslant S_2$. The environment Σ is used to collect checked hypotheses, and $S_1 \leqslant S_2$ is the *goal*. Then, $\leqslant$ must typically be shown to be sound and complete with respect to $\leqslant_c$, but for a practical implementation there is a prerequisite of decidability and, if possible, efficiency. Decidability depends on a number of key conditions: the number of possible goals that may need to be asserted in a single derivation must be finite; goals must not be checked repeatedly; the subtyping relation should be invertible, enabling the "guessing" of the shape of the subtype based on a known supertype. Typically the first two conditions are used to construct a well-founded measure that guarantees the termination of the algorithm. The last condition relates to computational efficiency; we return to this later.

A Digression on Termination

Now we turn our attention to the asynchronous subtyping relation $\leqslant_c$, which has a distinguishing feature compared to the standard simulations of Gay and Hole [39] and of Pierce and Sangiorgi [77]: some subtypings are only proved with infinite simulations. One such example is $\mu\mathbf{t}.![U_1].\mathbf{t} \leqslant_c \mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$ from § 5.3.1. A straightforward adaptation of the standard works (mentioned above) produces an algorithmic system that, for the aforementioned example, will produce an infinite derivation; thus the induced algorithm is non-terminating. Consider the following three algorithmic subtyping rules:

$$\text{(Out)}\ \frac{\Sigma \vdash^{\circledast} U_1 \leqslant U_2 \qquad \Sigma \vdash S_1 \leqslant \mathcal{A}\langle S_{2h}\rangle^{h\in H}}{\Sigma \vdash ![U_1].S_1 \leqslant \mathcal{A}\langle ![U_2].S_{2h}\rangle^{h\in H}} \qquad \text{(RecR)}\ \frac{n = \mathsf{depth}\langle \mathsf{top}(S), S'\rangle \quad n \geq 1 \qquad \Sigma, S \leqslant S' \vdash S \leqslant \mathsf{unfold}^n(S')}{\Sigma \vdash S \leqslant S'}$$

$$\text{(RecL)}\ \frac{\Sigma, \mu\mathbf{t}.S \leqslant S' \vdash \mathsf{unfold}^1(\mu\mathbf{t}.S) \leqslant S'}{\Sigma \vdash \mu\mathbf{t}.S \leqslant S'}$$

In (Out) the supertype has a comparable top-level output (selection is similar), possibly guarded under inputs and branching within $\mathcal{A}$. The rule (RecR) (which has less priority than (RecL)) uses some auxiliary functions. $\mathsf{top}(S)$ returns the constructor symbol at the top of S which must be guarded, for example ! if $S = ![U].S'$. Then, depth is a terminating function that returns the maximum number of recursions that need to be unrolled at the supertype before the desired constructor (such as !) is encountered. The purpose of (RecR) is, then, to unroll the supertype just enough for the desired constructor (the head of the subtype) to appear at the top-level (i.e., not under recursion), enabling the subsequent application of an asynchronous rule such as (Out). Moreover, all input-prefixed or branched types are possible targets of unfolding, contrary to the usual technique of only unfolding μ-prefixed types during the generation of subgoals.

We consider the previously mentioned infinite-simulation example, which can be typed using only the rules we mentioned. For clarity we use a simplified syntax ignoring carried types. Let

$S_1 = \mu\mathbf{t}\,.\,!\,.\,\mathbf{t}$ and $S_2 = \mu\mathbf{t}\,.\,!\,.\,?\,.\,\mathbf{t}$. Then we obtain the following infinite inference:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{\ldots, !.S_1 \leqslant ?.S_2, !.S_1 \leqslant ?.!.?.S_2\} \;\vdash\; S_1 \leqslant ?.?.S_2 \quad (\star)}\,\textsf{(RecL)}}{\{\ldots, !.S_1 \leqslant ?.S_2\} \;\vdash\; !.S_1 \leqslant ?.!.?.S_2}\,\textsf{(Out)}}{\{S_1 \leqslant S_2, !.S_1 \leqslant S_2, !.S_1 \leqslant !.?.S_2, S_1 \leqslant ?.S_2\} \;\vdash\; !.S_1 \leqslant ?.S_2}\,\textsf{(RecR)}}{\{S_1 \leqslant S_2, !.S_1 \leqslant S_2, !.S_1 \leqslant !.?.S_2\} \;\vdash\; S_1 \leqslant ?.S_2 \quad (\star)}\,\textsf{(RecL)}}{\{S_1 \leqslant S_2, !.S_1 \leqslant S_2\} \;\vdash\; !.S_1 \leqslant !.?.S_2}\,\textsf{(Out)}}{\{S_1 \leqslant S_2\} \;\vdash\; !.S_1 \leqslant S_2}\,\textsf{(RecR)}}{\emptyset \;\vdash\; S_1 \leqslant S_2}\,\textsf{(RecL)}$$

Notice that the initial goal depends on proving an increasingly larger subgoal resulting from unfolding and fetching an output from the respective supertype, leaving residual input actions that accumulate; we mark these points with $(\star)$. However, our empirical observation is that the shape of the supertype in those subgoals evolves following a regular pattern, with the same sequence of rules (RecL,RecR,Out) applied ad infinitum. This motivates the search for a discriminating condition for exactly those diverging goals, effectively bounding these infinite derivations.

A First Condition In our previous work by Mostrous and Yoshida [64], we formalised an algorithmic subtyping for HO$\pi^{\mathbf{as}}$ where the recursion rule (RecR) has an additional sidecondition:

$$\textsf{(RecR)}\ \dfrac{\begin{array}{c} n = \mathsf{depth}\langle \mathsf{top}(S), S' \rangle \quad n \geq 1 \\ \Sigma, S \leqslant S' \vdash S \leqslant \mathsf{unfold}^n(S') \quad S \bowtie S' \end{array}}{\Sigma \vdash S \leqslant S'}$$

where $S \bowtie S'$ means that S and S' have the same corresponding constructors for actions inside recursion. With this condition, the above non-terminating example is *rejected* in a finite derivation, and specifically in the first instance of (RecR) starting from the root.

A Better Solution Our aim is to formulate a sound and complete algorithmic system, therefore the above solution which rejects the diverging (but valid by $\leqslant_c$) example is not fully satisfactory, even though the method is correct by the restriction to $\bowtie$-compatible subtypings. Hence we follow a new approach, by replacing the standard axiom (Asmp) for assumed hypotheses (shown on the left below) with a modified version (shown on the right below). As expected (Asmp) is applicable with priority over other rules, ensuring that a goal already assumed does not trigger a re-check.

$$\text{(Asmp)}\ \frac{}{\Sigma, S_1 \leqslant S_2 \vdash S_1 \leqslant S_2} \quad \Longrightarrow \quad \text{(Asmp)}\ \frac{}{\Sigma, S_1 \leqslant \mathcal{A}\langle S_{2h}\rangle^{h\in H} \vdash S_1 \leqslant \mathcal{A}\langle \mathcal{A}\langle S_{2h}\rangle^{h\in H}\rangle^{h\in H}}$$

When $\mathcal{A} = \langle\cdot\rangle^{h\in\{0\}}$, the new rule gives an instance of the standard axiom (left above), since $\mathcal{A}\langle S_{2h}\rangle^{h\in H} = \mathcal{A}\langle \mathcal{A}\langle S_{2h}\rangle^{h\in H}\rangle^{h\in H} = S_{2h}$. It is straightforward to check that the algorithm now terminates when approximating an infinite simulation that can be constructed with a regular input-action increase in the structure of the supertype. This regularity, reflecting the residual inputs/branching from eager unfolding, is captured by the nested $\mathcal{A}$-context. We have checked that this rule gives an affirmative answer to $\mu\mathbf{t}.![U_1].\mathbf{t} \leqslant \mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$ (and other similar diverging examples) in a finite number of steps. This is easy to see in the previous derivation, where the topmost instance of (RecL) would be replaced with the new (Asmp) instantiated with $\mathcal{A} = ?.\langle\cdot\rangle$, validating the intended subtyping.

We have not yet determined whether this system is sound and complete with respect to the coinductive method, as it is not immediately clear if with the rule (Asmp) it accepts any subtypings not supported in $\leqslant_c$, or if all infinite simulations can indeed be approximated in this way.

Towards an Algorithm

Achieving a provably sound and complete algorithmic subtyping remains an important future work, and one upon which subsequent practical efforts will be based on. There are a number of technical components that need to be developed:

Bounding the goals In the standard systems, the maximum number of subgoals that may be checked in a single derivation is bounded by the cartesian product of the subexpressions of the initial goal. With our proposed rules, the finiteness of the set of possible subgoals must be reconsidered, but we expect that the modified axiom (Asmp) will offer exactly such a bound. Then, we can prove termination following the method of Gay and Hole [39].

Soundness and completeness Rule (Asmp) holds the key to the correctness of the algorithm: we need to establish a correspondence between all infinite simulations and the use of (Asmp) in the respectively induced algorithmic derivations.

Invertibility of $\leqslant_c$ This property implies that the shape of a subtype can be determined from a given supertype. It is useful in order to avoid having a combinatorial explosion in the paths that need to be checked in a derivation: an algorithm starts from the initial goal and moves "up," encountering subtypes (in the premises) that can be determined from the supertype (in the conclusion), using an inversion lemma or an inverted definition for subtyping. We believe that $\leqslant_c$ can be adapted to obtain an inverted "supertype-of" relation $\leqslant_c^{\mathsf{inv}}$ with the property $\leqslant_c^{\mathsf{inv}} = \leqslant_c^{-1}$.

Buffers and Asynchronous Subtypes

One final consideration is implementation; even with a correct and terminating algorithm, infinite buffer resources may be needed due to subtyping alone. For example let, omitting carried types, $S_1 = \mu\mathbf{t}\,.\,!\,.\,\mathbf{t}$ and $S_2 = \mu\mathbf{t}\,.\,!\,.\,?\,.\,\mathbf{t}$. Then $\overline{S_2} = \mu\mathbf{t}\,.\,?\,.\,!\,.\,\mathbf{t}$, and a process performing S_2 may communicate with a process that implements $\overline{S_2}$. By the subtyping $S_1 \leqslant_c S_2$, a process which acts according to S_1 can also interact with one that follows $\overline{S_2}$, but clearly the former never receives the outputs of the latter. Hence the input buffer of the process with session S_1 will continually and indefinitely increase storing the outputs of $\overline{S_2}$. Moreover, note that a subtype of $\overline{S_2}$ can also ignore the input, creating the same situation at both buffers of the session. It may therefore be preferable to identify those subtypes, which seem to be the ones inducing infinite simulations and (in the standard algorithmic system without the modified (Asmp) axiom) divergence, as undesirable and decline to accept them. To implement this, it is enough to *fail* — instead of succeeding — the algorithmic derivation with (Asmp) when an infinite simulation is approximated:

$$\text{(Asmp)}\ \frac{\mathcal{A} \neq \langle\cdot\rangle \Rightarrow \mathsf{fail}}{\Sigma, S_1 \leqslant \mathcal{A}\langle S_{2h}\rangle^{h\in H} \vdash S_1 \leqslant \mathcal{A}\langle \mathcal{A}\langle S_{2h}\rangle^{h\in H}\rangle^{h\in H}}$$

The sidecondition $\mathcal{A} \neq \langle\cdot\rangle \Rightarrow \mathsf{fail}$ ensures that the rule can still be used as the standard axiom when there is no duplicated context, that is, when $\mathcal{A} = \langle\cdot\rangle$. Otherwise a match on a nested (and duplicated) non-empty $\mathcal{A}$ should cause failure, immediately bringing to an end the building of the derivation without following alternative goal paths which can lead to divergence. This solution has similar effect to the use of $\bowtie$ in our original modification of (RecR), since we again have $S_1 \leqslant_c S_2$ but not $\emptyset \vdash S_1 \leqslant S_2$. However, the solutions are not equivalent, since $\bowtie$ is more rigid and rejects desirable subtypings such as $\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t} \leqslant \mu\mathbf{t}.![U_1].?[U_2].![U_1].?[U_2].\mathbf{t}$ whose constructors do not exactly match, but are still proportional; this example is accepted with both the proposed versions of (Asmp). The resulting system will not be formally sound without a suitable sidecondition, as a valid class of subtypings in $\leqslant_c$ will be rejected by $\leqslant$, but it may be the most reasonable compromise for practical applications with limited resources towards the allocation of buffer space.

7.3 Progress in Asynchronous Higher-order Sessions

Another direction is communications *progress*, discussed in Section 3.7. The existing methods, such as the *interaction* typing system by Dezani, de' Liguoro, and Yoshida [29], can be adapted to provide a solution for languages with mobile code such as HO$\pi^{\mathbf{s}}$ and HO$\pi^{\mathbf{as}}$. Nevertheless, in the presence of higher-order code mobility, the extension demands care since it requires tracking dependencies inside mobile code. For example, if $s!\langle\ulcorner P\urcorner\rangle$ is blocked, the sessions inside $\ulcorner P\urcorner$ are also blocked. On the other hand, we postulate that asynchronous subtyping does not introduce deadlock to a deadlock-free supertype, as outputs and selections can only be done in advance (partial commutativity), satisfying even stricter input dependencies than those required by the dual session of the supertype. Also, an alleviating factor might be that sessions in mobile code (and in general, within structures which can be used as values) must be completed. This indicates that a sufficient analysis might only need to explicitly consider linear functions in the ordering of session actions, obtaining a straightforward solution.

7.4 Algorithmic Type-checking for the Session Object Calculus

To facilitate itself as a foundation for practical language design, the **session**ς calculus would greatly benefit from concrete type-checking methods, and the possibility of (even restricted) type inference. In the present iso-recursive setting, subtyping does not pose a significant challenge, but type-checking and inference still require a careful formulation.

A useful property with regard to type-checking is that of *unique* types, or more generally *minimum* types. A type system with minimum types assigns a unique type to each typable term, and moreover the assigned type is the smallest, by subtyping. The aim of minimum typing is to simplify type-checking algorithms, since the construction of a single successful derivation is always sufficient. Unique types arise in the simpler setting where there is no subtyping. Minimum types have been used to guide algorithmic systems for λ-calculus, see for example Pierce's book [76, § 16.2], and for the first-order object calculus with subtyping $\mathbf{Ob_{1<:}}$ of Abadi and Cardelli [2]. In the case of **session**ς, minimum types are not easy to achieve, or even necessarily desirable, since, for example, the same object w in a branching $u \triangleright w$ can induce incomparable types for u. Consider, for instance:

$$\begin{aligned}\mathsf{let}\, w \;&=\; [\tau \,|\, l_1 = \varsigma(x_1)\,\lambda(y_1)\, y_1 \triangleright x_1\,,\\ &\qquad l_2 = \varsigma(x_2)\,\lambda(y_2)\, y_2 \triangleright x_2\,,\\ &\qquad l_3 = \varsigma(x_3)\,\lambda(y_3)\, \mathsf{close}(y_3)]\\ \mathsf{in}\ & u \triangleright w\end{aligned}$$

with:

$$\tau = [l_1 : (\sigma)\,[], l_2 : (\sigma)\,[], l_3 : (\mathsf{end})\,[]\,] \qquad \text{and} \qquad \sigma = \&(X)[l_1 : X\,, l_2 : X\,]$$

Then (BRANCH) (from Figure 6.6) can type $u \triangleright w$ with type σ for u, after subtyping of τ to $[l_1 : (\sigma)\,[], l_2 : (\sigma)\,[]\,]$ which enables the folding (to σ), and also with:

$$\sigma' = \&(Y)[l_1 : \sigma, l_2 : \sigma, l_3 : \mathsf{end}]$$

The larger branching type σ' (including $l_3 : \mathsf{end}$) does not match the use in the methods (where the subset of branches in σ is assigned based on τ), thus folding to σ does not take place. Moreover, the types σ' and σ cannot be related by $\preccurlyeq$, since in the iso-recursive method folding and unfolding are explicitly triggered within terms (in our case branching and selection). However, this difficulty in finding a suitable typing system with minimum types may become irrelevant, if an equi-recursive subtyping method is used instead, since then we could accept σ' as a minimum typing for u in $u \triangleright w$, obtaining σ as a supertype if desired, using a coinductive framework.

Although the minimum types method seems unsuited to the object calculus with iso-recursive sessions, we are motivated to find a solution since this form of subtyping enjoys a very simple theory compared to the coinductive formalisation. An appropriate strategy might be to recast algorithmic subtyping as a *constraint* solving problem. This approach stems from the observation that typing information from the root of a derivation can inform the choice of types deep in the derivation tree. Instead of trying to type these deep subderivations directly as they appear, this method progressively introduces subtyping constraints that, if solvable, induce a typing. A solution in that case is a substitution of concrete types, extracted from the type annotations, for type variables in the generated constraint set. Type reconstruction, or inference, works similarly, but in that case there are no type annotations, and the substitution is inferred. For a detailed exposition to constraint-based algorithmic typing, see Pierce's book [76, § 22.3]. For $\mathbf{Ob_{1<:}}$, Palsberg [74] has developed efficient algorithmic type reconstruction systems. In the context of inference for session types without asynchronous subtyping, Dezani *et al.* [31, 30] define a typing system with unification-based substitution inference.

7.5 Extension to Objects with Self Types

An important concept in the work of Abadi and Cardelli [2] is that of *self types*, which give recursive types to special classes of objects, such as those returning the object itself (or a modified version of it) from within a method. We decided not to include them in **session**ς, for simplicity of presentation, and to obtain a more manageable calculus. It is an important future work for us to investigate the interplay between session subtyping, object subtyping, and self types. In particular, we want to answer the question of how to obtain correctly the deep subtypings that arise when a recursive branching $y \triangleright x$ within a method executes the session argument y with a subtyped implementation of self as x, one that provides a larger interface and hence, more branches. This situation arises in systems in which objects can be extended, such as when using inheritance or mixin-composition.

7.6 Implementation of Sessions

Buffer Size Bounding and Asynchronous Subtyping The buffer size bounds inferred in the system of Gay and Vasconcelos [40] can in some cases enable the static allocation of (possibly optimised) buffer resources at the compilation level. It would be interesting to see if this method, which specifically calculates the size of the required input buffer given a type, can be extended to work in the presence of asynchronous subtyping. A first observation, hinting at a difficult point, is that subtyping affects the buffer bound. Consider the subtyping $\mu\mathbf{t}.![U_1].\mathbf{t} \leqslant_c \mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$. For $\mu\mathbf{t}.![U_1].\mathbf{t}$ their method would give a bound of zero, since only inputs and branching are considered, while for $\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$ the bound would be one, corresponding to the (repetitive) input. However, for both types the other end can be typed with $\mu\mathbf{t}.?[U_1].![U_2].\mathbf{t}$, and the fact that $\mu\mathbf{t}.![U_1].\mathbf{t}$ ignores inputs from $![U_2]$ does not mean that these are not received at its input buffer, hence the assigned bound should be ∞ instead of zero. Similar situations arise whenever the subtype has a greater proportion of outputs/selections than the supertype, as in $\mu\mathbf{t}.![U_1].![U_1].?[U_2]\mathbf{t} \leqslant_c \mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$, causing an incremental buffer size increase at the receiver of $![U_1]$. In the last example the dual of the supertype $\mu\mathbf{t}.![U_1].?[U_2].\mathbf{t}$ induces a buffer bound of one, but the subtype $\mu\mathbf{t}.![U_1].![U_1].?[U_2].\mathbf{t}$ causes it to actually need a buffer of size at least two in order to accommodate for the double output (between inputs) at each recursion. Hence it may be the case that such bounds *cannot* be inferred in the presence of asynchronous subtyping, but even in that case, there may still be a suitable adaptation of the technique for the typing restriction detailed previously in § 7.2.

Sessions for Erlang A solution to the problem of algorithmic type checking in both higher-order processes and in objects with sessions, and to a lesser degree the ability to optimise the allocation of finite resources for session running programs, lead us to the path of implementation equipped with a strong arsenal of foundational theories. One particularly interesting language that encompasses *message-passing processes* at the core of interacting programs is Erlang [53], a functional language with sums (cases on the shape of a value), records, and higher-order functions. In the words of Armstrong [4, § 6.3], Erlang's creator:

> "Erlang views the world as communicating black boxes, exchanging streams of message that obey defined protocols."

Essentially, Erlang processes are always in a *session* which is identified with the process identity (or *pid*). Communications towards the process use the pid to locate the buffer to which the message should be appended, and only a process can read its own pid-indexed buffer. Complex protocols arise when multiple processes interact and cooperate through their respective pids. Moreover, the dynamic update of programs without interruption (*hot-swapping*), a distinguishing feature of Erlang, indicates that there is an element of code mobility, combined with a dynamic modification of runtime processes which is straightforwardly subsumed by the one found in objects (manifested through method update). Session typing can certainly benefit Erlang. From this starting point we hope to follow two avenues: first, a study into type-inference of session-based types for *existing* Erlang code, perhaps enabling us to retroactively verify part of the codebase, which has not been done for the process-oriented part of the language; second, to implement a language similar to Erlang in efficiency and programming style, but with a typed session discipline at its core, paving the future for a more powerful and verification-friendly programming paradigm for massively concurrent applications.

Our design can be partly informed by the subtyping-based type inference system of Wadler [56], and by the soft typing system by Nyström [70]. The *success typings* of Lindahl and Sagonas [55] seem to be a suitable basis, and are used in Dialyzer, the official static analysis component of the open source Erlang compiler. Note, however, that none of these systems address the process (communications) part of Erlang, instead focusing on the functional core. Armstrong [3] addresses the interconnection of Erlang and other languages with contracts that specify state-machines describing the behaviour of Erlang processes, but still, the method cannot capture behaviour as precisely as session types, and is not applicable as a general static verification of communicating processes.

Sessions on the GO Session typing could benefit the newly released GO language by Google [28]. The concurrency mechanism of GO is similar to that of Erlang, but channels are explicit and typed with a singleton carried type, facilitating a more direct adaptation of the standard session typing methodologies. Communication is buffered. Moreover, GO has structurally typed objects, indicating that our session objects calculus might provide a suitable base for formalisation.

Towards Language-independent Implementations of Session-oriented Programs As sessions become more and more pervasive, their restriction to serve as a language-specific verification mechanism may become a limiting factor for some domains: we envision that eventually there will be a need to interconnect diverse session-oriented programs implemented in heterogeneous environments. Thus, we might need to decide how session types can be used at a more abstract level, using a restricted set of universal datatypes such as XML types, possibly implemented using a standardised mechanism such as SOAP.

Already, the W3C Choreography Web Description Language (CDL) [90, 21] addresses many of these aspects in a web services context, but what we are suggesting is a session middleware framework similar in principle (but not technically related) to CORBA, connecting binary-incompatible communicating programs.

7.7 Concluding Remarks

A fundamental motivation of the typing techniques that are utilised in this work is that communication must be treated as *central* to programming, deserving the same attention in its verification as any other construction, such as a function, object, or variable. Session types address precisely this, and we presented a theoretical foundation integrating sessions with higher-order processes and objects, which can support structured type-safe communications in functional, process, object, and multi-paradigm languages. Furthermore, we have formalised a subtyping approach in which certain re-structurings of communications are allowed, when they preserve the desired safety properties of programs. This *asynchronous subtyping* was presented in both a coinductive and an inductive formulation, yielding respectively an equi-recursive and iso-recursive system; the first method admits many more subtypings, the second has the benefit of simplicity. For the future we have important directions to explore using this work as a basis, aiming at more theoretical results, algorithms for practical subtyping implementations, and the adaptation of session typing to the actor-declarative paradigm of the language Erlang.

This concludes the thesis.

Bibliography

[1] SENSORIA project. `http://www.sensoria-ist.eu/`.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Joe Armstrong. Getting erlang to talk to the outside world. In *Erlang Workshop*, pages 64–72, 2002.

[4] Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 1–26, New York, NY, USA, 2007. ACM.

[5] Franco Barbanera, Sara Capecchi, and Ugo de' Liguoro. Typing Asymmetric Client-Server Interaction. In *Proc. of FSEN'09*, LNCS. Springer-Verlag, 2009.

[6] Henk Barendregt. *The Lambda Calculus*. North Holland, 1985.

[7] Martin Berger, Kohei Honda, and Nobuko Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In *Proceedings of ICALP 2008 Part II – Automata, Languages and Programming*, volume 5126/2009 of *LNCS*, pages 99–111. Springer-Verlag, 2009.

[8] Lorenzo Bettini, Sara Capecchi, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Betti Venneri. Session and Union Types for Object Oriented Programming. In Rocco De Nicola, Pierpaolo Degano, and José Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 659–680. Springer-Verlag, 2008.

[9] 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, 2008.

[10] Eduardo Bonelli and Adriana Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256, 2008.

[11] Eduardo Bonelli, Adriana Compagnoni, and Elsa Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.

[12] Michele Boreale, Roberto Bruni, Luis Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco T. Vasconcelos, and Gianluigi Zavattaro. SCC: a Service Centered Calculus. In M. Bravetti and G. Zavattaro, editors, *Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 38–57. Springer-Verlag, 2006.

[13] Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and Pipelines for Structured Service Programming. In Gilles Barthe and Frank S. de Boer, editors, *Proceedings of FMOODS 2008, 10th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *LNCS*, pages 19–38. Springer-Verlag, 2008 2008.

[14] Mario Bravetti and Gianluigi Zavattaro. A theory for strong service compliance. In *COORDINATION'07*, volume 4467 of *LNCS*, pages 96–112. Springer-Verlag, 2007.

[15] Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Software Composition*, volume 4829 of *LNCS*, pages 34–50, 2007.

[16] Roberto Bruni, Ivan Lanese, Hernan Melgratti, and Emilio Tuosto. Multiparty Sessions in SOC. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 67–82. Springer-Verlag, 2008.

[17] Lus Caires and Hugo T. Vieira. Conversation types. In *Proceedings of ESOP 2009*, volume 5502, pages 285–300. Springer-Verlag, 2009.

[18] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, and Elena Giachino. Amalgamating Sessions and Methods in Object Oriented Languages with Generics. *Theoretical Computer Science*, 410:142–167, 2009.

[19] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17, 2007.

[20] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions for session types. In *19th International Conference on Concurrency Theory (Concur'08)*, LNCS, pages 402–417. Springer-Verlag, 2008.

[21] Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. To be published by W3C. Available at `http://www.dcs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf`, 2006.

[22] Luca Cardelli. An Accidental Simula User. In *ECOOP '07*, volume 4609 of *LNCS*, page 201. Springer-Verlag, 2007. Dahl-Nygaard Senior Prize Talk, available at `http://lucacardelli.name/Talks.htm`.

[23] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. Foundations of Session Types. In *PPDP'09*. ACM Press, 2009. Full version: `http://www.di.unito.it/~dezani/papers/cdgpFull.pdf`.

[24] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *POPL'08*, pages 261–272. ACM, 2008.

[25] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5):1–61, 2009.

[26] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer-Verlag, 2007.

[27] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James J. Leifer. Secure implementations for typed session abstractions. In *CSF*, pages 170–186, 2007.

[28] Google Corporation. The GO programming language, November 2009. `http://golang.org/`.

[29] Mariangiola Dezani-Ciancaglini, Ugo de' Liguoro, and Nobuko Yoshida. On Progress for Structured Communications. In Gilles Barthe and Cédric Fournet, editors, *TGC'07*, volume 4912 of *LNCS*, pages 257–275. Springer-Verlag, 2008.

[30] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Information and Computation*, 207(5):595–641, 2009.

[31] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352, 2006. Available from `www.doc.ic.ac.uk/~mostrous/`.

[32] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. A distributed object oriented language with session types. In *TGC*, volume 3705 of *LNCS*, pages 299–318, 2005.

[33] Sophia Drossopoulou, Mariangiola Dezani-Ciancaglini, and Mario Coppo. Amalgamating the Session Types and the Object Oriented Programming Paradigms. Presented at MPOOL'07, 2007.

[34] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, , and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In Willy Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

[35] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003.

[36] Pablo Garralda, Adriana Compagnoni, and Mariangiola Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In Michael Maher, editor, *PPDP'06*, pages 61–72. ACM Press, 2006.

[37] Simon Gay. Bounded polymorphism in session types. *MSCS*, 18(5):895–930, 2008.

[38] Simon Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *ESOP'99*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.

[39] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[40] Simon Gay and Vasco T. Vasconcelos. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming*, 20(1):19–50, 2010. Subsumes Technical Report 2007–251, University of Glasgow.

[41] Simon Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *37th ACM SIGCAT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010)*, 2010.

[42] Marco Giunti, Kohei Honda, Vasco T. Vasconcelos, and Nobuko Yoshida. Session-based type discipline for pi calculus with matching. Presented at PLACES 2009 — 2nd International Workshop in Programming Language Approaches to Concurrency and Communication-cEntric Software, 2009.

[43] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*. Elsevier ENTCS, 1998.

[44] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, 2007.

[45] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. Safedpi: A language for controlling mobile code. *Acta Informatica*, 42(4-5):227–290, 2005.

[46] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[47] Kohei Honda. Types for Dyadic Interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.

[48] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, 1998.

[49] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[50] Raymond Hu. Implementation of a distributed mobile Java. Master's thesis, Imperial College London, 2006.

[51] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

[52] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[53] Claes Wikstrm Joe Armstrong, Robert Virding and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[54] Cosimo Laneve and Luca Padovani. *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, chapter The Pairing of Contracts and Session Types. Springer-Verlag, 2008.

[55] Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 167–178, New York, NY, USA, 2006. ACM.

[56] Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *Proceedings of 2nd International Conference on Functional Programming*, June 1997.

[57] Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In *COORDINATION*, volume 5052 of *LNCS*, pages 216–231. Springer-Verlag, 2008.

[58] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, Berlin, 1980.

[59] Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.

[60] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. CUP, 1999.

[61] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.

[62] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation*, 100(1), 1992.

[63] Dimitris Mostrous and Nobuko Yoshida. Two Session Typing Systems for Higher-Order Mobile Processes. In *TLCA'07*, volume 4583 of *LNCS*, pages 321–335. Springer-Verlag, 2007. Available from `www.doc.ic.ac.uk/~mostrous/`.

[64] Dimitris Mostrous and Nobuko Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA'09*, number 5608 in LNCS. Springer-Verlag, 2009. Available from `www.doc.ic.ac.uk/~mostrous/`.

[65] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, volume 5502 of *LNCS*, pages 316–332. Springer-Verlag, 2009. Available from `www.doc.ic.ac.uk/~mostrous/`.

[66] Stavros Mostrous. Extending Objective Caml with Sessions, May 2007. Bachelor's Thesis, Nottingham Trent University.

[67] The Message Passing Interface (MPI) standard. `http://www-unix.mcs.anl.gov/mpi/usingmpi/examples/intermediate/main.htm`.

[68] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer-Verlag, 2004.

[69] Matthias Neubauer and Peter Thiemann. Session types for asynchronous communication, 2004. Unpublished manuscript.

[70] Sven-Olof Nyström. A soft-typing system for erlang. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 56–71, New York, NY, USA, 2003. ACM.

[71] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[72] Luca Padovani. Contract-directed synthesis of simple orchestrators. In *CONCUR*, volume 5201 of *LNCS*, pages 131–146, 2008.

[73] Luca Padovani. Session types at the mirror. In *Proceedings of the 2nd Workshop on Interaction and Concurrency Experience (ICE'09)*. lncs, 2009.

[74] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, December 1995.

[75] Press Association Photos. ShootLive technology, 2009. Description available at `http://www.shootlive.com/technology/`.

[76] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[77] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.

[78] D. Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In *Proc. TAPSOFT'93*, volume 668 of *LNCS*, pages 151–166. Springer-Verlag, 1993.

[79] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[80] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[81] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[82] UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.

[83] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the Behavior of Objects and Components using Session Types. In *FOCLASA'02*, volume 68(3) of *ENTCS*. Elsevier, 2002.

[84] Vasco T. Vasconcelos. A note on a typing system for the higher-order π-calculus. Keio University, September 1993.

[85] Vasco T. Vasconcelos. *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM 2009)*, volume 5569 of *LNCS*, chapter Fundamentals of Session Types, pages 158–186. Springer-Verlag, 2009.

[86] Vasco T. Vasconcelos, Simon Gay, and António Ravara. Typechecking a Multithreaded Functional Language with Session Types. *TCS*, 368(1–2):64–87, 2006.

[87] Vasco T. Vasconcelos, Simon Gay, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Dynamic interfaces. In *International Workshop on Foundations of Object-Oriented Languages (FOOL'09)*, 2009.

[88] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In *ESOP*, volume 4960 of *LNCS*, pages 269–283, 2008.

[89] David Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT, 2005. Editor Benjamin C. Pierce.

[90] Web Services Choreography Working Group. Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.

[91] Nobuko Yoshida. Channel dependency types for higher-order mobile processes. In *POPL '04*, pages 147–160. ACM Press, 2004. Full version available at `www.doc.ic.ac.uk/~yoshida/`.

[92] Nobuko Yoshida and Vasco T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisited. In *SecRet'06*, volume 171(3) of *ENTCS*, pages 127–151. Elsevier, 2007.