Tampereen yliopisto

Pekka Peltola

A GENTLE INTRODUCTION TO SESSION TYPES

Faculty of Information Technology and Communication Sciences
Master's Thesis
June 2021

ABSTRACT

Pekka Peltola: A Gentle Introduction to Session Types
Master's Thesis
Tampere University
Computer Science
June 2021

We present session types, a type formalism for structured communication. The goal of the thesis is to give an elementary introduction to session types for a new reader with only a little or no knowledge of related concepts, such as π-calculus or type systems. We start by motivating the reader with an informal example and later define the language formally. Only basic language constructs are defined at the beginning and the language is extended gradually. We prove the soundness of the type system, give practical example how session types help developing server-client systems and finally talk briefly about extensions and related work.

Keywords: type systems, session types, pi-calculus, concurrency, processes

Contents

1 Introduction

In concurrent and distributed systems, for example server-client systems, interaction partners communicate on an agreed protocol. The protocol describes the type and direction of the exchanged data. Session types, first proposed almost thirty years ago by Honda [1] and later revised by Takeuchi et al. [2] and Honda et al. [3], are a type formalism to model such interactions. They allow describing protocols as types and verifying the conformance of an implementation by static typechecking.

Session types have been mostly developed using Milner's π-calculus [4] as a framework. The original work used a variant of π-calculus and later session types and session primitives were added to the standard π-calculus [5] by Gay and Hole. Many extensions to session types have been studied, such as subtyping [5], multiparty session types [6] and polymorphism [7]. In [8] Kobayashi presented an encoding of session types to standard π-calculus with types, which was further studied by Dardha et al. [9].

A *session* is a unit of abstraction for describing information exchange between two interacting partners. Whenever a new session is established, a fresh channel, a *session channel*, is created, and all communication in the session is done privately via the session channel. A session type is used to describe one end of a session channel and the other end has a dual type. *Duality* guarantees *communication safety*, namely that the types of received and sent messages match, and *communication fidelity*, namely that the structure of the messages is as expected.

A session type consists of a sequence of input and output operations, together with external and internal choices, namely the offering of options and the selection of an option, respectively. Let us consider session types

$$S_1 \coloneqq ?\mathit{string}.!\mathit{int}.\mathit{end},$$
$$S_2 \coloneqq ?\mathit{string}.?\mathit{string}.!\mathit{string}.\mathit{end} \text{ and}$$
$$S \coloneqq \&\langle \mathbf{length} : S_1, \mathbf{concat} : S_2 \rangle.$$

The first one describes a process that receives a string and sends back an integer, the second one receives two strings and sends back a string and the last one gives options, namely 'length' and 'concat', and behaves as described by S_1 or S_2 depending on the choice. Thus, S could describe a server that gives the length of a string or concatenates two strings, when a client selects 'length'

or 'concat', respectively. A client communicating with the server should have a dual type, that is,

$$C_1 \coloneqq !\mathit{string}.?\mathit{int}.\mathit{end},$$
$$C_2 \coloneqq !\mathit{string}.!\mathit{string}.?\mathit{string}.\mathit{end} \text{ and}$$
$$C \coloneqq \oplus\langle \mathbf{length} : C_1, \mathbf{concat} : C_2 \rangle.$$

We introduce basics of π-calculus and do not assume the reader to have prior knowledge of the calculus. However, we encourage the reader to refer to books by Milner [10] and Sangiorgi and Walker [11] for more information. Similarly, we introduce basics to understand the typing rules and derivations we use. About type systems in general and about types for π-calculus the reader can refer to [12] and [11], respectively. A reader familiar with (simply typed) λ-calculus [13] will find many of the concepts familiar. We assume the reader to know basics of university level mathematics.

The goal of this thesis is to give a solid basis for understanding session types. We are not trying to give an overview nor dive too deep in the theory, but show the very basics in great detail. Our goal is to

1. give a clear introduction to π-calculus and session types for a new reader,
2. combine syntax and rules from multiple sources to create a language, which is easy to approach and clearly distinguishes standard channels and session channels,
3. show detailed examples and proofs to reduce the reader's learning curve.

We begin Section 2 with an informal example to familiarise the reader with the concepts of the theory. In Section 3 we give syntax for process terms and define operational semantics. In Section 4 we give syntax for types and define typing rules. The soundness of the type system is proved in Section 5. After formal theory sections we give a practical example in Section 6 to demonstrate the power of session types for any developer. We briefly discuss extensions and related work in Section 7.

2 Session types informally

In this section we give an example from the literature of session types. The language used is formally defined in the next sections. For illustration purposes the language in this section is extended with data values and data types, and statements such as if-then-else.

The example, an automated teller machine (ATM), was presented in [3] and is used in many works, for example in [14] by Dezani-Ciancaglini and de'Liguoro and in [15] by Hüttel et al. Consider an ATM, which requests for an identifier from the user and then gives three options to choose from: deposit, withdraw and balance. If the user deposits money, the ATM asks for an amount and sends back the new balance. If the user withdraws, the ATM asks for an amount and either sends back that amount of money (dispense) or notifies that an overdraft occurred. If user selects balance, the ATM simply sends back the current balance. Session type for the ATM is

$$
\begin{aligned}
&S \coloneqq ?\mathit{string}.\&\langle \mathbf{deposit} : S_{dep}, \mathbf{withdraw} : S_{wd}, \mathbf{balance} : S_{bal} \rangle,\\
&S_{dep} \coloneqq ?\mathit{int}.!\mathit{int}.\mathit{end},\\
&S_{wd} \coloneqq ?\mathit{int}. \oplus \langle \mathbf{dispense} :!\mathit{int}.\mathit{end}, \mathbf{overdraft} :!\mathit{string}.\mathit{end} \rangle,\\
&S_{bal} \coloneqq !\mathit{int}.\mathit{end}.
\end{aligned}
$$

Here ! denotes receiving, $?$ sending, $\&$ offering and finally $\oplus$ selection. Furthermore, a dot denotes sequencing. Similarly to the example in the introduction, the user using the ATM should have a dual type, which can be obtained by replacing ! with $?$ and $\&$ with $\oplus$. Session type for the user is then

$$
\begin{aligned}
&C \coloneqq !\mathit{string}. \oplus \langle \mathbf{deposit} : C_{dep}, \mathbf{withdraw} : C_{wd}, \mathbf{balance} : C_{bal} \rangle,\\
&C_{dep} \coloneqq !\mathit{int}.?\mathit{int}.\mathit{end},\\
&C_{wd} \coloneqq !\mathit{int}.\&\langle \mathbf{dispense} :?\mathit{int}.\mathit{end}, \mathbf{overdraft} :?\mathit{string}.\mathit{end} \rangle,\\
&C_{bal} \coloneqq ?\mathit{int}.\mathit{end}.
\end{aligned}
$$

We now have a protocol for the communication between the ATM and the user. Next we need process implementations conforming with it. A π-calculus process

implementing the ATM is

$$
\begin{aligned}
ATM &\coloneqq a?(z).z?(id).z \triangleright \{\mathbf{deposit} : ATM_{dep} \\
&\qquad\qquad\qquad\quad \mathbf{withdraw} : ATM_{wd} \\
&\qquad\qquad\qquad\quad \mathbf{balance} : ATM_{bal}\} \\
ATM_{dep} &\coloneqq z?(amount).z!\langle amount + curr\_bal\rangle.\mathbf{0} \\
ATM_{wd} &\coloneqq z?(amount).[\text{if } amount > curr\_bal]\ z \triangleleft \mathbf{overdraft}.z!\langle\text{'ERR'}\rangle.\mathbf{0} \\
&\qquad\qquad\qquad\quad [\text{else}]\ z \triangleleft \mathbf{dispense}.z!\langle amount\rangle.\mathbf{0} \\
ATM_{bal} &\coloneqq z!\langle curr\_bal\rangle.\mathbf{0}.
\end{aligned}
$$

Above $x?(y)$ denotes receiving via x and replacing y with the received value. $x!\langle y\rangle$ denotes sending value y via x. Further, $x \triangleright \{\ldots\}$ and $x \triangleleft \ldots$ denote offering and selection, respectively. Value $curr\_bal$ denotes the current balance for the user. The ATM receives one end of a session channel via channel a and uses the received channel to communicate with the user. Note that the type of values a session channel is capable of sending or receiving changes during the communication.

A process implementing a user depositing money is

$$
\begin{aligned}
User &\coloneqq (\boldsymbol{\nu} xy)\ (a!\langle y\rangle.x!\langle'identifier'\rangle.x \triangleleft \mathbf{deposit}.User_{dep}) \\
User_{dep} &\coloneqq x!\langle amount\rangle.x?(newbalance).\mathbf{0}.
\end{aligned}
$$

The user creates and restricts $((\boldsymbol{\nu} xy)\ \ldots)$ two ends of a session channel and sends the other end, y, to the ATM via a. Restriction guarantees that the communication occurs privately. Then, the user sends his/her identifier and chooses to deposit money. Finally the user sends the amount to deposit and receives the new balance.

3 Processes

Before presenting the main topic, session types, we need a programming language for processes, which is extended with types later. We start from the very basics, define and explain the language carefully and show examples, how processes can interact. We proceed slowly starting from basic constructs and extend the language gradually. First we present the syntax (of a subset) of standard π-calculus together with operational semantics. We omit sum, name matching and silent action, as is common in the literature of session types. Later, in section 3.3, we add session primitives. For simplicity we use monadic π-calculus. For more details on the theory in sections 3.1 and 3.2 the reader can refer to [11, pp. 11-36].

3.1 Syntax

Terms in π-calculus are processes, which express mobile systems. Processes can interact by exchanging messages on a channel.

We assume a countably-infinite set $\mathcal{N}$ of *names*, which can be thought of as channel names. Elements of the set $\mathcal{N}$ are denoted by lower case letters, usually x, y, z, u, v, w.

Definition 3.1. (π-calculus) The *processes* of the π-calculus are given by syntax

$$
\begin{array}{rcll}
P & ::= & P \mid P' & \text{(composition)} \\
& \mid & (\boldsymbol{\nu} x)\, P & \text{(scope restriction)} \\
& \mid & *P & \text{(replication)} \\
& \mid & \mathbf{0} & \text{(inaction)} \\
& \mid & x?(y).P & \text{(input)} \\
& \mid & x!\langle y \rangle.P & \text{(output)}
\end{array}
$$

We use P, Q, R to range over processes. We call $x?(y)$ and $x!\langle y \rangle$ *prefixes* and say that processes $x?(y).P$ and $x!\langle y \rangle.P$ are *prefixed at name* x.

In examples we sometimes use meaningful words for processes and names, such as $Server$, $Client$, $port$, $thread$, etc. However, this is only to make examples cleaner and more illustrative by splitting the processes in parts. For example we do not accept recursive process definitions such as $Server \coloneqq Server \mid P$.

An *occurrence of a name* or an *occurence of a process* in a process is defined intuitively. A *subterm* or *subprocess* Q of a process P is an occurrence of a process Q in P. For example there are three occurrences of x and one occurrence of $x?(y).\mathbf{0}$ in process $R \coloneqq x!\langle y\rangle.(\boldsymbol{\upsilon}x)\,(x?(y).\mathbf{0})$. Subprocesses of R are R, $(\boldsymbol{\upsilon}x)\,(x?(y).\mathbf{0})$, $x?(y).\mathbf{0}$ and $\mathbf{0}$.

Next we give a short description on how different process forms are meant to be interpreted.

- An *inaction* $\mathbf{0}$ denotes a terminated process, that is, a process that does nothing.
- An *output prefix* $x!\langle y\rangle.P$ denotes a process that waits to send the name y via x and continues behaving as P.
- An *input prefix* $x?(y).P$ denotes a process that waits to receive any name via x and continues behaving as P with y as a placeholder for the received name.
- A *composition* $P \mid P'$ denotes a process consisting of two subprocesses proceeding in parallel. Furthermore, P and P' can interact via shared names. We call P, P' *components* of $P \mid P'$.
- A *restriction* $(\boldsymbol{\upsilon}x)\,P$ denotes a process, where name x is restricted to P. Subprocesses of P are able to use x to interact, but other processes can not send or receive messages via x.
- A *replication* $*P$ denotes an infinite composition $P \mid P \mid \cdots$.

Note. We use parentheses for readability. When parentheses are missing, composition has the weakest precedence, others having the same. For example

- $(\boldsymbol{\upsilon}x)\,x!\langle y\rangle.\mathbf{0} \mid \mathbf{0} = ((\boldsymbol{\upsilon}x)\,(x!\langle y\rangle.\mathbf{0})) \mid \mathbf{0}$
- $*x?(y).\mathbf{0} \mid x!\langle u\rangle.\mathbf{0} = (*(x?(y).\mathbf{0})) \mid x!\langle u\rangle.\mathbf{0}$.

Example 3.2. Process $x?(y).u?(v).v!\langle y\rangle.\mathbf{0}$ first receives a name via x, then another name via u and finally sends the first received name via the second received name.

Process $P_1 \coloneqq x?(y).y!\langle z\rangle.\mathbf{0} \mid x!\langle u\rangle.\mathbf{0}$ can evolve in three different ways:

1. receive a name via x (, with y as a placeholder),
2. send name u via x or

3. interact locally via shared name x.

Process $P_2 \coloneqq (\boldsymbol{\nu} x)\, P_1$ has only one way to evolve, that is, interacting locally via shared name x. In both processes P_1 and P_2 the components of the process share the name x and, thus, can interact via it.

A process of the form $*x?(y).P$ can be thought of as a server waiting on channel x. Whenever a client sends a name, say z, via x, the server creates a new copy of P, where y is replaced by z. Consider processes $Server \coloneqq *x?(y).y!\langle v\rangle.\mathbf{0}$ and $Client \coloneqq x!\langle z\rangle.z?(w).\mathbf{0}$. $Client$ sends name z via x to $Server$ and $Server$ creates $Copy \coloneqq z!\langle v\rangle.\mathbf{0}$. Then $Copy$ sends name v via received name z to $Client$ and both processes terminate. $Server$ continues waiting on channel x.

Note the difference in processes P_1 and P_2 in Example 3.2; P_2 can not send or receive via x, because x is restricted.

Definition 3.3. We say that y is *binding* with *scope* P in processes $x?(y).P$ and $(\boldsymbol{\nu} y)\, P$. An occurrence of a name, say y, is *bound*, if it is a binding occurrence or it lies within the scope of a binding occurrence y. If an occurrence is not bound, we say it is *free*. We write **fn**(P) for set of names having a free occurrence in P.

A name can be both free and bound in a process, for example in $Q_1 \coloneqq x!\langle y\rangle.z?(x).\mathbf{0}$ the first occurrence of x is free and the second is bound. The second occurrence is a placeholder and should be thought of as different name than the first. Changing the name of the second occurrence does not change the behaviour of the process, for example $Q_2 \coloneqq x!\langle y\rangle.z?(u).\mathbf{0}$ is essentially the same as Q_1.

Recall process $P_1 = x?(y).y!\langle z\rangle.\mathbf{0} \mid x!\langle u\rangle.\mathbf{0}$ from example 3.2. The component on the right-hand side sends the name u and continues behaving as $\mathbf{0}$. On the other hand, the component on the left-hand side receives the name u and continues behaving as $u!\langle z\rangle.\mathbf{0}$, that is, every free occurrence of y in $y!\langle z\rangle.\mathbf{0}$ is replaced by u. Thus, P_1 evolves to $u!\langle z\rangle.\mathbf{0} \mid \mathbf{0}$.

When processes receive names, we must substitute names for names. More precisely, we want to substitute the free occurrences of y in P, when $x?(y).P$ receives a name via x. However, we need to be careful with name conflicts; substituting z by y in $x?(y).y!\langle z\rangle.\mathbf{0}$ results to $x?(y).y!\langle y\rangle.\mathbf{0}$, which is not what we want. Before the substitution we should change the bound occurrences of y to, say u, and we will end up to $x?(u).u!\langle y\rangle.\mathbf{0}$.

Next we give formal definitions for substitution and α-convertibility.

Definition 3.4. A *substitution* is a function $\sigma : \mathcal{N} \to \mathcal{N}$, such that $\sigma(x) \neq x$ for $x \in N \subset \mathcal{N}$, where N is finite, and $\sigma(x) = x$ otherwise. We write $\{y_1, \ldots, y_n/x_1, \ldots, x_n\}$ for such substitution, that $N = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = y_i$ for $i = 1, \ldots, n$.

If P is a process and there is no occurence of y in P, then the process obtained by substituting every free occurrence of x by y is denoted by $P\{y/x\}$.

Definition 3.5. Let P be a process. Replacing a subterm $x?(y).Q$ or subterm $(\boldsymbol{\nu} y)\, Q$ of P by $x?(u).Q\{u/y\}$ or $(\boldsymbol{\nu} u)\, Q\{u/y\}$, respectively, is called a *change of bound names*. We say that processes P and P' are α-convertible, if P (or P') can be obtained from P' (or P) by a finite number of changes of bound names.

As we have seen previously, substituting free occurrences of names can lead to name conflicts with bound names. By Definition 3.5 we can always rename bound occurrences. Therefore, we can give the following definition for application of substitution.

Definition 3.6. Let P be a process. The process obtained by applying a substitution σ to P, denoted by $P\sigma$, is obtained by replacing every free occurrence of x in P by $\sigma(x)$ with α-conversion whenever needed, that is, $\sigma(x)$ must be free in $P\sigma$.

For example

- $(x!\langle y\rangle.z?(y).y!\langle z\rangle.\mathbf{0})\{z/y\} = x!\langle z\rangle.z?(y).y!\langle z\rangle.\mathbf{0}$
- $(x!\langle y\rangle.z?(y).y!\langle z\rangle.\mathbf{0})\{y/z\} = x!\langle y\rangle.y?(u).u!\langle y\rangle.\mathbf{0}$
- $(x!\langle y\rangle.z?(y).y!\langle z\rangle.\mathbf{0})\{u, v/y, z\} = x!\langle u\rangle.v?(y).y!\langle v\rangle.\mathbf{0}$.

Note. Substitution has the highest precedence. For example

$$x!\langle y\rangle.P\sigma = x!\langle y\rangle.(P\sigma).$$

3.2 Operational Semantics

Operational semantics, that is, the behaviour of processes in π-calculus, is defined by the *reduction* relation, denoted by $\longrightarrow$. A process P *reduces* to process Q, $P \longrightarrow Q$, if P has two parallel subprocesses, say P', P'', and P can evolve to Q as a result of P' and P'' communicating. Reduction is formally defined in Definition 3.13 and reduction rules are given in Table 3.2. Before defining reduction, we need to have tools to manipulate term-structure.

We have α-convertibility to identify processes that differ only in the choice of bound names. Similarly our intuition identifies for example processes $P \mid Q$ and $Q \mid P$. *Structural congruence* is the relation to identify processes that have intuitively the same behaviour, that is, represent the same process.

To define structural congruence, denoted by $\equiv$, we first define *context* and *congruence*.

Definition 3.7. A term obtained by replacing an occurence of $\mathbf{0}$ in a process by the *hole* $[\cdot]$ is called a *context*. If C is a context and P is a process, the result of replacing the hole in C by P is denoted by $C[P]$.

For example

$$C \coloneqq x?(y).[\cdot]$$

is a context and

$$\begin{aligned} P &\coloneqq u!\langle v\rangle.\mathbf{0}, \\ Q &\coloneqq y!\langle x\rangle.\mathbf{0}, \\ C[P] &= x?(y).u!\langle v\rangle.\mathbf{0} \text{ and} \\ C[Q] &= x?(y).y!\langle x\rangle.\mathbf{0}. \end{aligned}$$

Note that $y \in \mathbf{fn}(Q)$, but $y \notin \mathbf{fn}(C[Q])$.

Definition 3.8. Let $\mathcal{R}$ be an equivalence relation on processes. The relation $\mathcal{R}$ is a *congruence*, if for every context C and processes P, Q

$$\text{if } P \,\mathcal{R}\, Q, \text{ then } C[P] \,\mathcal{R}\, C[Q].$$

Definition 3.9. *Structural congruence* (for π-calculus), denoted by $\equiv$, is the smallest congruence on processes such that

1. if processes P and Q are α-convertible, then $P \equiv Q$,
2. $\equiv$ satisfies the axioms in Table 3.1.

Other intuitive results can be inferred from the axioms. For example $(\boldsymbol{\nu} x)\, P \equiv$

SC-ASSOC	$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
SC-COMM	$P \mid Q \equiv Q \mid P$
SC-INACT	$P \mid \mathbf{0} \equiv P$
SC-RES	$(\boldsymbol{\nu} x)\,(\boldsymbol{\nu} y)\,P \equiv (\boldsymbol{\nu} y)\,(\boldsymbol{\nu} x)\,P$
SC-RES-INACT	$(\boldsymbol{\nu} x)\,\mathbf{0} \equiv \mathbf{0}$
SC-RES-COMP	$(\boldsymbol{\nu} x)\,(P \mid Q) \equiv P \mid (\boldsymbol{\nu} x)\,Q$, if $x \notin \mathbf{fn}(P)$
SC-REP	$*P \equiv P \mid *P$

Table 3.1: Axioms for structural congruence

P for all such processes P that $x \notin \mathbf{fn}(P)$:

$$
\begin{aligned}
(\boldsymbol{\nu} x)\,P &\equiv (\boldsymbol{\nu} x)\,(P \mid \mathbf{0}) && \text{(SC-INACT)}\\
&\equiv P \mid (\boldsymbol{\nu} x)\,\mathbf{0} && \text{(SC-RES-COMP)}\\
&\equiv P \mid \mathbf{0} && \text{(SC-RES-INACT)}\\
&\equiv P. && \text{(SC-INACT)}
\end{aligned}
$$

Next we give a few examples to demonstrate how term-structure can be manipulated using structural congruence.

Example 3.10. Suppose $P \coloneqq x?(y).(R_1 \mid R_2)$ and $Q \coloneqq x?(y).(R_2 \mid R_1)$. We want to prove that $P \equiv Q$. In the previous example we used the axioms. However, none of them can be used in this case. Therefore, we need to find a suitable context and use the definition of congruence (Definition 3.8).

Consider context $C \coloneqq x?(y).[\cdot]$. By (SC-COMM) $R_1 \mid R_2 \equiv R_2 \mid R_1$. Then, $P = C(R_1 \mid R_2) \equiv C(R_2 \mid R_1) = Q$.

By Definition 3.8 $C(P) \equiv C(Q)$, whenever $P \equiv Q$. The other direction is not true in general.

Example 3.11. Consider processes $P \coloneqq x!\langle v\rangle.\mathbf{0}$ and $Q \coloneqq y!\langle v\rangle.\mathbf{0}$. Clearly P

and Q are not structurally congruent. Now suppose $C \coloneqq (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} y)\,[\cdot]$. Then,

$$\begin{aligned}
C(P) &\equiv (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} z)\,(x!\langle v\rangle.\mathbf{0}) && (\alpha\text{-convertibility})\\
&\equiv (\boldsymbol{\nu} y)\,(\boldsymbol{\nu} z)\,(y!\langle v\rangle.\mathbf{0}) && (\alpha\text{-convertibility})\\
&\equiv (\boldsymbol{\nu} y)\,(\boldsymbol{\nu} x)\,(y!\langle v\rangle.\mathbf{0}) && (\alpha\text{-convertibility})\\
&\equiv (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} y)\,(y!\langle v\rangle.\mathbf{0}) && (\text{SC-RES})\\
&= C(Q).
\end{aligned}$$

Example 3.12. Suppose $R \coloneqq (\boldsymbol{\nu} x)\,P \mid (\boldsymbol{\nu} x)\,Q$, $z \notin \mathbf{fn}(P)$, z does not occur in Q and $Q' \coloneqq Q\{z/x\}$. Then

$$\begin{aligned}
R &\equiv (\boldsymbol{\nu} x)\,P \mid (\boldsymbol{\nu} z)\,Q\{z/x\} && (\alpha\text{-convertibility})\\
&\equiv (\boldsymbol{\nu} z)\,((\boldsymbol{\nu} x)\,P \mid Q') && (\text{SC-RES-COMP})\\
&\equiv (\boldsymbol{\nu} z)\,(Q' \mid (\boldsymbol{\nu} x)\,P) && (\text{SC-COMM})\\
&\equiv (\boldsymbol{\nu} z)\,(\boldsymbol{\nu} x)\,(Q' \mid P) && (\text{SC-RES-COMP})\\
&\equiv (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} z)\,(Q' \mid P) && (\text{SC-RES})\\
&\equiv (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} z)\,(P \mid Q'). && (\text{SC-COMM})
\end{aligned}$$

Note that in the previous example the occurences of x in $(\boldsymbol{\nu} x)\,P$ and $(\boldsymbol{\nu} x)\,Q$ are different binding occurrences and represent different names. Thus, we have to use α-convertibility and have certain assumptions about z before using the axioms for structural congruence.

It is always possible to use α-convertibility so that all binding occurrences are distinct from the free names and other binding occurrences. Therefore, from now on in any mathematical context we use Barendregt's variable convention, that is,

1. no name can be both free and bound,
2. no name can be binding with two different scopes.

Next we define reduction. The rules of the form

$$\frac{Prem_1 \quad \ldots \quad Prem_n}{Conc}$$

are called *inference rules*, meaning that the *conclusion*, $Conc$, can be inferred from the *premises*, $Prem_1, \ldots, Prem_n$.

R-COM	$\dfrac{}{x!\langle y\rangle.P \mid x?(z).Q \longrightarrow P \mid Q\{y/z\}}$
R-PAR	$\dfrac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R}$
R-RES	$\dfrac{P \longrightarrow Q}{(\boldsymbol{\nu} x)\, P \longrightarrow (\boldsymbol{\nu} x)\, Q}$
R-STRUCT	$\dfrac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'}$

Table 3.2: Rules for reduction

Definition 3.13. Reduction (for π-calculus), denoted by $\longrightarrow$, is a relation defined by the rules in Table 3.2. The reflexive and transitive closure of $\longrightarrow$ is denoted by $\longrightarrow^*$.

We call a proof of a reduction a *reduction derivation*, which is viewed as a tree, whose leaves are usages of rule R-COM and root is the reduction that is derived.

Rule R-COM is the base rule for reduction. A process consisting of two components can evolve, when the components are prefixed by the same name with opposite behaviours, that is, one can send and the other can receive. Rules R-PAR and R-RES inductively specify that communication can occur inside composition and scope restriction. For example $(\boldsymbol{\nu} x)\,(x!\langle y\rangle.P \mid x?(z).Q)$ reduces to $(\boldsymbol{\nu} x)\,(P \mid Q\{y/z\})$ by R-RES and R-COM. The last rule, R-STRUCT, specifies that processes can be restructured before and after reduction and reduction is still preserved. By R-STRUCT (and R-COM) we can for example prove a reduction as simple as $x?(z).Q \mid x!\langle y\rangle.P \longrightarrow Q\{y/z\} \mid P$.

Recall processes $Server \coloneqq *x?(y).y!\langle v\rangle.\mathbf{0}$ and $Client \coloneqq x!\langle z\rangle.z?(w).\mathbf{0}$ in

Example 3.2. Now

$$
\begin{aligned}
& Server \mid Client \\
& \equiv (x?(y).y!\langle v\rangle.\mathbf{0} \mid Server) \mid Client && \text{(SC-REP)} \\
& \equiv (Client \mid x?(y).y!\langle v\rangle.\mathbf{0}) \mid Server && \text{(SC-COMM and SC-ASSOC)} \\
& = (x!\langle z\rangle.z?(w).\mathbf{0} \mid x?(y).y!\langle v\rangle.\mathbf{0}) \mid Server \\
& \longrightarrow (z?(w).\mathbf{0} \mid (y!\langle v\rangle.\mathbf{0})\{z/y\}) \mid Server && \text{(R-COM and R-PAR)} \\
& = (z?(w).\mathbf{0} \mid z!\langle v\rangle.\mathbf{0}) \mid Server && \text{(Substitution)} \\
& \equiv (z!\langle v\rangle.\mathbf{0} \mid z?(w).\mathbf{0}) \mid Server && \text{(SC-COMM)} \\
& \longrightarrow (\mathbf{0} \mid \mathbf{0}\{v/w\}) \mid Server && \text{(R-COM)} \\
& = (\mathbf{0} \mid \mathbf{0}) \mid Server && \text{(Substitution)} \\
& \equiv Server. && \text{(SC-INACT twice)}
\end{aligned}
$$

Therefore, by R-STRUCT

$$Server \mid Client \longrightarrow (z?(w).\mathbf{0} \mid (y!\langle v\rangle.\mathbf{0})\{z/y\}) \mid Server \longrightarrow Server$$

and $Server \mid Client \longrightarrow^* Server$.

Consider process $(Server \mid Client) \mid z?(u).P$. Like above, $Server$ and $Client$ can communicate via x. The process evolves to

$$((z?(w).\mathbf{0} \mid z!\langle v\rangle.\mathbf{0}) \mid Server) \mid z?(u).P,$$

which in turn has two ways to evolve: via z using either $z?(w).\mathbf{0}$ or $z?(u).P$ (after a few usages of SC-COMM and SC-ASSOC). Thus, the continuation of $Client$, that is, $z?(w).\mathbf{0}$, waiting on channel z might never get any answer. We can fix this issue by changing $Client$ to create a private channel before communicating with $Server$:

$$Client_{PRIV} \coloneqq (\boldsymbol{\nu} z)\, x!\langle z\rangle.z?(w).\mathbf{0}.$$

In $Server \mid Client_{PRIV}$ we can move $(\boldsymbol{\nu} z)$ to the beginning by SC-RES-COMP. Then, using R-RES the communication proceeds like above. However, this time with a private channel z.

To infer a reduction, say $P \longrightarrow Q$, we usually have to manipulate the term-structure of P to bring the correct terms next to each other in correct order so that R-RES, R-PAR and R-COM can be used. Then, using R-STRUCT we can infer $P \longrightarrow Q$. Because composition is associative and commutative, we can

freely change the order of the components. Then, it is easy to show that

$$R_1 \mid x!\langle y\rangle.P \mid R_2 \mid x?(z).Q \mid R_3 \longrightarrow R_1 \mid P \mid R_2 \mid Q\{y/z\} \mid R_3 \text{ and}$$
$$R_1 \mid x?(z).Q \mid R_2 \mid x!\langle y\rangle.P \mid R_3 \longrightarrow R_1 \mid Q\{y/z\} \mid R_2 \mid P \mid R_3$$

for all processes P, Q, R_1, R_2, R_3.

How a process evolves can drastically change due to the order of reductions.

Example 3.14. Consider process

$$P \coloneqq x?(v).y?(w).x?(u).\mathbf{0} \mid x!\langle a\rangle.\mathbf{0} \mid x!\langle b\rangle.y!\langle c\rangle.\mathbf{0}.$$

If the first two components in the composition interact, the process reduces to

$$y?(w).x?(u).\mathbf{0} \mid \mathbf{0} \mid x!\langle b\rangle.y!\langle c\rangle.\mathbf{0}.$$

The subprocess in the middle terminates and two other subprocesses can not interact. Therefore, the process is deadlocked. If, on the other hand, the first and the third component had interacted in P, the process would have reduced to

$$y?(w).x?(u).\mathbf{0} \mid x!\langle a\rangle.\mathbf{0} \mid y!\langle c\rangle.\mathbf{0}.$$

Then, the first and the third component would have been able to interact again. The previous process would have first reduced to

$$x?(u).\mathbf{0} \mid x!\langle a\rangle.\mathbf{0} \mid \mathbf{0}.$$

and finally to $\mathbf{0} \mid \mathbf{0} \mid \mathbf{0} \equiv \mathbf{0}$.

Next two examples show reduction derivations.

Example 3.15. Let $P \coloneqq (\boldsymbol{\nu} x)\ (x!\langle y\rangle.\mathbf{0} \mid x?(z).\mathbf{0})$. Then $P \longrightarrow \mathbf{0}$.

Proof.

$$\dfrac{\dfrac{}{x!\langle y\rangle.\mathbf{0} \mid x?(z).\mathbf{0} \longrightarrow \mathbf{0} \mid \mathbf{0}\{y/z\}}\ \text{R-COM}}{(\boldsymbol{\nu} x)\ (x!\langle y\rangle.\mathbf{0} \mid x?(z).\mathbf{0}) \longrightarrow (\boldsymbol{\nu} x)\ (\mathbf{0} \mid \mathbf{0}\{y/z\})}\ \text{R-RES}$$

where

$$(\boldsymbol{\nu} x)\ (\mathbf{0} \mid \mathbf{0}\{y/z\}) = (\boldsymbol{\nu} x)\ (\mathbf{0} \mid \mathbf{0}) \equiv \mathbf{0}.$$

Therefore, by R-STRUCT $P \longrightarrow \mathbf{0}$. □

Example 3.16. Let

$$P \coloneqq (\boldsymbol{\nu} x)\,(R \mid (x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0})) \text{ and}$$
$$Q \coloneqq (\boldsymbol{\nu} x)\,(R \mid y!\langle v\rangle.\mathbf{0}).$$

Then $P \longrightarrow Q$.

Proof. We show the derivation in two parts. Firstly

$$\dfrac{\dfrac{\dfrac{x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0} \longrightarrow y!\langle v\rangle.\mathbf{0}}{(x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0}) \mid R \longrightarrow y!\langle v\rangle.\mathbf{0} \mid R}\,\text{R-PAR}}{R \mid (x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0}) \longrightarrow R \mid y!\langle v\rangle.\mathbf{0}}\,\text{R-STRUCT}}{(\boldsymbol{\nu} x)\,(R \mid (x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0})) \longrightarrow (\boldsymbol{\nu} x)\,(R \mid y!\langle v\rangle.\mathbf{0})}\,\text{R-RES}$$

where

$$R \mid (x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0}) \equiv (x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0}) \mid R \text{ and}$$
$$R \mid y!\langle v\rangle.\mathbf{0} \equiv y!\langle v\rangle.\mathbf{0} \mid R.$$

Further,

$$\dfrac{\dfrac{}{x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0} \longrightarrow \mathbf{0} \mid (z!\langle v\rangle.\mathbf{0})\{y/z\} \equiv y!\langle v\rangle.\mathbf{0}}\,\text{R-COM}}{x!\langle y\rangle.\mathbf{0} \mid x?(z).z!\langle v\rangle.\mathbf{0} \longrightarrow y!\langle v\rangle.\mathbf{0}}\,\text{R-STRUCT}$$

□

This finishes our introduction of π-calculus. We presented only basics of the calculus and the reader can refer to aforementioned books [10], [11] for more information.

3.3 Session primitives

The main session primitive is session restriction (or session creation). Interaction between two processes in a session needs to be private. Different syntax has been used for it in the literature. For example accept/request in [3], co-variables in [16] and [9] or polarities in [5]. We use co-variables and follow the syntax in [9] excluding data values. Though we exclude data values from the syntax for simplicity, we sometimes use them in examples for illustration purposes. Examples of languages with Boolean values can be found for example

in [16] and [17].

In session restriction two names are created and binded together as co-variables. We also introduce external and internal choice, which denote the offering of options and the selection of an option, respectively. These will be explained below. We use the same name for the language, π*-calculus with sessions*, as in [9].

We assume a countably-infinite set $\mathcal{L}$ of *labels* distinct from $\mathcal{N}$ (names), denoted by $l, l_1, l_2, \ldots$. In examples we sometimes use meaningful words, such as set, $read$, etc.

Definition 3.17. (π-calculus with sessions) The processes of session-calculus are given by syntax

$$
\begin{array}{rcll}
P & ::= & P \mid P' & \text{(composition)} \\
& \mid & (\boldsymbol{\nu} x)\, P & \text{(standard channel restriction)} \\
& \mid & (\boldsymbol{\nu} xy)\, P & \text{(session channel restriction)} \\
& \mid & *P & \text{(replication)} \\
& \mid & \mathbf{0} & \text{(inaction)} \\
& \mid & x?(y).P & \text{(input)} \\
& \mid & x!\langle y\rangle.P & \text{(output)} \\
& \mid & x \triangleleft \mathbf{l}.P & \text{(selection)} \\
& \mid & x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I} & \text{(branching)}
\end{array}
$$

In $(\boldsymbol{\nu} xy)\, P\, x \neq y$, xy is an unordered pair and x, y are both binding with scope P. We call x, y *co-variables*. Because xy is an unordered pair, $(\boldsymbol{\nu} xy)\, P = (\boldsymbol{\nu} yx)\, P$.

Branching offers a set of options called *branches* with pairwise distinct labels. P in $x?(y).P$, $x!\langle y\rangle.P$, $x \triangleleft \mathbf{l}.P$ and P_i in $x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I}$ are called *continuation processes*.

We say that processes $x?(y).P$, $x!\langle y\rangle.P$, $x \triangleleft \mathbf{l}.P$ and $x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I}$ are *prefixed at name* x.

Notation. We write

$$(\boldsymbol{\nu} \widetilde{x})\, P \text{ for } (\boldsymbol{\nu} x_1) \cdots (\boldsymbol{\nu} x_n)\, P$$

and

$$(\boldsymbol{\nu} \widetilde{xy})\, P \text{ for } (\boldsymbol{\nu} x_1 y_1) \cdots (\boldsymbol{\nu} x_n y_n)\, P,$$

where $n \in \{0, 1, 2, \ldots\}$. We say that v is in $\widetilde{x}$ (or v is in $\widetilde{xy}$ or vu is in $\widetilde{xy}$), denoted

by $v \in \widetilde{x}$ (or $v \in \widetilde{xy}$ or $vu \in \widetilde{xy}$), if $v = x_i$ (or $v \in \{x_i, y_i\}$ or $vu = x_i y_i$) for some $i \in \{1, 2, \dots\}$.

The syntax for π-calculus with sessions is almost the same as for π-calculus. The difference is that three new primitives are added, namely session channel restriction, selection and branching.

Note. Session restriction has the same precedence as standard restriction. Therefore, composition has the weakest precedence, substitution the highest and others the same.

We extend α-convertibility to session restriction and follow Barendregt's variable convention.

Co-variables represent opposite ends of a session channel and interaction between them is possible. External choice $x \triangleright \{\mathbf{l}_i : P_i\}_{i \in I}$, which we call *branching*, offers options and after selection behaves according to the selection, that is, continues behaving as P_j, when label l_j is chosen. Internal choice $x \triangleleft \mathbf{l}.P$, which we call *selection*, chooses one option labeled l and continues behaving as P.

We illustrate these new primitives with an example.

Example 3.18. Consider process $(\boldsymbol{\nu} xy)\,(x?(z).P \mid y!\langle v\rangle.Q)$. Because x and y are co-variables, interaction between them is possible. The process evolves to $(\boldsymbol{\nu} xy)\,(P\{v/z\} \mid Q)$.

Communication via standard channels is possible under a session restriction. Process $(\boldsymbol{\nu} xy)\,(a?(z).P \mid a!\langle v\rangle.Q)$ evolves to $(\boldsymbol{\nu} xy)\,(P\{v/z\} \mid Q)$.

Recall the session type from introduction describing a server that gives options $length$ and $concat$ to get the length of a string or to concatenate two strings, respectively. Let us assume that integer and string values are added to the above syntax, $|s|$ denotes the length of a string s and $s_1 + s_2$ denotes the concatenation of strings s_1 and s_2. Then,

$$
\begin{aligned}
&StringServer \coloneqq *Thread,\\
&\quad Thread \coloneqq a?(z).z \triangleright \{\mathbf{length} : z?(str).z!\langle |str|\rangle.\mathbf{0},\\
&\qquad\qquad\qquad\qquad \mathbf{concat} : z?(str_1).z?(str_2).z!\langle str_1 + str_2\rangle.\mathbf{0}\}
\end{aligned}
$$

implements the specification. $StringServer$ is waiting on a standard channel a

SC-RES-S-RES	$(\boldsymbol{\nu} xy)\,(\boldsymbol{\nu} v)\,P \equiv (\boldsymbol{\nu} v)\,(\boldsymbol{\nu} xy)\,P$
SC-RES-S	$(\boldsymbol{\nu} xy)\,(\boldsymbol{\nu} vw)\,P \equiv (\boldsymbol{\nu} vw)\,(\boldsymbol{\nu} xy)\,P$
SC-RES-S-INACT	$(\boldsymbol{\nu} xy)\,\mathbf{0} \equiv \mathbf{0}$
SC-RES-S-COMP	$(\boldsymbol{\nu} xy)\,(P \mid Q) \equiv P \mid (\boldsymbol{\nu} xy)\,Q$, if $x, y \notin \mathbf{fn}(P)$

Table 3.3: New axioms for structural congruence for π-calculus with sessions

for an endpoint of a session channel (with z as a placeholder). Further,

$$StringClient \coloneqq (\boldsymbol{\nu} xy)\,ClientBody$$
$$ClientBody \coloneqq a!\langle y\rangle.x \triangleleft \mathbf{concat}.x!\langle text_1\rangle.x!\langle text_2\rangle.x?(text).\mathbf{0}$$

implements a client selecting $concat$. $StringClient$ creates two endpoints of a session channel and sends the other, y, via a to $StringServer$. After that, the server and the client can communicate privately via x and y.

For operational semantics we define structural congruence and reduction as we did for π-calculus. The axioms for structural congruence are the same as before (Table 3.1) except that axioms for session restriction are added. For reduction there are three new rules, R-COM-SESS, R-SELECT and R-RES-SESS. Definitions for context and congruence are adapted to the new syntax.

Definition 3.19. *Structural congruence* (for π-calculus with sessions), denoted by $\equiv$, is the smallest congruence on processes such that

1. if processes P and Q are α-convertible, then $P \equiv Q$,
2. $\equiv$ satisfies the axioms in Tables 3.1 and 3.3.

Definition 3.20. Reduction (for π-calculus with sessions), denoted by $\longrightarrow$, is a relation defined by the rules in Tables 3.2 and 3.4. The reflexive and transitive closure of $\longrightarrow$ is denoted by $\longrightarrow^*$.

Note that in rules R-COM-SESS and R-SELECT there are only two parallel processes under the restriction. Therefore, interaction can not occur via x and y in process

$$(\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q \mid x?(w).R).$$

R-COM-SESS	$\dfrac{}{(\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q) \longrightarrow (\boldsymbol{\nu} xy)\,(P \mid Q\{v/z\})}$
R-SELECT	$\dfrac{j \in I}{(\boldsymbol{\nu} xy)\,(x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I} \mid y \triangleleft \mathbf{l}_j.Q) \longrightarrow (\boldsymbol{\nu} xy)\,(P_j \mid Q)}$
R-RES-SESS	$\dfrac{P \longrightarrow Q}{(\boldsymbol{\nu} xy)\,P \longrightarrow (\boldsymbol{\nu} xy)\,Q}$

Table 3.4: New rules for reduction for π-calculus with sessions

However, in process

$$(\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q \mid a?(w).\mathbf{0})$$

interaction can occur via x and y, because by SC-RES-S-COMP we can move $a?(w).\mathbf{0}$ out of the restriction. Then, by R-PAR and R-COM-RESS the interaction is possible. To summarise, any process of the form

$$(\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q \mid R),$$

where $x, y \notin \mathbf{fn}(R)$ can reduce to

$$(\boldsymbol{\nu} xy)\,(P \mid Q\{v/z\} \mid R)$$

by SC-RES-S-COMP and R-STRUCT (with R-PAR and R-COM-RESS).

Note. As noted above, composition is associative and commutative. Thus, we can freely change the order of the components. We sometimes want to emphasise the usages of structural congruence, substitutions and rules R-STRUCT and R-PAR, but otherwise we skip them to make the examples shorter and easier to read.

Though interaction can not occur via x and y in process

$$(\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q \mid x?(w).R),$$

it can occur via x by R-COM and R-RES-SESS. This conflicts with the interpretation that co-variable represents only one end of a session channel. However, this process is not correctly typed in the type system (Section 4).

Example 3.21. Recall

$$\begin{aligned} StringServer &= *Thread \\ Thread &= a?(z).z \triangleright \{\mathbf{length} : z?(str).z!\langle |str| \rangle.\mathbf{0}, \\ &\qquad\qquad \mathbf{concat} : z?(str_1).z?(str_2).z!\langle str_1 + str_2 \rangle.\mathbf{0}\} \end{aligned}$$

and

$$\begin{aligned} StringClient &= (\boldsymbol{\nu} xy)\, ClientBody \\ ClientBody &= a!\langle y \rangle.x \triangleleft \mathbf{concat}.x!\langle text_1 \rangle.x!\langle text_2 \rangle.x?(text).\mathbf{0} \end{aligned}$$

from Example 3.18. Then,

$$\begin{aligned} & StringClient \mid StringServer \\ & \equiv (\boldsymbol{\nu} xy)\, ClientBody \mid Thread \mid StringServer && \text{(SC-REP)} \\ & \equiv (\boldsymbol{\nu} xy)\, (ClientBody \mid Thread) \mid StringServer && \text{(SC-RES-S-COMP)} \\ & \longrightarrow (\boldsymbol{\nu} xy)\, (x \triangleleft \mathbf{concat}. \ldots \mid y \triangleright \{\ldots\}) \\ & \qquad \mid StringServer && \text{(R-COM, R-RES)} \\ & \longrightarrow (\boldsymbol{\nu} xy)\, (x!\langle text_1 \rangle.x!\langle text_2 \rangle.x?(text).\mathbf{0} \\ & \qquad\qquad \mid y?(str_1).y?(str_2).y!\langle str_1 + str_2 \rangle.\mathbf{0}) \\ & \qquad \mid StringServer && \text{(R-SELECT)} \\ & \longrightarrow (\boldsymbol{\nu} xy)\, (x!\langle text_2 \rangle.x?(text).\mathbf{0} \\ & \qquad\qquad \mid y?(str_2).y!\langle text_1 + str_2 \rangle.\mathbf{0}) \\ & \qquad \mid StringServer && \text{(R-COMM-SESS)} \\ & \longrightarrow (\boldsymbol{\nu} xy)\, (x?(text).\mathbf{0} \mid y!\langle text_1 + text_2 \rangle.\mathbf{0}) \\ & \qquad \mid StringServer && \text{(R-COMM-SESS)} \\ & \longrightarrow (\boldsymbol{\nu} xy)\, (\mathbf{0} \mid \mathbf{0}) \mid StringServer && \text{(R-COMM-SESS)} \\ & \equiv StringServer. \end{aligned}$$

4 Types

In Section 3.3, we saw that session channels behave slightly differently than standard channels. These differences become apparent, when typing rules are given later in Section 4.2. The main difference is that standard channels always have the same type, but the type of session channels changes during a derivation. Session types describe interaction, which changes whenever one part of the conversation is completed. Further, session types have more restrictions. For example standard channels might occur in multiple threads, that is, in multiple parallel (sub)processes, while session channels occur in exactly one. For example process

$$(\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q \mid x?(w).R),$$

is not valid, because x occurs in both $x!\langle v\rangle.P$ and $x?(w).R$. In this section we extend our language with types. It is structured the same way as Section 3: we first present syntax for types and continue with typing rules.

4.1 Syntax

We use the syntax in [9] except we do not have data types. However, we sometimes use them in examples for illustration purposes like before.

Definition 4.1. The *types* of the session-calculus are given by syntax

$$\begin{array}{rcll} T & ::= & \#T & \text{(channel type)} \\ & \mid & S & \text{(session type)} \\ S & ::= & \mathit{end} & \text{(termination)} \\ & \mid & ?T.S & \text{(input)} \\ & \mid & !T.S & \text{(output)} \\ & \mid & \&\langle \mathbf{l}_i : S_i\rangle_{i\in I} & \text{(branch)} \\ & \mid & \oplus\,\langle \mathbf{l}_i : S_i\rangle_{i\in I} & \text{(select)} \end{array}$$

We use $T, T_1, T_2, \ldots$ and $U, U_1, U_2, \ldots$ to range over types and $S, S_1, S_2, \ldots$ to range over session types. Labels $\mathbf{l}_i$, $i \in I$, are pairwise distinct and the order of labelled items does not matter. S in $?T.S$, $!T.S$ and S_i in $\&\langle \mathbf{l}_i : S_i\rangle_{i\in I}$, $\oplus\langle \mathbf{l}_i : S_i\rangle_{i\in I}$ are called *continuation types*.

Type *end* describes a terminated session. *Input type* $?T.S$ describes a (ses-

sion) channel that receives a name of type T and continues interacting according to type S. Similarly, *output type* $!T.S$ describes a channel that sends a name of type T and continues interacting according to type S. *Branch type* and *select type* describe channels offering and selecting options, respectively, and continuing interacting according to type S_i (depending on the choice of a label). A standard channel of type $\#T$ sends or receives a name of type T and continues interacting according to the same type $\#T$.

When interaction partners follow a protocol, they need to have dual behaviour: whenever one sends, the other receives and whenever one selects, the other offers options. Duality guarantees that the interaction proceeds in a correct manner.

Definition 4.2. *Duality*, denoted by $\overline{\cdot}$, is a relation defined recursively by the following equations:

$$\begin{aligned}
\overline{\mathit{end}} &= \mathit{end} \\
\overline{?T.S} &= !T.\overline{S} \\
\overline{!T.S} &= ?T.\overline{S} \\
\overline{\&\langle \mathbf{l}_i : S_i \rangle_{i \in I}} &= \oplus\langle \mathbf{l}_i : \overline{S_i} \rangle_{i \in I} \\
\overline{\oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I}} &= \&\langle \mathbf{l}_i : \overline{S_i} \rangle_{i \in I}
\end{aligned}$$

Note that $\overline{\overline{S}} = S$ for all session types S. Next we give a few informal examples about typed processes.

Example 4.3. Recall

$$\begin{aligned}
&\mathit{StringServer} = *\mathit{Thread} \\
&\quad \mathit{Thread} = a?(z).z \triangleright \{\mathbf{length} : z?(\mathit{str}).z!\langle |\mathit{str}| \rangle.\mathbf{0} \\
&\qquad\qquad\qquad\qquad \mathbf{concat} : z?(\mathit{str}_1).z?(\mathit{str}_2).z!\langle \mathit{str}_1 + \mathit{str}_2 \rangle.\mathbf{0}\}
\end{aligned}$$

and

$$\begin{aligned}
&\mathit{StringClient} = (\boldsymbol{\upsilon} xy)\, \mathit{ClientBody} \\
&\mathit{ClientBody} = a!\langle y \rangle.x \triangleleft \mathbf{concat}.x!\langle \mathit{text}_1 \rangle.x!\langle \mathit{text}_2 \rangle.x?(\mathit{text}).\mathbf{0}
\end{aligned}$$

from Example 3.18 and types

$$
\begin{aligned}
&S_{server1} \coloneqq ?\textit{string}.!\textit{int}.\textit{end},\\
&S_{server2} \coloneqq ?\textit{string}.?\textit{string}.!\textit{string}.\textit{end},\\
&S_{server} \coloneqq \&\langle \mathbf{length} : S_{server1}, \mathbf{concat} : S_{server2}\rangle,\\
&S_{client1} \coloneqq !\textit{string}.?\textit{int}.\textit{end},\\
&S_{client2} \coloneqq !\textit{string}.!\textit{string}.?\textit{string}.\textit{end},\\
&S_{client} \coloneqq \oplus\langle \mathbf{length} : S_{client1}, \mathbf{concat} : S_{client2}\rangle
\end{aligned}
$$

from introduction. Suppose the type of a channel a is $\#S_{server}$, that is, a is capable of sending or receiving a name of type S_{server}. Then, the type of y in $StringClient$ (and the type of z in $StringServer$) must be S_{server}. Further, the type of x must be $\overline{S_{server}} = S_{client}$, because x and y are co-variables. The interaction continues as described by (dual) types $S_{server2}$ and $S_{client2}$, that is, the client sends two strings and receives one and vice versa for the server. Therefore, $StringClient$ and $StringServer$ are correctly typed, when the type of a is $\#S_{server}$.

4.2 Typing rules

We now have syntax for types. However, it does not tell, whether a process conforms with a type. With typing rules we can find out, whether a typing is correct or incorrect. In this section we present typing rules formally, explain every rule and give examples, how a typing for a process can be validated.

Before presenting the typing rules for the type system, we define *type environment* and introduce related terminology. $\Gamma \vdash P$ is a *type judgement* asserting that process P uses names according to the *assignments* in Γ. We write $x : T$ for an assignment of type T to name x. Typing rules in this section are adapted from [5] except we do not introduce recursive types or subtyping.

Definition 4.4. A *type environment* (or an *environment*) is a partial function from names to types, denoted by Γ. We say that type environment Γ *assigns* type T to name x, if $\Gamma(x) = T$. We write

$$x_1 : T_1, \ldots, x_n : T_n$$

for such environment that assigns T_i to x_i, $i \in \{1, \ldots, n\}$ and is undefined other-

wise. We say that x (or $x : T$) is in Γ or Γ contains x (or $x : T$), if $\Gamma(x)$ is defined (and $\Gamma(x) = T$). Occurrences of names in an environment Γ are not bound. Therefore, they are distinct from all binding occurrences of names.

If Γ_1 and Γ_2 have disjoint domains, we write Γ_1, Γ_2 for such an environment that assigns to each name $x \in dom(\Gamma_1) \cup dom(\Gamma_2)$ type $\Gamma_1(x)$, if $\Gamma_1(x)$ is defined, or type $\Gamma_2(x)$, if $\Gamma_2(x)$ is defined, and is undefined otherwise.

Let P be a process. We say that environment Γ is

1. *completed*, if only session type in the range of Γ is *end*,
2. *unlimited*, if there are no session types in the range of Γ,
3. *unlimited for* P, if Γ is completed and for every x in Γ the following holds:

$$\text{if } \Gamma(x) = \textit{end}, \text{ then } x \notin \mathbf{fn}(P).$$

In a completed environment every session channel has completed its interactions, that is, every session channel has communicated in a correct manner. In an unlimited environment there are no session types. However, during the evolution of a process session channels can be created and, therefore, session types can be added to an environment during a typing derivation. An environment that is unlimited for a specific process can contain completed session types, but the process cannot contain any session channels typed by those session types.

Some of the typing rules use addition operation on environments. This guarantees that session channels are not used by multiple parallel processes. Session channels can only be added to an environment, if they do not already exist there, while standard channels can be added as many times as needed, whenever the type of the channel does not change. Thus, addition is a partial operation. It is implicitly assumed in the typing rules that the addition of environments is defined.

Definition 4.5. *Addition* on environments, denoted by $+$, is a partial binary operation defined inductively by the following rules:

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

T-NIL	$\dfrac{\Gamma \text{ completed}}{\Gamma \vdash \mathbf{0}}$	T-PAR	$\dfrac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}$
T-REP	$\dfrac{\Gamma \vdash P \quad \Gamma \text{ unlimited for } P}{\Gamma \vdash *P}$		
T-RES	$\dfrac{\Gamma, x : \#T \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x)\, P}$	T-RES-S	$\dfrac{\Gamma, x : S, y : \overline{S} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)\, P}$
T-IN	$\dfrac{\Gamma, x : \#T, y : T \vdash P}{\Gamma, x : \#T \vdash x?(y).P}$	T-IN-S	$\dfrac{\Gamma, x : S, y : T \vdash P}{\Gamma, x : ?T.S \vdash x?(y).P}$

T-OUT	$\dfrac{\Gamma, x : \#T \vdash P}{(\Gamma, x : \#T) + y : T \vdash x!\langle y \rangle.P}$
T-OUT-S	$\dfrac{\Gamma, x : S \vdash P}{(\Gamma, x : !T.S) + y : T \vdash x!\langle y \rangle.P}$
T-SELECT	$\dfrac{j \in I \qquad \Gamma, x : S_j \vdash P}{\Gamma, x : \oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I} \vdash x \triangleleft \mathbf{l}_j.P}$
T-BRANCH	$\dfrac{\forall i \in I \qquad \Gamma, x : S_i \vdash P_i}{\Gamma, x : \&\langle \mathbf{l}_i : S_i \rangle_{i \in I} \vdash x \triangleright \{\mathbf{l}_i : P_i\}_{i \in I}}$

Table 4.1: Typing rules for session-calculus

Suppose $\Gamma \coloneqq x : S_1, z : \#T_1$. Then,

$\Gamma, z : \#T_1$ and $\Gamma + z : \#T_2$ are not defined,
$\Gamma + z : \#T_1 = \Gamma$,
$\Gamma, x : S_1$ and $\Gamma + x : S_1$ and $\Gamma + x : S_2$ are not defined,
$\Gamma + y : S_2 = \Gamma, y : S_2$,
$\Gamma + z : \#T_1, v : \#T_2, y : S_2 = \Gamma, v : \#T_2, y : S_2$,
$\Gamma + z : \#T_1, v : \#T_2, x : S_1$ is not defined.

We now have all the machinery in place to present the typing rules. We call a proof of a type judgement a *typing derivation*, which is viewed as a tree, whose leaves are usages of rule T-NIL and root is the judgement that is derived. We explain every rule below and give a few example derivations.

Definition 4.6. We say that process P is *well-typed* (or *correctly typed*) in Γ, if $\Gamma \vdash P$ can be inferred from the rules in Table 4.1. If P is well-typed in Γ, we say that $\Gamma \vdash P$ is *valid* and often write just $\Gamma \vdash P$. Otherwise we say that $\Gamma \vdash P$ is *invalid* and P is *ill typed* or *incorrectly typed* in Γ.

Note. If $\Gamma, x : T \vdash P$, then x is either free in P or there are no occurrences of x in P. This follows from the variable convention.

For example, $x : \#T \vdash (\boldsymbol{\upsilon} x)\, P$ is ill formed, because the first occurrence of x is free and the second is bound.

T-SELECT and T-BRANCH are rules for typechecking selection and branching, respectively. Selection $x \triangleleft \mathbf{l}_j.P$, where the type of x is $\oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I}$, is well-typed, if the label exists and the continuation process P uses x with continuation type S_j. Branching $x \triangleright \{\mathbf{l}_i : P_i\}_{i \in I}$, where the type of x is $\&\langle \mathbf{l}_i : S_i \rangle_{i \in I}$, is well-typed, if each of the continuation processes P_i, $i \in I$, use x with continuation type S_i.

Rules T-IN and T-IN-S are used to typecheck input for standard channels and session channels, respectively. Input $x?(y).P$, where the type of x is either $\#T$ or $?T.S$, is well-typed, if the type of y is T and continuation process P uses x with the same type $\#T$ for standard channels or with continuation type S for session channels.

Rules for output $x!\langle y \rangle.P$, T-OUT and T-OUT-S, are similar to the rules for input. However, this time $y : T$ is added to the environment in the conclusion. Note that addition is not defined, if already existing session channel is added. Therefore, T-OUT and T-OUT-S prevent sending a session channel, if it is already in the environment. This guarantees that a session channel occurring in the continuation process P is not sent to another subprocess running in parallel.

T-RES and T-RES-S are rules for standard and session channel restriction, respectively. Session channel restriction $(\boldsymbol{\upsilon} xy)\, P$ is well-typed only, when x and y have dual (session) types and process P uses them with those types.

Rule T-PAR is used to typecheck composition $P \mid Q$; It is well-typed, if the environment can be split to two such parts, Γ_1 and Γ_2, that P is well-typed in Γ_1 and Q is well typed in Γ_2. Again, addition of environments Γ_1 and Γ_2 is not defined, if they both contain same session channel. This guarantees (together with the rules for output) that no session channel is used in two parallel subprocesses.

Replication $*P$ is well-typed in Γ by T-REP, if P is well-typed in Γ and Γ is unlimited for P.

Finally, T-NIL is the base rule; Inaction $\mathbf{0}$ is well-typed in every environment containing no session types except *end*. T-NIL ensures session channels are not discarded before fully used. However, as pointed out in [5], the type system does not prevent deadlocks. For example, type judgement

$$x :?\#T.\mathit{end}, y :!\#T.\mathit{end}, u :?\#T.\mathit{end}, v :!\#T.\mathit{end}, a : \#T \vdash x?(z).v!\langle a\rangle.\mathbf{0} \mid u?(w).y!\langle a\rangle.\mathbf{0},$$

where x, y are co-variables and so are u, v, is valid, but the process is deadlocked.

Example 4.7. We show a type derivation for the previous type judgement. Only one of the parallel subprocesses is derived, because the derivations are similar.

By T-PAR we can derive the previous type judgment from the two jugdements below

$$x :?\#T.\mathit{end}, v :!\#T.\mathit{end} + a : \#T \vdash x?(z).v!\langle a\rangle.\mathbf{0}$$
$$y :!\#T.\mathit{end}, u :?\#T.\mathit{end} + a : \#T \vdash u?(w).y!\langle a\rangle.\mathbf{0}$$

The derivation for the first judgement is

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathit{end}, z : \#T, v : \mathit{end}}}{x : \mathit{end}, z : \#T, v : \mathit{end} \vdash \mathbf{0}}\ \text{T-NIL}}{x : \mathit{end}, z : \#T, v :!\#T.\mathit{end} + a : \#T \vdash v!\langle a\rangle.\mathbf{0}}\ \text{T-OUT-S}}{x :?\#T.\mathit{end}, v :!\#T.\mathit{end} + a : \#T \vdash x?(z).v!\langle a\rangle.\mathbf{0}}\ \text{T-IN-S}$$

Note that in T-PAR $a : \#T$ can be in both sides, because $\#T$ is standard channel type. Therefore, the addition in the conclusion is defined. In addition, T-NIL could not have been used, if $\#T$ was not a standard channel type or *end*.

Another form of deadlock is, when both ends of a session channel occur in the same thread. For example,

$$v : \mathit{end} \vdash (\boldsymbol{\nu} xy)\, x!\langle v\rangle.y?(u).\mathbf{0}$$

is valid, but the process is deadlocked. The typechecking algorithm in [5] eliminates deadlocks of this form. However, it does not eliminate deadlocks of the previous form.

Example 4.8. Type derivation for the previous judgement is

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \mathit{end}, y : \mathit{end}, u : \mathit{end}}{x : \mathit{end}, y : \mathit{end}, u : \mathit{end} \vdash \mathbf{0}}\ \text{T-NIL}}{x : \mathit{end}, y : ?\mathit{end}.\mathit{end} \vdash y?(u).\mathbf{0}}\ \text{T-IN-S}}{x : !\mathit{end}.\mathit{end}, y : ?\mathit{end}.\mathit{end}, v : \mathit{end} \vdash x!\langle v\rangle.y?(u).\mathbf{0}}\ \text{T-OUT-S}}{v : \mathit{end} \vdash (\boldsymbol{\upsilon} xy)\, x!\langle v\rangle.y?(u).\mathbf{0}}\ \text{T-RES-S}$$

A process can be well-typed in one environment and ill typed in another:

Example 4.9. Consider process $P \coloneqq x?(u).\mathbf{0} \mid x?(v).\mathbf{0}$ and environments $\Gamma_1 \coloneqq x : ?T.\mathit{end}$ and $\Gamma_2 \coloneqq x : \#T$. Then, P is ill typed in Γ_1. Further, P is well-typed in Γ_2, if T is a standard channel type or *end*.

Addition $x : \#T + x : \#T = x : \#T$ is defined, because x is a standard channel. Therefore, we can derive

$$\dfrac{\dfrac{\dfrac{x : \#T, u : T}{x : \#T, u : T \vdash \mathbf{0}}\ \text{T-NIL}}{x : \#T \vdash x?(u).\mathbf{0}}\ \text{T-IN} \qquad \dfrac{\dfrac{x : \#T, v : T}{x : \#T, v : T \vdash \mathbf{0}}\ \text{T-NIL}}{x : \#T \vdash x?(v).\mathbf{0}}\ \text{T-IN}}{x : \#T + x : \#T \vdash x?(u).\mathbf{0} \mid x?(v).\mathbf{0}}\ \text{T-PAR}$$

Therefore, $\Gamma_2 \vdash P$ is valid. Again, T-NIL could not have been used, if T was not a standard channel type or *end*.

Typing P in Γ_1 would require splitting the environment in two parts. Both parts should contain $x : ?T.\mathit{end}$. However, addition $x : ?T.\mathit{end} + x : ?T.\mathit{end}$ is not defined and thus T-PAR cannot be used. Therefore, P is not well-typed in Γ_1.

Example 4.10. We show that

$$a : \#S_{server} \vdash StringServer$$

(see Example 4.3) is valid. We assume types and expressions for integers and strings are added to the syntax and omit the details. We write S S_1, S_2, len and cc for S_{server}, $S_{server1}$, $S_{server2}$, $length$ and $concat$, respectively.

The environment $a : \#S$ contains no session types and is thus unlimited. Therefore, rule T-REP can be used as the last rule in the derivation below. We show the derivation in three parts; from usage of T-BRANCH to T-REP and separately to both branches from T-NIL until T-BRANCH.

The last part of the derivation is

$$\dfrac{\dfrac{\dfrac{a : \#S, z : S_1 \vdash P_{len} \quad a : \#S, z : S_2 \vdash P_{cc}}{a : \#S, z : \&\langle \mathbf{len} : S_1, \mathbf{cc} : S_2 \rangle \vdash z \triangleright \{\mathbf{len} : P_{len}, \mathbf{cc} : P_{cc}\}}\ \text{T-BRANCH}}{a : \#S \vdash a?(z).z \triangleright \{\mathbf{len} : P_{len}, \mathbf{cc} : P_{cc}\}}\ \text{T-IN}}{a : \#S \vdash *Thread}\ \text{T-REP}$$

where $P_{len} \coloneqq z?(str).z!\langle |str| \rangle.\mathbf{0}$ and $P_{cc} \coloneqq z?(str_1).z?(str_2).z!\langle str_1 + str_2 \rangle.\mathbf{0}$ are the continuation processes for labels len and cc. Derivation for the len branch is

$$\dfrac{\dfrac{\dfrac{a : \#S, z : \mathit{end}, str : \mathit{string}}{a : \#S, z : \mathit{end}, str : \mathit{string} \vdash \mathbf{0}}\ \text{T-NIL}}{a : \#S, z : !\mathit{int}.\mathit{end}, str : \mathit{string} \vdash z!\langle |str| \rangle.\mathbf{0}}\ \text{T-OUT-S}}{a : \#S, z : ?\mathit{string}.!\mathit{int}.\mathit{end} \vdash z?(str).z!\langle |str| \rangle.\mathbf{0}}\ \text{T-IN-S}$$

and for cc branch

$$\dfrac{\dfrac{\dfrac{\dfrac{a : \#S, z : \mathit{end}, str_1 : \mathit{string}, str_2 : \mathit{string}}{a : \#S, z : \mathit{end}, str_1 : \mathit{string}, str_2 : \mathit{string} \vdash \mathbf{0}}}{a : \#S, z : !\mathit{string}.\mathit{end}, str_1 : \mathit{string}, str_2 : \mathit{string} \vdash z!\langle str_1 + str_2 \rangle.\mathbf{0}}}{a : \#S, z : ?\mathit{string}.!\mathit{string}.\mathit{end}, str_1 : \mathit{string} \vdash z?(str_2).z!\langle str_1 + str_2 \rangle.\mathbf{0}}}{a : \#S, z : ?\mathit{string}.?\mathit{string}.!\mathit{string}.\mathit{end} \vdash z?(str_1).z?(str_2).z!\langle str_1 + str_2 \rangle.\mathbf{0}}$$

5 Type soundness

In this section we prove type preservation with respect to operational semantics. Furthermore, we prove that well-typed processes use session types in a correct manner. Together these two results form type soundness. We first prove a few properties to help proving the main results.

We follow a similar process for proving type soundness as in [5] and [16]. The theorems are mostly adapted from [5]. However, due to differences in the syntax, especially in the syntax for session restriction, proofs are given. As this thesis is targeted for a new reader, proofs are very detailed. This, we believe, makes it easier to understand the theory and gives confidence for the reader to continue her adventure in the world of session types.

In [5] subtypes and recursive types are defined for the language. We have not included those, which makes the proofs slightly simpler. The biggest difference between our language and the language in [5] is in the syntax of session channels and reduction. We have used co-variables together with separate syntax for session restriction and allowed communication for session channels only inside a restriction. In [5] optional polarities for channels are used and reductions are annotated. Furthermore, communication between session channels is possible without a restriction. Our choice, in our opinion, makes the difference between standard channels and session channels clearer. In addition, it simplifies the proofs and makes them easier to follow.

Firstly we show that, if a process is correctly typed in an environment, then every free name of the process must be in the environment. We continue by showing that in certain circumstances it is possible to weaken or strengthen an environment, that is, to add assignments to the environment or remove assignments from the environment, respectively.

Lemma 5.1. *If* $\Gamma \vdash P$*, then* ***fn***$(P) \subseteq dom(\Gamma)$*.*

Proof. By induction on the derivation of $\Gamma \vdash P$. Induction hypothesis (IH) is, that the statement holds for the premises. Rule T-NIL is the base case.

T-NIL: Clearly $\mathbf{fn}(\mathbf{0}) = \emptyset \subseteq dom(\Gamma)$.

T-PAR: By IH $\mathbf{fn}(P) \subseteq dom(\Gamma_1)$ and $\mathbf{fn}(Q) \subseteq dom(\Gamma_2)$. Then,

$$\mathbf{fn}(P \mid Q) = \mathbf{fn}(P) \cup \mathbf{fn}(Q) \subseteq dom(\Gamma_1) \cup dom(\Gamma_2) = dom(\Gamma_1 + \Gamma_2).$$

T-REP: By IH $\mathbf{fn}(P) \subseteq dom(\Gamma)$. Then, $\mathbf{fn}(*P) = \mathbf{fn}(P) \subseteq dom(\Gamma)$.

T-RES: By IH $\mathbf{fn}(P) \subseteq dom(\Gamma, x : \#T)$. Then,

$$\mathbf{fn}((\boldsymbol{\upsilon} x)\, P) = \mathbf{fn}(P) - \{x\} \subseteq dom(\Gamma, x : \#T) - \{x\} = dom(\Gamma).$$

T-RES-S: Similar to T-RES.

T-IN: Similar to T-IN-S.

T-IN-S: By IH $\mathbf{fn}(P) \subseteq dom(\Gamma, x : S, y : T)$. Then,

$$\begin{aligned}
\mathbf{fn}(x?(y).P) &= (\mathbf{fn}(P) \cup \{x\}) - \{y\} \\
&\subseteq (dom(\Gamma, x : S, y : T) \cup \{x\}) - \{y\} \\
&= dom(\Gamma, x : S, y : T) - \{y\} \\
&= dom(\Gamma, x :?T.S).
\end{aligned}$$

T-OUT: By IH $\mathbf{fn}(P) \subseteq dom(\Gamma, x : \#T)$. Then,

$$\begin{aligned}
\mathbf{fn}(x!\langle y\rangle.P) &= \mathbf{fn}(P) \cup \{x, y\} \\
&\subseteq dom(\Gamma, x : \#T) \cup \{x, y\} \\
&= dom(\Gamma, x : \#T) \cup \{y\} \\
&= dom((\Gamma, x : \#T) + y : T).
\end{aligned}$$

T-OUT-S: Similar to T-OUT.

T-SELECT: Similar to T-BRANCH.

T-BRANCH: By IH $\mathbf{fn}(P_i) \subseteq dom(\Gamma, x : S_i)$, for all $i \in I$. Then,

$$\begin{aligned}
\mathbf{fn}(x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I}) &= \bigcup_{i\in I} \mathbf{fn}(P_i) \cup \{x\} \\
&\subseteq \bigcup_{i\in I} dom(\Gamma, x : S_i) \cup \{x\} \\
&= dom(\Gamma, x : S_i) \text{ for any } i \in I \\
&= dom(\Gamma, x : \&\langle \mathbf{l}_i : S_i\rangle_{i\in I}).
\end{aligned}$$

□

Lemma 5.2 (**Weakening**). *Let $\Gamma \vdash P$. If U is a standard channel type or end and $\Gamma, z : U$ is defined, then $\Gamma, z : U \vdash P$.*

Proof. By induction on the derivation of $\Gamma \vdash P$. We show proofs for cases T-NIL, T-OUT, T-REP and T-PAR. Other cases are similar to T-OUT. Note that by Lemma 5.1 $\mathbf{fn}(P) \subseteq dom(\Gamma)$. Therefore, $z \notin \mathbf{fn}(P)$.

T-NIL: Now $\Gamma \vdash \mathbf{0}$ and Γ is completed. Clearly $\Gamma, z : U$ is completed, when Γ is completed. Therefore, we can derive $\Gamma, z : U \vdash \mathbf{0}$.

T-REP: The only derivation for $\Gamma \vdash *P$ is

$$\frac{\Gamma \vdash P \quad \Gamma \text{ unlimited for } P}{\Gamma \vdash *P}\ \text{T-REP}$$

Now, $\Gamma, z : U$ is unlimited for P, because $z \notin \mathbf{fn}(P)$. Then, by IH $\Gamma, z : U \vdash P$ and we can derive

$$\Gamma, z : U \vdash *P.$$

T-PAR: Derivation for $\Gamma \vdash P \mid Q$ is

$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}$$

where $\Gamma = \Gamma_1 + \Gamma_2$. By IH $\Gamma_1, z : U \vdash P$. Then, we can derive

$$\frac{\Gamma_1, z : U \vdash P \quad \Gamma_2 \vdash Q}{(\Gamma_1, z : U) + \Gamma_2 \vdash P \mid Q}$$

where $(\Gamma_1, z : U) + \Gamma_2 = (\Gamma_1 + \Gamma_2), z : U = \Gamma, z : U$.

T-OUT: Derivation for $\Gamma \vdash x!\langle y\rangle.P$ is

$$\frac{\Gamma_1, x : \#T \vdash P}{(\Gamma_1, x : \#T) + y : T \vdash x!\langle y\rangle.P}$$

where $\Gamma = (\Gamma_1, x : \#T) + y : T$. Because $z \notin \mathbf{fn}(x!\langle y\rangle.P)$, z cannot be x nor y. Then, by IH $\Gamma_1, z : U, x : \#T \vdash P$ and we can derive

$$\frac{\Gamma_1, x : \#T \vdash P}{(\Gamma_1, z : U, x : \#T) + y : T \vdash x!\langle y\rangle.P}$$

where

$$(\Gamma_1, z : U, x : \#T) + y : T = ((\Gamma_1, x : \#T) + y : T), z : U = \Gamma.$$

□

Corollary 5.3. *Let* $\Gamma_1 \vdash P$ *and* Γ_2 *is completed.*

1. *If* Γ_1, Γ_2 *is defined, then* $\Gamma_1, \Gamma_2 \vdash P$*.*
2. *If* $\Gamma_1 + \Gamma_2$ *is defined, then* $\Gamma_1 + \Gamma_2 \vdash P$*.*

Lemma 5.4 (**Strengthening**)**.** *Let* $\Gamma, z : T \vdash P$ *and* $z \notin$ ***fn***(P)*.*

(1) If T *is a session type, then* $T =$ *end.*

(2) $\Gamma \vdash P$*.*

Proof. By induction on the derivation of $\Gamma, z : T \vdash P$. We show proofs for cases T-NIL and T-PAR below. For other cases we can simply replace Γ with $\Gamma, z : T$ in the typing rules, because $z \notin$ **fn**(P). Then, by induction hypothesis the judgements in the premises are valid for the environment without $z : T$ and we can derive $\Gamma \vdash P$ (2). Furthermore, by induction hypothesis T has to be a standard channel type or *end* in the premises and, thus, in the conclusion (1).

T-NIL: Because $\Gamma, z : T \vdash \mathbf{0}$, environment $\Gamma, z : T$ has to be completed. Then, T is either a standard channel type or *end* (1). Further, Γ is also completed and, thus, $\Gamma \vdash \mathbf{0}$ (2).

T-PAR: Now

$$\Gamma, z : T \vdash P \mid Q,$$
$$\Gamma_1 \vdash P \text{ and}$$
$$\Gamma_2 \vdash Q,$$

where $\Gamma_1 + \Gamma_2 = \Gamma, z : T$ and $z \notin$ **fn**$(P \mid Q)$. Then, if T is standard channel type, $\Gamma_1(z)$ is defined, $\Gamma_2(z)$ is defined or both of them are defined. If T is a session type, either $\Gamma_1(z)$ or $\Gamma_2(z)$ is defined. Let us assume $\Gamma_1(z)$ is defined and $\Gamma_2(z)$ is not (other cases are similar). We can write $\Gamma_1 = \Gamma_1', z : T$. Because $z \notin$ **fn**$(P \mid Q)$, $z \notin$ **fn**(P) either. Thus, by IH for (2) $\Gamma_1' \vdash P$ and by T-PAR

$$\Gamma_1' + \Gamma_2 \vdash P \mid Q.$$

Furthermore, $\Gamma_1' + \Gamma_2 = \Gamma$ and, therefore, $\Gamma \vdash P \mid Q$ (2). If T is a session type, then by IH (in $\Gamma_1', z : T \vdash P$) for (1) T must be *end*.

□

Ideally an environment should contain assignments for free names of a process only. Strengthening allows us to remove assignments from an environment one by one to get rid of assignments, which are not needed. The next result follows.

Corollary 5.5. *If $\Gamma \vdash P$ and Γ is unlimited for P, then there exists an unlimited environment Γ' such that $\Gamma' \vdash P$.*

Before proving the main theorems, we now prove results for substitution and structural congruence. The result for substitution is needed, when proving preservation under reduction rules R-COM and R-COM-SESS. Preservation for structural congruence is needed to prove preservation under reduction rule R-STRUCT and to prove type safety.

Lemma 5.6 (**Substitution**). *If $\Gamma, v : T \vdash P$ and $\Gamma + z : T$ is defined, then $\Gamma + z : T \vdash P\{z/v\}$.*

Proof. By induction on the derivation of $\Gamma, v : T \vdash P$.

T-NIL: Clearly $\Gamma + z : T$ is completed, because $\Gamma, v : T$ is completed. In addition, $\mathbf{0}\{z/v\} = \mathbf{0}$. Then, by T-NIL $\Gamma + z : T \vdash \mathbf{0}\{z/v\}$.

T-PAR: Now

$$\Gamma, v : T \vdash P \mid Q,$$
$$\Gamma_1 \vdash P \text{ and}$$
$$\Gamma_2 \vdash Q,$$

where $\Gamma_1 + \Gamma_2 = \Gamma, v : T$. Simliarly to the proof of Lemma 5.4 let us assume $\Gamma_1(v)$ is defined and $\Gamma_2(v)$ is not (other cases are similar) and write $\Gamma_1 = \Gamma_1', v : T$. By Lemma 5.1 $\mathbf{fn}(Q) \subseteq dom(\Gamma_2)$. Because v is not in Γ_2, v is not in $\mathbf{fn}(Q)$ and, then, $Q\{z/v\} = Q$. On the other hand, $\Gamma_1', v : T \vdash P$ and clearly $\Gamma_1' + z : T$ is defined. Then by IH

$$\Gamma_1' + z : T \vdash P\{z/v\}.$$

By T-PAR we get

$$(\Gamma_1' + z : T) + \Gamma_2 \vdash P\{z/v\} \mid Q\{z/v\},$$

where

$$P\{z/v\} \mid Q\{z/v\} = (P \mid Q)\{z/v\} \text{ and}$$
$$(\Gamma_1' + z : T) + \Gamma_2 = (\Gamma_1' + \Gamma_2) + z : T = \Gamma + z : T.$$

T-REP: Clearly $\Gamma + z : T$ is unlimited for P, when $\Gamma, v : T$ is unlimited for P. Then, by IH $\Gamma + z : T \vdash P\{z/v\}$ and by T-REP

$$\Gamma + z : T \vdash *P\{z/v\}$$

and $*P\{z/v\} = (*P)\{z/v\}$.

T-RES: Now

$$\Gamma, v : T \vdash (\boldsymbol{\nu} x)\, P \text{ and}$$
$$\Gamma, v : T, x : \#U \vdash P.$$

By IH

$$\Gamma, x : \#U + z : T \vdash P\{z/v\},$$

where $\Gamma, x : \#U + z : T = (\Gamma + z : T), x : \#U$ is defined, because $\Gamma + z : T$ is defined. Then, by T-RES

$$\Gamma + z : T \vdash (\boldsymbol{\nu} x)\, P\{z/v\}$$

and $(\boldsymbol{\nu} x)\, P\{z/v\} = ((\boldsymbol{\nu} x)\, P)\{z/v\}$.

T-RES-S: Similar to T-RES.

T-IN: Similar to T-IN-S.

T-IN-S: Now $\Gamma, v : T \vdash x?(y).P$. There are two possibilities for v: either $v = x$ or $v \neq x$.

($v = x$) We have

$$\Gamma_1, v :?U.S \vdash v?(y).P \text{ and}$$
$$\Gamma_1, v : S, y : U \vdash P.$$

By IH

$$\Gamma_1, y : U + z : S \vdash P\{z/v\},$$

where

$$\Gamma_1, y : U + z : S = \Gamma_1, z : S, y : U$$

is defined, because $\Gamma_1 + z :?U.S$ is defined. Then, by T-IN-S

$$\Gamma_1, z :?U.S \vdash z?(y).P\{z/v\},$$

where

$$\Gamma_1, z :?U.S = \Gamma_1 + z :?U.S \text{ and}$$
$$z?(y).P\{z/v\} = (v?(y).P)\{z/v\}.$$

($v \neq x$) Now

$$\Gamma_2, x :?U.S, v : T \vdash x?(y).P \text{ and}$$
$$\Gamma_2, x : S, y : U, v : T \vdash P.$$

By IH

$$\Gamma_2, x : S, y : U + z : T \vdash P\{z/v\},$$

where

$$\Gamma_2, x : S, y : U + z : T = (\Gamma_2 + z : T), x : S, y : U$$

is defined, because $\Gamma_2, x :?U.S + z : T$ is defined. Then, by T-IN-S

$$(\Gamma_2 + z : T), x :?U.S \vdash x?(y).P\{z/v\}$$

and $(\Gamma_2 + z : T), x :?U.S = \Gamma_2, x :?U.S + z : T$.

T-OUT: Similar to T-OUT-S.

T-OUT-S: Now $\Gamma, v : T \vdash x!\langle y\rangle.P$. There are three possibilities for v: $v = x$, $v = y$ or $v \neq x, y$.

($v = x$) We have

$$(\Gamma_1, v :!U.S) + y : U \vdash v!\langle y\rangle.P \text{ and}$$
$$\Gamma_1, v : S \vdash P.$$

By IH

$$\Gamma_1 + z : S \vdash P\{z/v\},$$

where

$$\Gamma_1 + z : S = \Gamma_1, z : S$$

is defined, because $(\Gamma_1 + y : U) + z :!U.S$ is defined. Then, by T-OUT-S

$$(\Gamma_1, z :!U.S) + y : U \vdash z!\langle y\rangle.P\{z/v\},$$

where

$$(\Gamma_1, z :!U.S) + y : U = (\Gamma_1 + y : U) + z :!U.S \text{ and}$$
$$z!\langle y\rangle.P\{z/v\} = (v!\langle y\rangle.P)\{z/v\}.$$

($v = y$) In this case

$$(\Gamma_2, x :!U.S) + v : U \vdash x!\langle v\rangle.P \text{ and}$$
$$\Gamma_2, x : S \vdash P.$$

Then, either v is in $dom(\Gamma_2)$ or v is not in $dom(\Gamma_2)$.

($v \in dom(\Gamma_2)$) Let us write $\Gamma_2 = \Gamma_2', v : U$. By IH

$$(\Gamma_2', x : S) + z : U \vdash P\{z/v\},$$

where

$$(\Gamma_2', x : S) + z : U = (\Gamma_2' + z : U), x : S$$

is defined, because $(\Gamma_2', x :!U.S) + z : U$ is defined. Then, by T-OUT-S

$$((\Gamma_2' + z : U), x : S) + y : U \vdash x!\langle y\rangle.P\{z/v\}$$

and $((\Gamma_2' + z : U), x : S) + y : U = ((\Gamma_2', x :!U.S) + y : U) + z : U$.

($v \notin dom(\Gamma_2)$) Because v is not in $dom(\Gamma_2)$, $v \notin$ **fn**(P) by Lemma 5.1. Then, $P\{z/v\} = P$. By T-OUT-S

$$(\Gamma_2, x :!U.S) + z : U \vdash x!\langle z\rangle.P$$

and $x!\langle z\rangle.P = (x!\langle v\rangle.P)\{z/v\}$.

($v \neq x, y$) Now

$$(\Gamma_3, v : T, x :!U.S) + y : U \vdash x!\langle y\rangle.P \text{ and}$$
$$\Gamma_3, v : T, x : S \vdash P.$$

By IH

$$(\Gamma_3, x : S) + z : T \vdash P\{z/v\},$$

where

$$(\Gamma_3, x : S) + z : T = (\Gamma_3 + z : T), x : S$$

is defined, because $((\Gamma_3, x :!U.S) + y : U) + z : T$ is defined. Then, by T-OUT-S

$$((\Gamma_3 + z : T), x :!U.S) + y : U \vdash x!\langle y\rangle.P\{z/v\}$$

and $((\Gamma_3 + z : T), x :!U.S) + y : U = ((\Gamma_3, x :!U.S) + y : U) + z : T$.

T-SELECT: Now $\Gamma, v : T \vdash x \triangleleft \mathbf{l}_j.P$. Then, there are two possibilities for v: either $v = x$ or $v \neq x$.

($v = x$) In this case

$$\Gamma_1, v : \oplus\langle \mathbf{l}_i : S_i\rangle_{i\in I} \vdash v \triangleleft \mathbf{l}_j.P \text{ and}$$
$$\Gamma_1, v : S_j \vdash P,$$

where $j \in I$. By IH

$$\Gamma_1 + z : S_j \vdash P\{z/v\},$$

where

$$\Gamma_1 + z : S_j = \Gamma_1, z : S_j$$

is defined, because $\Gamma_1 + z : \oplus\langle \mathbf{l}_i : S_i\rangle_{i\in I}$ is defined. Then, by T-SELECT

$$\Gamma_1, z : \oplus\langle \mathbf{l}_i : S_i\rangle_{i\in I} \vdash z \triangleleft \mathbf{l}_j.P\{z/v\},$$

where

$$\Gamma_1, z : \oplus\langle \mathbf{l}_i : S_i\rangle_{i\in I} = \Gamma_1 + z : \oplus\langle \mathbf{l}_i : S_i\rangle_{i\in I} \text{ and}$$
$$z \triangleleft \mathbf{l}_j.P\{z/v\} = (v \triangleleft \mathbf{l}_j.P)\{z/v\}.$$

($v \neq x$) Now

$$\Gamma_2, x : \oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I}, v : T \vdash x \triangleleft \mathbf{l}_j.P \text{ and}$$
$$\Gamma_2, x : S_j, v : T \vdash P,$$

where $j \in I$. By IH

$$\Gamma_2, x : S_j + z : T \vdash P\{z/v\},$$

where

$$\Gamma_2, x : S_j + z : T = (\Gamma_2 + z : T), x : S_j$$

is defined, because $\Gamma_2, x : \oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I} + z : T$ is defined. Then, by T-IN-S

$$(\Gamma_2 + z : T), x : \oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I} \vdash x \triangleleft \mathbf{l}_j.P\{z/v\}$$

and $(\Gamma_2 + z : T), x : \oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I} = (\Gamma_2, x : \oplus\langle \mathbf{l}_i : S_i \rangle_{i \in I}) + z : T$.

T-BRANCH: Similar to T-SELECT.

□

Lemma 5.7 (**Preservation for structural congruence**). *If* $\Gamma \vdash P$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q$*.*

Proof. By induction on the derivation of $P \equiv Q$. Base cases are the axioms. Inductive cases are the congruence rules other than reflexivity, which is trivial. Because of symmetry of congruence relation, we have to check both directions of the axioms, left-to-right (1) and right-to-left (2).

SC-ASSOC: $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.

1. Derivation for $\Gamma \vdash P \mid (Q \mid R)$ is

$$\dfrac{\Gamma_P \vdash P \quad \dfrac{\Gamma_Q \vdash Q \quad \Gamma_R \vdash R}{\Gamma_Q + \Gamma_R \vdash Q \mid R}\,\text{T-PAR}}{\Gamma_P + (\Gamma_Q + \Gamma_R) \vdash P \mid (Q \mid R)}\,\text{T-PAR}$$

where $\Gamma = \Gamma_P + (\Gamma_Q + \Gamma_R)$. Then, we can derive

$$\dfrac{\dfrac{\Gamma_P \vdash P \quad \Gamma_Q \vdash Q}{\Gamma_P + \Gamma_Q \vdash P \mid Q}\,\text{T-PAR} \quad \Gamma_R \vdash R}{(\Gamma_P + \Gamma_Q) + \Gamma_R \vdash (P \mid Q) \mid R}\,\text{T-PAR}$$

where

$$(\Gamma_P + \Gamma_Q) + \Gamma_R = \Gamma_P + (\Gamma_Q + \Gamma_R) = \Gamma.$$

2. Similar to the other direction.

SC-COMM: Similar to SC-ASSOC.

SC-INACT: $P \mid \mathbf{0} \equiv P$.

1. Derivation for $\Gamma \vdash P \mid \mathbf{0}$ is

$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash \mathbf{0}}{\Gamma_1 + \Gamma_2 \vdash P \mid \mathbf{0}}\ \text{T-PAR}$$

where $\Gamma_1 + \Gamma_2 = \Gamma$. Because Γ_2 is completed, by Corollary 5.3

$$\Gamma_1 + \Gamma_2 \vdash P.$$

2. Now $\Gamma \vdash P$. Then, we can derive

$$\frac{\Gamma \vdash P \quad \vdash \mathbf{0}}{\Gamma \vdash P \mid \mathbf{0}}\ \text{T-PAR}$$

SC-RES: Similar to SC-RES-S-RES.

SC-RES-INACT: $(\boldsymbol{\nu} x)\, \mathbf{0} \equiv \mathbf{0}$.

1. The only derivation for $\Gamma \vdash (\boldsymbol{\nu} x)\, \mathbf{0}$ is

$$\frac{\Gamma, x : \#T \vdash \mathbf{0}}{\Gamma \vdash (\boldsymbol{\nu} x)\, \mathbf{0}}\ \text{T-RES}$$

Then, by Lemma 5.4 $\Gamma \vdash \mathbf{0}$.

2. Now $\Gamma \vdash \mathbf{0}$. Then, by Lemma 5.2 $\Gamma, x : \#T \vdash \mathbf{0}$ and we can derive

$$\frac{\Gamma, x : \#T \vdash \mathbf{0}}{\Gamma \vdash (\boldsymbol{\nu} x)\, \mathbf{0}}\ \text{T-RES}$$

SC-RES-COMP: Similar to SC-RES-S-COMP.

SC-REP: $*P \equiv P \mid *P$.

1. The only derivation for $\Gamma \vdash *P$ is

$$\frac{\Gamma \vdash P \quad \Gamma \text{ unlimited for } P}{\Gamma \vdash *P}\ \text{T-REP}$$

By Corollary 5.5 there exists an unlimited environment Γ' such, that $\Gamma' \vdash P$. Then,

$$\frac{\Gamma' \vdash P \quad \Gamma \vdash *P}{\Gamma' + \Gamma \vdash P \mid *P}\ \text{T-PAR}$$

where $\Gamma' + \Gamma = \Gamma$ is defined, because Γ' is unlimited.

2. Derivation for $\Gamma \vdash P \mid *P$ is

$$\frac{\Gamma_1 \vdash P \quad \dfrac{\Gamma_2 \vdash P \quad \Gamma_2 \text{ unlimited for } P}{\Gamma_2 \vdash *P}\ \text{T-REP}}{\Gamma \vdash P \mid *P}\ \text{T-PAR}$$

where $\Gamma = \Gamma_1 + \Gamma_2$. Because Γ_2 is completed and $\Gamma_1 + \Gamma_2$ is defined, then by Corollary 5.3 $\Gamma \vdash P$.

SC-RES-S-RES: $(\boldsymbol{\nu} xy)\,(\boldsymbol{\nu} z)\,P \equiv (\boldsymbol{\nu} z)\,(\boldsymbol{\nu} xy)\,P$.

1. The only derivation for $\Gamma \vdash (\boldsymbol{\nu} xy)\,(\boldsymbol{\nu} z)\,P$ is

$$\frac{\dfrac{\dfrac{\Gamma, x : S, y : \overline{S}, z : \#T \vdash P}{\Gamma, x : S, y : \overline{S} \vdash (\boldsymbol{\nu} z)\,P}\ \text{T-RES}}{\Gamma \vdash (\boldsymbol{\nu} xy)\,(\boldsymbol{\nu} z)\,P}\ \text{T-RES-S}}{}$$

Then, we can derive

$$\frac{\dfrac{\dfrac{\Gamma, x : S, y : \overline{S}, z : \#T \vdash P}{\Gamma, z : \#T \vdash (\boldsymbol{\nu} xy)\,P}\ \text{T-RES-S}}{\Gamma \vdash (\boldsymbol{\nu} z)\,(\boldsymbol{\nu} xy)\,P}\ \text{T-RES}}{}$$

2. Similar to the other direction.

SC-RES-S: Similar to SC-RES-S-RES.

SC-RES-S-INACT: $(\boldsymbol{\nu} xy)\,\mathbf{0} \equiv \mathbf{0}$.

1. The only derivation for $\Gamma \vdash (\boldsymbol{\nu} xy)\, \mathbf{0}$ is

$$\frac{\Gamma, x : S, y : \overline{S} \vdash \mathbf{0}}{\Gamma \vdash (\boldsymbol{\nu} xy)\, \mathbf{0}}\ \text{T-RES-S}$$

   Then, by Lemma 5.4 $S = \overline{S} =$ *end* and $\Gamma \vdash \mathbf{0}$.

2. Now $\Gamma \vdash \mathbf{0}$. If $\Gamma' \vdash (\boldsymbol{\nu} xy)\, \mathbf{0}$ for some environment Γ', then the only derivation is the same as above (for the other direction, where Γ is replaced by Γ'). Similarly, by Lemma 5.4 must be, that $S = \overline{S} =$ *end*. Then, we can derive

$$\frac{\dfrac{\Gamma, x : \textit{end}, y : \textit{end}\ \text{completed}}{\Gamma, x : \textit{end}, y : \textit{end} \vdash \mathbf{0}}\ \text{T-NIL}}{\Gamma \vdash (\boldsymbol{\nu} xy)\, \mathbf{0}}\ \text{T-RES-S}$$

SC-RES-S-COMP: $(\boldsymbol{\nu} xy)\, (P \mid Q) \equiv P \mid (\boldsymbol{\nu} xy)\, Q$ if $x, y \notin \mathbf{fn}(P)$.

1. Derivation for $\Gamma \vdash (\boldsymbol{\nu} xy)\, (P \mid Q)$ is

$$\frac{\dfrac{\Gamma_P \vdash P \quad \Gamma_Q \vdash Q}{\Gamma, x : S, y : \overline{S} \vdash P \mid Q}\ \text{T-PAR}}{\Gamma \vdash (\boldsymbol{\nu} xy)\, (P \mid Q)}\ \text{T-RES-S}$$

   where $\Gamma_P + \Gamma_Q = \Gamma, x : S, y : \overline{S}$. Then, x (and similarly y) is either in Γ_P or in Γ_Q. If x is in Γ_P, by Lemma 5.4 $S =$ *end* and $\Gamma'_P \vdash P$, where $\Gamma_P = \Gamma'_P, x : S$. On the other hand, by Lemma 5.2 $\Gamma_Q, x : S \vdash Q$. If x is in Γ_Q, then $\Gamma'_Q, x : S \vdash Q$, where $\Gamma_Q = \Gamma'_Q, x : S$. Therefore, there are such environments Γ''_P and Γ''_Q that

$$\begin{aligned}
&x, y \text{ not in } \Gamma''_P,\\
&\Gamma''_P \vdash P,\\
&\Gamma''_Q, x : S, y : \overline{S} \vdash Q \text{ and}\\
&\Gamma = \Gamma''_Q + \Gamma''_P.
\end{aligned}$$

   Then, we can derive

$$\frac{\Gamma''_P \vdash P \quad \dfrac{\Gamma''_Q, x : S, y : \overline{S} \vdash Q}{\Gamma''_Q \vdash (\boldsymbol{\nu} xy)\, Q}\ \text{T-RES-S}}{\Gamma''_P + \Gamma''_Q \vdash P \mid (\boldsymbol{\nu} xy)\, Q}\ \text{T-PAR}$$

2. Derivation for $\Gamma \vdash P \mid (\boldsymbol{\nu} xy)\, Q$ is

$$\dfrac{\Gamma_P \vdash P \quad \dfrac{\Gamma_Q, x : S, y : \overline{S} \vdash Q}{\Gamma_Q \vdash (\boldsymbol{\nu} xy)\, Q}\ \text{T-RES-S}}{\Gamma \vdash P \mid (\boldsymbol{\nu} xy)\, Q}\ \text{T-PAR}$$

where $\Gamma = \Gamma_P + \Gamma_Q$. Then,

$$\dfrac{\dfrac{\Gamma_P \vdash P \quad \Gamma_Q, x : S, y : \overline{S} \vdash Q}{\Gamma_P + (\Gamma_Q, x : S, y : \overline{S}) \vdash P \mid Q}\ \text{T-PAR}}{\Gamma \vdash (\boldsymbol{\nu} xy)\, (P \mid Q)}\ \text{T-RES-S}$$

where

$$\Gamma_P + (\Gamma_Q, x : S, y : \overline{S}) = (\Gamma_P + \Gamma_Q), x : S, y : \overline{S} = \Gamma, x : S, y : \overline{S}$$

is clearly defined.

Congruence rules: Symmetry and transitivity are straightforward. Let C be a context, $P = C[P'] \equiv C[Q'] = Q$, where $P' \equiv Q'$, and $\Gamma \vdash P$. In the derivation of $\Gamma \vdash P$ there is a sub-derivation for $\Gamma' \vdash P'$, where Γ' is an environment. Then, by induction hypothesis $\Gamma' \vdash Q'$. Therefore, replacing P' by Q' in the derivation of $\Gamma \vdash P$ we get a derivation for $\Gamma \vdash Q$.

□

Theorem 5.8 (**Preservation under reduction**). *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$*, then* $\Gamma \vdash Q$.

Proof. By induction on the derivation of $P \longrightarrow Q$. Base cases are R-COM, R-COM-SESS and R-SELECT.

R-COM: Derivation for $\Gamma \vdash x!\langle y \rangle.P \mid x?(z).Q$ is

$$\dfrac{\dfrac{\Gamma_P, x : \#T \vdash P}{(\Gamma_P, x : \#T) + y : T \vdash x!\langle y \rangle.P}\ \text{OUT} \quad \dfrac{\Gamma_Q, x : \#T, z : T \vdash Q}{\Gamma_Q, x : \#T \vdash x?(z).Q}\ \text{IN}}{\Gamma \vdash x!\langle y \rangle.P \mid x?(z).Q}\ \text{PAR}$$

where

$$\begin{aligned}\Gamma &= ((\Gamma_P, x : \#T) + y : T) + (\Gamma_Q, x : \#T) \\ &= (\Gamma_P, x : \#T) + ((\Gamma_Q, x : \#T) + y : T)\end{aligned}$$

and x has to be standard channel, because otherwise the addition of environments would not be defined. By Lemma 5.6

$$(\Gamma_Q, x : \#T) + y : T \vdash Q\{y/z\},$$

where $(\Gamma_Q, x : \#T) + y : T$ is defined, because Γ is defined. Then, we can derive

$$\frac{\Gamma_P, x : \#T \vdash P \quad (\Gamma_Q, x : \#T) + y : T \vdash Q\{y/z\}}{\Gamma \vdash P \mid Q\{y/z\}}\ \text{PAR}$$

where $\Gamma = (\Gamma_P, x : \#T) + ((\Gamma_Q, x : \#T) + y : T)$.

R-COM-SESS: Derivation for $\Gamma \vdash (\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q)$ is

$$\frac{\dfrac{\dfrac{\Gamma_P, x : S \vdash P}{(\Gamma_P, x :!T.S) + v : T \vdash x!\langle v\rangle.P}\ \text{OUT} \quad \dfrac{\Gamma_Q, y : \overline{S}, z : T \vdash Q}{\Gamma_Q, y :?T.\overline{S} \vdash y?(z).Q}\ \text{IN}}{\Gamma, x :!T.S, y :?T.\overline{S} \vdash x!\langle v\rangle.P \mid y?(z).Q}\ \text{PAR}}{\Gamma \vdash (\boldsymbol{\nu} xy)\,(x!\langle v\rangle.P \mid y?(z).Q)}\ \text{RES-SESS}$$

where

$$\begin{aligned}\Gamma, x :!T.S, y :?T.\overline{S} &= ((\Gamma_P, x :!T.S) + v : T) + (\Gamma_Q, y :?T.\overline{S}) \\ &= (\Gamma_P, x :!T.S) + ((\Gamma_Q, y :?T.\overline{S}) + v : T)\end{aligned}$$

and $\Gamma = (\Gamma_P + \Gamma_Q) + v : T$. By Lemma 5.6

$$(\Gamma_Q, y : \overline{S}) + v : T \vdash Q\{y/z\},$$

where $(\Gamma_Q, y : \overline{S}) + v : T$ is defined, because $\Gamma, x :!T.S, y :?T.\overline{S}$ is defined. Then, we can derive

$$\frac{\dfrac{\Gamma_P, x : S \vdash P \quad (\Gamma_Q, y : \overline{S}) + v : T \vdash Q\{y/z\}}{(\Gamma_P, x : S) + ((\Gamma_Q, y : \overline{S}) + v : T) \vdash P \mid Q\{y/z\}}\ \text{PAR}}{(\Gamma_P + \Gamma_Q) + v : T \vdash (\boldsymbol{\nu} xy)\,P \mid Q\{y/z\}}\ \text{RES-SESS}$$

where $(\Gamma_P + \Gamma_Q) + v : T = \Gamma$.

R-SELECT: Derivation for $\Gamma \vdash (\boldsymbol{\nu} xy)\,(x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I} \mid y \triangleleft \mathbf{l}_j.Q)$ is

$$\dfrac{\dfrac{\forall i \in I \quad \Gamma_P, x : S_i \vdash P_i}{\Gamma_P, x : S \vdash x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I}}\,\text{BRANCH} \quad \dfrac{j \in I \quad \Gamma_Q, x : \overline{S_j} \vdash Q}{\Gamma_Q, y : \overline{S} \vdash y \triangleleft \mathbf{l}_j.Q}\,\text{SEL}}{\dfrac{\Gamma, x : S, y : \overline{S} \vdash x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I} \mid y \triangleleft \mathbf{l}_j.Q}{\Gamma \vdash (\boldsymbol{\nu} xy)\,(x \triangleright \{\mathbf{l}_i : P_i\}_{i\in I} \mid y \triangleleft \mathbf{l}_j.Q)}\,\text{RES-SESS}}$$

where $S = \&\langle \mathbf{l}_i : S_i\rangle_{i\in I}$, $\overline{S} = \oplus\langle \mathbf{l}_i : \overline{S_i}\rangle_{i\in I}$ and

$$\begin{aligned}\Gamma, x : S, y : \overline{S} &= (\Gamma_P, x : S) + (\Gamma_Q, y : \overline{S}) \\ &= (\Gamma_P + \Gamma_Q), x : S, y : \overline{S}.\end{aligned}$$

Then,

$$\dfrac{\dfrac{\Gamma_P, x : S_j \vdash P_j \quad \Gamma_Q, x : \overline{S_j} \vdash Q}{(\Gamma_P, x : S_j) + (\Gamma_Q, y : \overline{S_j}) \vdash P_j \mid Q}\,\text{PAR}}{\Gamma_P + \Gamma_Q \vdash (\boldsymbol{\nu} xy)\,(P_j \mid Q)}\,\text{RES-SESS}$$

where $\Gamma_P + \Gamma_Q = \Gamma$.

R-PAR: Similar to R-RES-SESS.

R-RES: Similar to R-RES-SESS.

R-RES-SESS: Derivation for $\Gamma \vdash (\boldsymbol{\nu} xy)\,P$ is

$$\dfrac{\Gamma, x : S, y : \overline{S} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} xy)\,P}\,\text{T-RES-S}$$

Then, by induction hypothesis $\Gamma, x : S, y : \overline{S} \vdash Q$ and we can derive

$$\dfrac{\Gamma, x : S, y : \overline{S} \vdash Q}{\Gamma \vdash (\boldsymbol{\nu} xy)\,Q}\,\text{T-RES-S}$$

R-STRUCT: Now the reduction derivation ends with R-STRUCT:

$$\dfrac{P' \equiv P \quad P \longrightarrow Q \quad Q \equiv Q'}{P' \longrightarrow Q'}\,\text{R-STRUCT}$$

We have to prove $\Gamma \vdash Q'$, when $\Gamma \vdash P'$ is valid. By Lemma 5.7 $\Gamma \vdash P$, because $P' \equiv P$. Furthermore, by induction hypothesis $\Gamma \vdash Q$. Then, By

Lemma 5.7 $\Gamma \vdash Q'$, because $Q' \equiv Q$.

□

We have so far proved that whenever a correctly typed process reduces to a new process, the new process is correctly typed also. What can we say about a correctly typed process then? We have some expectations for session types, that is, the two ends of a channel need to behave dually and one end can occur in exactly one thread. The former result is proved in the next theorem and the latter follows immediately from the rules as mentioned in the previous section. Similar theorem and proof can be found in [16]. In that work the syntax is slightly different. The main difference is that we have not defined Boolean values, which makes type safety in our case simpler.

Definition 5.9. If x and y are co-variables, then processes of the form

$$a?(v).P \mid a!\langle u\rangle.Q,$$
$$x?(v).P \mid y!\langle u\rangle.Q,$$
$$x \triangleleft \mathbf{l}.P \mid y \triangleright \{\mathbf{l}_i : P_i\}_{i\in I}$$

are called *redexes*.

Theorem 5.10 (**Type safety**). *Let $\Gamma \vdash P_0$, where Γ is unlimited. For every process of the form $(\boldsymbol{\nu}\widetilde{u})\,(\boldsymbol{\nu}\widetilde{vw})\,(P \mid Q \mid R)$ structurally congruent with P_0 the following hold: if $xy \in \widetilde{vw}$, P is prefixed at x and Q is prefixed at y, then $P \mid Q$ is a redex.*

Proof. By Lemma 5.7 $\Gamma \vdash (\boldsymbol{\nu}\widetilde{u})\,(\boldsymbol{\nu}\widetilde{vw})\,(P \mid Q \mid R)$. Using structural congruence we can restructure P_0 to get

$$(\boldsymbol{\nu}\widetilde{u})\,(\boldsymbol{\nu}\widetilde{v'w'})\,((\boldsymbol{\nu}xy)\,(P \mid Q \mid R)),$$

where $\widetilde{v'w'}$ contains all the pairs in $\widetilde{vw}$ except xy. Then, again by Lemma 5.7 and by T-RES and T-RES-S repeatedly we get

$$\Gamma, u_1 : T_1, \ldots, v_1' : S_1, w_1' : \overline{S_1}, \ldots \vdash (\boldsymbol{\nu}xy)\,(P \mid Q \mid R),$$

where we write $\Gamma' \coloneqq \Gamma, u_1 : T_1, \ldots$ and $\Gamma_S \coloneqq v_1' : S_1, w_1' : \overline{S_1}, \ldots$ Furthermore, by T-PAR there can not be any occurrences of x or y in R and, then, by SC-RES-S-COMP

$$\Gamma', \Gamma_S \vdash (\boldsymbol{\nu}xy)\,(P \mid Q) \mid R.$$

Finally we have a derivation

$$\dfrac{\dfrac{\dfrac{\Gamma', \Gamma_P, x : S \vdash P \quad \Gamma', \Gamma_Q, y : \overline{S} \vdash Q}{\Gamma', \Gamma_{S_1}, x : S, y : \overline{S} \vdash (P \mid Q)}\ \text{T-PAR}}{\Gamma', \Gamma_{S_1} \vdash (\boldsymbol{\upsilon} xy)\,(P \mid Q)}\ \text{T-RES-S} \quad \Gamma', \Gamma_{S_2} \vdash R}{\Gamma', \Gamma_S \vdash (\boldsymbol{\upsilon} xy)\,(P \mid Q) \mid R}\ \text{T-PAR},$$

where $\Gamma_{S_1} + \Gamma_{S_2} = \Gamma_S$ and $\Gamma_P + \Gamma_Q = \Gamma_{S_1}$. Now the types of x and y need to be dual and, thus, $P \mid Q$ is a redex. □

We have now proved that any correctly typed process and any process it reduces to use session channels correctly. Therefore, processes interact in a correct manner in runtime. As noted before, deadlocks are still possible. However, the theorems proved in this section are powerful results and guarantee safety in interactions. Next section gives a simplified and informal example for a use case of session types.

6 Type-safe server-client programming

In this section we show a practical example to demonstrate, how session types can help developing server-client systems. We create a situation, where a server and a client are implemented, address a few problematic cases during the evolution of the system and explain how session types solve those problems. In addition, we discuss how text editors with support for session types can guide the developer for a nicer and faster developing experience. Again, for illustration purposes we extend the language quite a lot. However, the extensions should be easily understandable and followable. We use ... to denote local computations and implementations not of interest.

Let us assume that one developer creates a server along with documentation and another developer creates a client using the server. The first problem arises from the fact that there is no guarantee that the documentation and the implementation match. There might be something missing in the documentation, there could be misspelling or it has not been updated with latest changes. The only way for the client developer to find out the problems is to run her application and hope, that the response from the server tells, what the problem is. If we assume the server implementation to be exactly in line with the documentation, there are still numerous places, where a client implementation can go wrong: misspellings, incorrect data in messages and so on. Again the only way to find out is to run the application. With session types none of these problems would exist.

Let us first give an implementation for a server that can

- return a list of tasks,
- create a new task and
- delete an existing task.

We will start with the documentation. To fetch a list of tasks from the server, a client needs to send a GET request to path '/task'. The server will respond with status code 200 and a list of tasks or with status code 500, if an internal error occurs. A task is simply an object with fields 'id' and 'description'. For creating a new task a POST request to the same path is needed with data representing a task, that is, an object with field 'description'. The server will respond with status code 201, 400 or 500, when task is created successfully, server can not process the request or an internal error occurs, respectively. Deleting a task requires a

DELETE request and an id of a task. The server will respond with status code 204, 404 or 500, when task is deleted successfully, no task with the given id is found or an internal error occurs, respectively.

Recall $StringServer$ from Example 3.18. A similar implementation for a task server would be

$$
\begin{aligned}
TaskServer &= *Thread \\
Thread &= a?(z).z \triangleright \{/\mathbf{task} : TaskPath, \\
&\qquad\qquad /\mathbf{other} : OtherPath\} \\
TaskPath &= z \triangleright \{\mathbf{GET} : GetTask, \\
&\qquad\qquad \mathbf{POST} : PostTask, \\
&\qquad\qquad \mathbf{DELETE} : DeleteTask\} \\
OtherPath &= \ldots,
\end{aligned}
$$

where

$$
\begin{aligned}
GetTask = &[\text{if } \ldots] \; z \triangleleft \mathbf{200}.z!\langle Response(200, listOfTasks)\rangle.\mathbf{0} \\
&[\text{else}] \; z \triangleleft \mathbf{500}.z!\langle Response(500)\rangle.\mathbf{0}. \\
PostTask = z?(task).&[\text{if } \ldots] \; z \triangleleft \mathbf{201}.z!\langle Response(201)\rangle.\mathbf{0} \\
&[\text{else if } \ldots] \; z \triangleleft \mathbf{400}.z!\langle Response(400)\rangle.\mathbf{0} \\
&[\text{else}] \; z \triangleleft \mathbf{500}.z!\langle Response(500)\rangle.\mathbf{0}. \\
DeleteTask = z?(id).&[\text{if } \ldots] \; z \triangleleft \mathbf{204}.z!\langle Response(204)\rangle.\mathbf{0} \\
&[\text{else if } \ldots] \; z \triangleleft \mathbf{404}.z!\langle Response(404)\rangle.\mathbf{0} \\
&[\text{else}] \; z \triangleleft \mathbf{500}.z!\langle Response(500)\rangle.\mathbf{0}.
\end{aligned}
$$

Note how we have used branching to match different paths and request methods. Also, the labels for selections for different situations explicitly tell the status code of the response. A successful GET request returns a list of tasks, $listOfTasks$, and otherwise only status code is returned.

We have now documentation and an implementation for the server side. A client only knows about the documentation and does not have access to implementation details of the server. Below is an attempt to create a client creating a

task.

$$
\begin{aligned}
TaskClient &= (\boldsymbol{\nu} xy)\, ClientBody \\
ClientBody &= a!\langle y\rangle.x \triangleleft \mathbf{POST}.HandlePost \\
HandlePost &= x!\langle description\rangle.z \triangleright \{\mathbf{200} : \ldots, \\
&\qquad\qquad\qquad\qquad\quad \mathbf{400} : \ldots, \\
&\qquad\qquad\qquad\qquad\quad \mathbf{500} : \ldots\}
\end{aligned}
$$

This client is syntactically correct program. However, there are a couple of problems:

1. selection for path '/task' is missing,
2. task description is sent as a string instead of an object containing a field 'description' and
3. in $HandlePost$ status code 200 is matched instead of 201 returned by the server.

A few iterations would be needed to get the implementation correct. Any updates to the server would require more iterations. In addition, there would be no guarantee that the documentation is up to date.

Instead of publishing a documentation, let us assume the server developer publishes a session type. Implementation for the server is the same except the server can always process any request. Therefore, the server never responds with status code 400. Below is a session type for the server.

$$
\begin{aligned}
S_{server} &\coloneqq \&\langle /\mathbf{task} : S_{task}, /\mathbf{other} : S_{other}\rangle \\
S_{task} &\coloneqq \&\langle \mathbf{GET} : S_{GET}, \mathbf{POST} : S_{POST}, \mathbf{DELETE} : S_{DELETE}\rangle \\
S_{GET} &\coloneqq \oplus\langle \mathbf{200} :!\textit{List[Task].end}, \mathbf{500} :!\textit{Resp.end}\rangle \\
S_{POST} &\coloneqq \oplus\langle \mathbf{201} :!\textit{Resp.end}, \mathbf{500} :!\textit{Resp.end}\rangle \\
S_{DELETE} &\coloneqq \oplus\langle \mathbf{204} :!\textit{Resp.end}, \mathbf{404} :!\textit{Resp.end}, \mathbf{500} :!\textit{Resp.end}\rangle \\
S_{other} &\coloneqq \ldots,
\end{aligned}
$$

where *Resp* denotes the type of a response, *List* the type of a list and *Task* the type of a task. Now the type for a client is simply the dual, $\overline{S_{server}}$. Then, any client implementation could be statically typechecked, no test iterations would be needed and a text editor could guide the developer for example by suggesting

labels for selections. Whenever the server is updated, a new session type is published and a client implementation can be typechecked against that.

Session types are much more than this simplified example. However, this demonstrates one use case, how session types can make developing interactions safer, faster and easier.

7 Extensions and further reading

For simplicity we have used monadic π-calculus, that is, each message sent or received contains exactly one name. Thus, any process sending or receiving multiple names needs the same number of outputs or inputs. In polyadic π-calculus [4] a channel can input or output any number of names. For example, $x?(v,u,w).x!\langle v+u+w\rangle.\mathbf{0}$ is a process that receives three values and sends back the sum of them. Assuming the type of the values is *int*, then the type of x is $?(\mathit{int},\mathit{int},\mathit{int}).!\mathit{int}.\mathit{end}$.

Consider a process $P \coloneqq x?(num_1).x?(num_2).x!\langle num_1 + num_2\rangle.\mathbf{0}$, which receives two numbers and sends back the sum of them. What is the type of x? If x can receive and send real numbers, x should accept integers also. However, the type system proposed above would not accept that. The language needs to be extended with subtyping to handle these kinds of cases. Subtyping for session types was first defined by Gay and Hole in [17] and revised in [5]. If the type of a channel is T and T is a subtype of U, then that channel can be used, whenever a channel of type U is expected.

Recall the ATM example in the beginning of the thesis. Whenever a user makes a selection, the session ends after the ATM has processed the request. For example, if a user wishes to first see her balance and then based on the response withdraw some amount of money, she would need to create a new session and send her indentifier again. With recursive types [5], [18], [16] it is possible to describe an ATM, which allows the user to select another option without creating a new session. A session type would be

$$
\begin{aligned}
&S \coloneqq ?\mathit{string}.\mu X.\&\langle \mathbf{dep} : S_{dep}, \mathbf{wd} : S_{wd}, \mathbf{bal} : S_{bal}, \mathbf{quit} : S_{quit}\rangle,\\
&S_{dep} \coloneqq ?\mathit{int}.!\mathit{int}.X,\\
&S_{wd} \coloneqq ?\mathit{int}. \oplus \langle \mathbf{dispense} :!\mathit{int}.X, \mathbf{overdraft} :!\mathit{string}.X\rangle,\\
&S_{bal} \coloneqq !\mathit{int}.X,\\
&S_{quit} \coloneqq \mathit{end}.
\end{aligned}
$$

The differences are that recursion, written as μX, is added and type for terminating process, *end*, is replaced by type variable X. In addition, we have introduced a new option, **quit**, so that the user can end the session.

In a real-life situation the ATM would need to communicate with the bank of the user. Multiparty session types, first proposed by Honda et al. [6], describe

interactions between any number of partners. A global type describes the big picture, while local types describe the communication from the view of the communication partners. In [19] Yoshida and Gheri provide a gentle introduction to multiparty sessions.

Type systems for concurrent processes, including session types, were studied in [8] by Yoshida. In [20] Honda et al. introduce key ideas of session types and illustrate their usage in programming. Dezani-Ciancaglini and de'Liguoro give a comprehensive overview of session types in [14].

8 Conclusion

We introduced basics of session types, a type formalism for protocols, and π-calculus, a process calculus. Our goal was to give a simple, yet comprehensive, introduction to session types for a new reader. We used a simple, clear and easily extensible language containing all the essential primitives for session types. We were detailed in the examples and especially in the proofs, which helps the reader to fully understand the notation and the thinking behind definitions, such as reduction, or typing rules.

The thesis is focusing mainly on theory, which can make it less approachable. Practical examples with diagrams and illustration could make it more approachable and help understanding the big picture better. However, theoretical approach gives better support for understanding the fundamentals and continuing to more advanced reading, in our opinion. Furthermore, we did not discuss equivalences, such as behavioural equivalence, for π-calculus processes. We decided to introduce only the minimum for understanding session types and, therefore, left this topic out.

Understanding session types requires a lot of preliminary knowledge. Especially it is often assumed that the reader is familiar with π-calculus in the literature of session types. We introduced both of these theories and think this makes it easier to start learning session types. As a future work the language could be extended with various ways, such as adding data types or recursion. In addition, discussion for equality of processes could be added.

References

[1] K. Honda, “Types for dyadic interaction,” in *CONCUR*, 1993.

[2] K. Takeuchi, K. Honda, and M. Kubo, “An interaction-based language and its typing system,” in *In PARLE’94, volume 817 of LNCS*, pp. 398–413, Springer-Verlag, 1994.

[3] K. Honda, V. T. Vasconcelos, and M. Kubo, “Language primitives and type discipline for structured communication-based programming,” in *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP ’98, (Berlin, Heidelberg), p. 122–138, Springer-Verlag, 1998.

[4] R. Milner, “The polyadic pi-calculus: a tutorial,” tech. rep., Logic and Algebra of Specification, 1991.

[5] S. J. Gay and M. Hole, “Subtyping for session types in the pi calculus,” *Acta Informatica*, vol. 42, pp. 191–225, 2005.

[6] K. Honda, N. Yoshida, and M. Carbone, “Multiparty asynchronous session types,” in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’08, (New York, NY, USA), p. 273–284, Association for Computing Machinery, 2008.

[7] S. J. Gay, “Bounded polymorphism in session types,” *Math. Struct. Comput. Sci.*, vol. 18, pp. 895–930, 2008.

[8] N. Kobayashi, *Type Systems for Concurrent Programs*, pp. 439–453. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[9] O. Dardha, E. Giachino, and D. Sangiorgi, “Session types revisited,” in *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP ’12, (New York, NY, USA), p. 139–150, Association for Computing Machinery, 2012.

[10] R. Milner, *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 5 ed., 2004.

[11] D. Sangiorgi and D. Walker, *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 1st paperback ed., 2003.

[12] L. Cardelli, “Type systems,” in *The Computer Science and Engineering Handbook* (A. B. Tucker, ed.), pp. 2208–2236, CRC Press, 1997.

[13] J. R. Hindley and J. P. Seldin, *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2 ed., 2008.

[14] M. Dezani-Ciancaglini and U. de’Liguoro, “Sessions and session types: An overview,” in *Web Services and Formal Methods* (C. Laneve and J. Su, eds.), (Berlin, Heidelberg), pp. 1–28, Springer Berlin Heidelberg, 2010.

[15] H. Hüttel, I. Lanese, V. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. Vieira, and G. Zavattaro, “Foundations of session types and behavioural contracts,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–36, 2016.

[16] V. T. Vasconcelos, “Fundamentals of session types,” *Inf. Comput.*, vol. 217, p. 52–70, Aug. 2012.

[17] S. Gay and M. Hole, “Types and subtypes for client-server interactions,” in *Programming Languages and Systems* (S. D. Swierstra, ed.), (Berlin, Heidelberg), pp. 74–90, Springer Berlin Heidelberg, 1999.

[18] O. Dardha, “Recursive session types revisited,” in *BEAT*, 2014.

[19] N. Yoshida and L. Gheri, “A Very Gentle Introduction to Multiparty Session Types,” in *16th International Conference on Distributed Computing and Internet Technology*, vol. 11969 of *LNCS*, pp. 73–93, Springer, 2020.

[20] K. Honda, R. Hu, R. Neykova, T.-C. Chen, R. Demangeon, P.-M. Deniélou, and N. Yoshida, “Structuring Communication with Session Types,” in *Concurrent Objects and Beyond*, vol. 8665 of *LNCS*, pp. 105–127, Springer, 2014.