IMT Institute for Advanced Studies, Lucca

Lucca, Italy

Typing Services

PhD Program in PhD Program in Computer Science and Engineering

XXI Cycle

By

Leonardo Gaetano Mezzina

2009

The dissertation of Leonardo Gaetano Mezzina is approved.

Program Coordinator: Prof. Ugo Montanari, University of Pisa

Supervisor: Dott. Roberto Bruni, University of Pisa

The dissertation of Leonardo Gaetano Mezzina has been reviewed by:

Prof. Matthew Hennessy, Trinity College Dublin

Prof. Vasco Thudichum Vasconcelos, University of Lisboa

IMT Institute for Advanced Studies, Lucca

2009

No young man, no matter how great, can know his destiny. He cannot glimpse his part in the great story that is about to unfold. Like everyone, he must live and learn.

Contents

List of Figures x

Acknowledgements xii

1 Introduction 1
1.1 Service Oriented Scenario 1
1.2 Research issues 2
1.3 Our approach 4
1.4 Main contributions 10
1.5 Outline of this thesis 11
1.5.1 An important remark on mathematical conventions 12
1.6 Origins of the Chapters 13

2 Background 14
2.1 Process calculi 14
2.2 Well behaving processes 17
2.3 Session handling 20

3 Extending the session types framework 22
3.1 Introduction 22
3.2 Session Types 23
3.3 Intersection of Session Types 26
3.3.1 Algebraic meet of session types 26
3.3.2 Inference Relations for the Meet 31
3.3.3 Algorithmic Meet 37
3.4 Types with free type variables 46
3.4.1 Beyond syntactic unifiers 51
3.5 Concluding remarks on the session types framework . . . 58

4 CaSPiS for Session Types **60**
4.1 Introduction . . . 60
4.2 Syntax and operational semantics . . . 62
4.3 Type system . . . 67
4.3.1 A type system for CST . . . 67
4.3.2 Subject reduction and safety . . . 72
4.4 A set based type system . . . 83
4.4.1 Soundness . . . 87
4.5 Type inference or Type checking . . . 103
4.6 A sound and complete syntax directed type system . . . 105
4.7 Concluding remarks on CST . . . 113

5 HVK-X a full session calculus **115**
5.1 Introduction . . . 115
5.2 Syntax and operational semantics . . . 116
5.3 Type system . . . 121
5.3.1 A type system for HVK-X . . . 121
5.3.2 Subject reduction and safety . . . 125
5.4 A syntax directed type system . . . 131
5.4.1 The solve algorithm . . . 140
5.5 Further issues on the completeness of `solve` . . . 148
5.5.1 Solve implementation . . . 152
5.6 Encodings . . . 153
5.6.1 Encoding the π-calculus . . . 154
5.6.2 Encoding CST . . . 157
5.7 Concluding remarks on HVK-X and encoding functions . . 166

6 Progress and comparison with related works **168**
6.1 A more powerful guarantee: The Progress Property . . . 168
6.2 Revisiting our sources of inspiration . . . 179
6.2.1 CaSPiS . . . 179
6.2.2 The Honda-Vasconcelos-Kubo Session Typing System . . . 181
6.2.3 The Gay-Hole Session Typing System . . . 186
6.3 Other related works . . . 190
6.4 Final remarks on CST progress . . . 194

7 Conclusion **195**
7.1 Future work . . . 199

Bibliography **207**

List of Figures

1 Syntax of types 24
2 The syntactic dual of a session types 25
3 The subtyping algorithm 28
4 Algorithmic membership checking for $\overset{c}{\wedge}$ 39
5 Algorithmic membership checking for $_\overset{c}{\wedge}_ = _$ 40
6 The algorithmic meet 42
7 Algorithmic meet at work 46
8 The set of free communicated variables 47
9 An algorithm to extract constraints for the syntactic unifier relative to the subtyping relation 48
10 An algorithm to extract constraints for the syntactic unifier relative to the meet exists relation 52
11 localsolve algorithm 53

12 Syntax of our service calculus 62
13 Definition of free names and free polarized names 64
14 Definition of free process variables 65
15 Structural congruence 65
16 Typing rules: rule (TREC) for recursion requires an additional consistency condition 69
17 Two predicates to check the presence of active actions . . . 70
18 Example of typing 73
19 Example of typing 73
20 Syntax directed typing rules 84
21 Auxiliary functions for sets of types (where $\Diamond \in \{\&, \oplus\}$) . 85
22 The algorithm to extract constraints in Ocaml-like syntax . 96
23 The solving algorithm 98

24 Syntax directed typing rules 107
25 The algorithm to extract constraints in Ocaml-like syntax . 111

26 Syntax of HVK-X 117
27 Definition of free names 119
28 Definition of free polarized session names 119
29 Definition of free process names 120
30 Structural congruence 120
31 Typing rules 123
32 Linear access function 133
33 Syntax directed typing rules 133
34 same function 134
35 The algorithm to extract constraints in Ocaml-like syntax . 138
36 An implementation of the solve algorithm 152
37 Simply typed π-calculus 154
38 Encoding of the π-calculus processes 154
39 Encoding of the π-calculus types 154
40 Encoding CST in HVK-X 159

41 The transitive closure of the session relation 177
42 Syntax of SL 182
43 The structural congruence of SL 183
44 The operational semantics of SL 183
45 Syntax of session types for SL 184
46 The SL type system 185
47 Syntax of πST 187
48 Syntax of πST types 187
49 Structural congruence of πST 188
50 Operational semantics of πST 188
51 The type system of πST 189

Acknowledgements

The author would like to acknowledge:

1. Roberto Bruni for many things, for being my supervisor, for supporting me, for standing me for these three years but the most important one for teaching me formal methods.
2. My brother Claudio who helped and entertained me during the thesis writing period.
3. Matthew Hennessy, Lucia Acciai, Mariangiola Dezani-Ciancaglini, Rocco De Nicola and Vasco Thudichum Vasconcelos for their help and useful suggestions.
4. Lisa simply for being Lisa.

Abstract

The notion of a session is fundamental in service-oriented applications, as it serves to separate interactions between clients and different instances of the same service, and to group together logical units of work. In the area of process calculi Honda, Kubo and Vasconcelos proposed their perspective of what a session should be from the perspective of theorical foundations. They presented a calculus equipped with a notion of session types that govern the interactions between peers. This first proposal gave rise to a new research direction and to a community of researchers interested in session types and their extensions and applications. The great merit of session types is in fact to be like a classical type system, intended to describe structural properties of the data manipulated by programs. One can think of a session type as the equivalent notion of channel sorting for the π-calculus. The novelty is that well-typedness of a process implies a stronger property than any other classical type systems, namely the session safety. Session safety guarantees that at runtime any interaction inside a session will proceed without errors due to mismatching communications. Moreover, with a little additional effort, session safety implies the progress property, which in some manner prevents deadlock. Well typing of a process written in a session calculus can be easily verified at the cost to annotate certain names of the processes with session types. Here we address the problem of finding efficient procedure for checking well-typedness in absence of any type annotation or said in other words the type inference of session types. It is interesting how different notions proposed in different works on session types are used together as tools to achieve the result. At the end our study leads to establish a formal theory of session types that can be applied and transferred to various settings and formalisms. Since type inference strictly depends on a specific calculus we show the wide applicability of our result studying the

problem for two particular calculi with very different mechanisms of session instantiation. Prototype implementation of the type algorithms are written in Ocaml and available at `http://www.di.unipi.it/~mezzina`.

Chapter 1

Introduction

1.1 Service Oriented Scenario

Recent years have witnessed a tremendous growth of the so-called Service Oriented Architectures. The main idea is to have loosely coupled components, called services, that can be described, published in publicly available registries, searched and dynamically assembled to form larger applications. A major scientific research trend has been aimed to extend the description of services to include behavioral information, like the communication protocol they use, that can be used to infer some guarantees of their composition.

Without entering into the technicalities of software engineering, we consider a service as a piece of software capable to carry out a particular function. It is similar to a classical server in a client-server architecture, but the main novelty is that services are globally available over networks and anyone can use them and their functionalities to build a more complex software. We call this scenario service oriented computing since all the software functionalities are delegated to services. Differently from the old client service architecture in which one has to program directly at the socket level to develop a particular client for a certain server, here every possible service can be accessed in the same uniform manner irrespectively of its location. From the programmers point of view this means that he/she can (ideally) develop an application visually by simply connecting basic compatible services to build the expected collaborative system. The first incarnation of this scenario was the Web Service

architecture which allowed users to interact with every web service in a uniform way so to have interoperable and cross-platform services. Today Web Services have not reached the widespread use they promised, and their potentialities are still not fully exploited. Nevertheless we find interesting how services can be accessed in an uniform and cross-platform manner by exploiting only three standards, namely SOAP, WSDL and XML Schema, viewing the service offering a simple remote procedure call. WSDL offers the possibility to describe a contract which is intended to describe how to prepare a SOAP message in order to invoke the service and receive a reply back. WSDL describes some sort of a basic contract since it embodies the service demands, i.e. how the service wishes to be invoked, syntactically. Each message exchanged with a web service is an XML packet with the invocation parameters. WSDL has a specific section for this purpose, which usually is a XML-Schema fragment. An XML-Schema describes the type of an XML fragment and such a fragment has to be validated against a certain Schema. Finally, SOAP describes how to encapsulate in a standard manner an XML file into another XML one.

However the needs to build real applications are many and very demanding ones, so numerous additions and standards came out. Some of these extensions are concerned with security, others are concerned with failures, others with cataloguing and discovery for available services. The presence of many standards that may lead to cumbersome and difficult to handle specifications and the lack of a mathematical foundations may undermine the success of Web Service in a near future. Yet the development of these new standards comes from pragmatical needs and it is a big challenge for computer scientists that studied for decades interaction models. The fact that the service architecture is reviving under a new shape, called REST architecture, means that services are very important also in practice for the development of future applications, which justifies also the growth of the research in this field.

1.2 Research issues

We address the problem of developing a solid theory of interaction for service oriented applications by means of process calculi and type systems. It is natural to place our analysis in the service oriented scenario because we assume that each entity is invoked in a uniform manner, without entering in the detail of how such invocation is realized. Moreover we go a step beyond with respect to the simple request/response method

invocation, which allows only to represent a series of communications as a collection of distinct, unrelated interactions, where for example if we need to send a value, get a reply back and then send another value computed on the basis of the first reply, we need at least two different services. Our approach builds on the well-established theory of process calculi. To logically group single communications into larger protocols, we use a basic structuring concept, called a *session*. A session is a chain of dyadic interactions between two parties, the client and the service, that carry out a conversation distinguished by the fact that we use a private channel (either implicit or explicit), through which the interactions belonging to the same session are performed and separate from other communications. The possibilities of interactions we consider within a session are the same as proposed initially in (37) with three basic communication primitives, value passing, label branching and delegation. Value passing is the standard synchronous message passing. Label branching is a form of method invocation without value passing: a set of labels is offered with different continuations and the possibility of choice, that is, we have an internal and an external choice. Delegation is for passing an opened session to another process, which allows substituting a certain partner in a conversation. These three primitives can be used to model a great number of interesting scenarios at the right level of abstraction and, more importantly, there exists a typing discipline for them very close to the concept of types in classical programming language, citing (63): "The methodological perspective that types are best used to describe structural properties of the data manipulated by programs, as well as the more pragmatic requirement that well-typedness must be automatically and efficiently verifiable".

Well-typedness holds if the type annotations are consistent with the process specification and it is important that well-typedness has to be preserved during the evaluation of processes. As an example consider the following description of a generic RPC service

- open a session k of type T relative to an invocation of a
- receive $x : int$ from the session k
- compute $\mathsf{f}(x)$ of type int and send the result back on the session k
- end

Service a is well-typed if T, the type of k, is $?(int).!(int)$, i.e. an integer is received $?(int)$ and then an integer is sent $!(int)$. The informal description

of service a can be rendered more compactly as $a(k).k?(x : int).k!(\mathsf{f}(x))$. A "compatible" client that wishes to consume the service must have some sort of dual type to T, like $!(int).?(int)$ (in which we switched input/output actions). Well-typedness in this setting directly implies *safety of the communication*, in the sense that the client always undertakes the action expected by the service. However more powerful guarantees can be enforced: *the progress property* which checks that the client does not deadlock in a opened session and the *deadlock freedom* property which checks that deadlock does not happen between both client and service.

The notion of type compatibility is also a topic of interest. Recently Gay and Hole studied a subtyping relation which is effectively computable by means of an algorithm inspired to the classical subtyping algorithm among infinite regular types (2; 31; 50). This algorithm is an important achievement, because it allows us to model infinite interactions and we have a terminating sound and complete algorithm to check subtyping of infinite session types.

1.3 Our approach

The topic which we shall address in this thesis is how to relief programmers from the burden of type annotations possibly providing a lightweight type engine. The answer is not obvious due to the typing rules for internal/external choices, recursion, session delegation and linearity.

- Internal and external choices introduce non-determinism since the type of an internal choice may have more options than the one actually selected while the type of an external choice can have less options than the ones actually offered. However, choices relax the exact matching in the sense that it is possible to define different replicas of a service with different behaviors, and different clients for the same service.
- Recursion requires the comparison among cyclic behaviors which are also mixed with subtyping and choices.
- Session delegation is not obvious in presence of subtyping since when we send a session to someone we have some expectation on the actions to be performed in order to complete the delegated session, but we do not know how he/she is going to actually use the session.

- Type systems for session types are mixed in the sense that they encompass both classical type systems and linear type systems (64). Here linearity refers to the fact that at any time only two peers can have the same session, because sessions are dyadic.

Let us introduces these concepts by means of examples described in the syntax of HVK-X (discussed in Chapter 5).

Internal/external choices. A database service is offered by means of an accept on the shared name db. After the client request is issued a new private session k, shared between the server and the client is created. The new session ensures the client and the service have a private conversation when communicating on k. Service db offers four options to choose from: `query`, `update`, `add` and `end`. The client, depending on the outcome of expression $test$ (representing its internal state) is interested in either `query` or `update`. If the client requests the `query` option, then the service expects a SQL query which will be bound in the string variable $sqlvar$ in the body process P_1.

$$
\begin{array}{l}
db(k).k?(\texttt{query})k?(sqlvar).P_1 \\
\qquad\qquad\qquad\qquad +k?(\texttt{update}).P_2 \\
\qquad\qquad\qquad\qquad +k?(\texttt{add}).P_3 \\
\qquad\qquad\qquad\qquad +k?(\texttt{end}).\mathbf{0} \\
\overline{db}(k).\texttt{if}\ \ test\ \ then\ \ k!(\texttt{query}).k!(sqlstr).P \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ \ k!(\texttt{update}).Q
\end{array}
$$

Suppose that k in each P_i has type T_i, k in P has type $\overline{T_1}$ and k in Q has type $\overline{T_2}$ (for $\overline{T_i}$ denoting the dual of T_i). If we use $\&\{l_1 : V_1, \ldots, l_n : V_n\}$ and $\oplus\{l_1 : V_1, \ldots, l_n : V_n\}$ for external and internal choices respectively with options $l_1, \ldots, l_n$ then the process is well typed when k has type, say, $U_{db} = \&\{\texttt{query}{:}?(string).T_1, \texttt{update}{:}T_2, \texttt{add}{:}T_3\}$ on service side and `k` has type $\oplus\{\texttt{query}{:}!(string).\overline{T_1}, \texttt{update}{:}\overline{T_2}, \texttt{add}{:}\overline{T_3}\}$ on client side, because these two types are syntactically equivalent if we exchange $\&$ with $\oplus$ and $?$ (the symbol for input) with $!$ (the symbol for output). We require in fact that both the rules for service request and service accept need to agree with the standard assumption relative to db which is $[U_{db}]$, a service with a session of type U_{db}. Notice however that the external choice removed `end` from the list of labels and that the internal choice locally guessed the type of the external choice provided by the service; i.e., the typing rule of a process of the form $k!(l).P$ must guess a set of choices offered by the counterpart. In turn, this will also force the two

branches of the conditional statement to take into account the same set of alternatives since in order to guarantee the subject reduction, the type for an if-then-else statement must be the same for both branches. To guide the intuitions, the following is a fragment of the proof-tree relative to the process within the `query` label. Type judgments here take the simple form of $\Gamma \vdash P : \Delta$ where P is a process, Γ is the standard environment and Δ is the linear environment.

$$
\dfrac{\dfrac{\dfrac{\overset{(1)}{\Gamma, sqlvar : string \vdash \mathbf{0} \triangleright k : \mathtt{end}}}{\vdots}\ (2)}{\Gamma, sqlvar : string \vdash P_1 \triangleright k : T_1, \Delta}}{\Gamma \vdash k?(sqlvar).P_1 \triangleright k :?(string).T_1, \Delta}\ (3)
$$

Notice how k in the axiom (1) is guessed, how the behavior $?(string).T_1$ in correspondence of an input (3) is built and how the assumption on $sqlvar$ is propagated towards the axiom (2), (1). The solution we propose is to generate a set of constraints whose satisfiability implies typability of the process. For instance, relatively to the above process we generate the following set of constraints where all α's are automatically generated type variables:

$$
\alpha_{db} \leq \&\{\mathtt{query} :?(string).T_1, \mathtt{update} : T_2, \mathtt{add} : T_3, \mathtt{end} : \mathtt{end}\}
$$
$$
\alpha_{if} \leq \oplus\{\mathtt{query} :!(string).\overline{T_1}\} \quad \alpha_{if} \leq \oplus\{\mathtt{update} : \overline{T_1}\} \quad \overline{\alpha_{if}} \leq \alpha_{db}
$$

The first constraint on α_{db} says that the type of the session relative to the service `db` must be less than the type of the internal choice considering all the available branches. In fact the type U_{db} which does not include the branch labeled with `end` satisfies the constraints. A type which offers an external choice is a subtype of an another type that offers a greater range of possibilities to select from, regardless of the order in which the labels are offered. Relatively to the "if" instruction of the client, we generate two constraints, one for each branch of the if-then-else, and α_{if} must satisfy both at the same time. The last inequality is relative to the invocation, since the body of the invocation is the if-then-else, we constrain the dual of α_{if} to be less than the type of the session relative to the service `db`. Here we use duality since we are on the client side. Finding a solution to α_{if} implies having a way to compute a minorant of the two types and since we are searching for the most general minorant we need

a way to compute the intersection of two session types. This case is simple the intersection of $\oplus\{\texttt{query} :!(string).\overline{T_1}\}$ and $\oplus\{\texttt{update} : \overline{T_2}\}$ is the type $\oplus\{\texttt{query} :!(string).\overline{T_1}, \texttt{update} : \overline{T_2}\}$ and then making a further step with the transitivity of $\leq$ we can conclude $\&\{\texttt{query} :!(string).T_1, \texttt{query} : !(string).T_1\} \leq \&\{\texttt{query} :?(string).T_1, \texttt{update} : T_2, \texttt{add} : T_3, \texttt{end} : \texttt{end}\}$ which holds implying the typability of the process.

Recursive processes The type system uses a third environment to store assumptions relative to the process variables, the process environment. The type of assumptions in the process environment strictly depends on the recursion construct offered, but at least it needs to store the linear environment assumed for each process variable. For instance consider the following client for a hypothetical service *dbrec*

$$\overline{dbrec}(k).\texttt{rec}\ X\ \texttt{if}\ test\ \texttt{then}\ k!(\texttt{query}).k!(sqlstr).X\ \texttt{else}\ k!(\texttt{query}).X$$

which uses the recursion operator to repeatedly choose of either query or update We have two ways to generate a set of constraints for a process involving recursion. The first is to account for the recursion *during* the computation of the set of constraints and then mime the recursion in the type by means of the μ operator:

$$\alpha_{dbrec} \leq \mu\alpha_X. \oplus \{\texttt{query} :!(string).\alpha_X\} \quad \alpha_{dbrec} \leq \mu\alpha_X. \oplus \{\texttt{update} : \alpha_X\}$$

The second is to leave the solution of the recursion be in the set of constrains as well:

$$\alpha_{dbrec} \leq \alpha_X \quad \alpha_{if} \leq \oplus\{\texttt{query} :!(string).\alpha_X\} \quad \alpha_{if} \leq \oplus\{\texttt{update} : \alpha_X\} \\ \alpha_X \leq \alpha_{if}$$

In the first case, the solution is at hand simply by computing the intersection which is equal to $\oplus\{\texttt{query} : \mu\alpha_X. \oplus \{\texttt{query} :!(string).\alpha_X\}, \texttt{update} : \mu\alpha_X. \oplus \{\texttt{update} : \alpha_X\}$, i.e. initially one can choose between query and update then after the first choice one can only choose of either query or update an unbounded number of times.

The solution to the second set of constraints is not so immediate since we have a cyclic dependency between α_{if} and α_{rec}. We formally prove that such dependencies can be solved by means of recursion hence:

$$\alpha_{dbrec} \leq \alpha_X \quad \alpha_X \leq \mu\alpha_X. \oplus \{\texttt{query} :!(string).\alpha_X\} \\ \alpha_X \leq \mu\alpha_X. \oplus \{\texttt{update} : \alpha_X\}$$

and then one can conclude in a similar manner as above computing the intersection.

An important condition is to consider recursion instead of replication that easily corrupt linearity. Consider for example a replicated output $*k!(sqlstr)$ with the expected operational semantics, then after two unfolding steps we have $k!(sqlstr)|k!(sqlstr)| * k!(sqlstr)$ which obviously violates the linearity of session k. Recursion instead allows for a much more careful use of *iteration inside* protocols.

Delegation In order to describe how delegation works we model inside P_1 the interaction with the query manager completely delegating to it the task of replying the client, in order for example, to be ready for another client request.

$dbdel(k).k?(\texttt{query}).k?(sqlvar).\overline{querymanager}(k1).k1!\langle\langle k\rangle\rangle.\mathbf{0}$ $|$
$querymanager(k1).k2?((k)).P_1$

Here we use $k1$ as the session to talk with the service $querymanager$ and the symbol $|$ indicates the execution of both services at the same time, i.e. their parallel composition. We send k through $k1$ using $k1!\langle\langle k\rangle\rangle.\mathbf{0}$ and we receive k through $k2$ using $k2?((k)).P_1$. Delegation allows starting a conversation directly with the client that issued a request on $dbdel$ to the query manager. The set of constraints relative to $dbdel$ contains also the following one

$$\alpha_{dbdel} \leq \oplus\{\texttt{query} :?(string).\alpha_{del}\}$$

where α_{del} is unknown. Since we cannot solve directly such constraint, we postpone the solution until α_{del} is solved by some substitution as in this case:

$$?(\alpha_{del}) \leq ?(\alpha_{P_1}) \quad \alpha_{P_1} \leq T_1$$

From the first constraint we get the substitution $\alpha_{del} = \alpha_{P_1}$ and then we can substitute α_{del} with T_1 in order to obtain the closed constraint $\alpha_{dbdel} \leq \oplus\{\texttt{query} :?(string).T_1\}$. If instead the specification of the service $querymanager$ is missed we must find a way to define a general substitution for α_{del} which does not compromise the typability of the entire process, that is we must be careful not to instantiate α_{del} too much. The point is that both delegation and recursion can generate open constraints.

Linearity Type systems for session types are mixed in the sense that they encompass both classical type systems and linear type systems (64). The classical realm allows collecting the type of services and the linear realm allows to collect the type of sessions. A linear type system for the π-calculus was first studied in (47).

Usually the differences between classical and linear type systems can be seen in both the axioms and the typing rule for parallel composition. The axioms of classical type systems may contain any assumption in addition to the needed one. The possibility of having arbitrary assumptions allows the proof of both the Weakening and the Strengthening Lemma. Axioms in linear type systems contain only needed assumptions and usually neither the Weakening Lemma nor Strengthening lemma do hold. The rule for parallel composition is also different in standard and linear type systems. In standard type systems the environment used to type each parallel process is the same, in linear type systems one has to develop an environment composition function that allows to linearly distributing the assumptions. However the linear realm of a session type system have a peculiarity, it allows any assumption in the axiom for the nil process as long as it refers to ended sessions. Ended sessions are sessions that do not perform any action and have a special type `end`. Type `end` is also used as a trailing (like the nil process) for every finite session types. Infinite session types are trailed with a type variable which allows recursion. This way to type the nil process allows to arbitrarily add ended assumptions used as trailing for each open session appearing in a process. The superfluous ended assumptions that create linearity conflict will be ruled out by the linear environment composition function. This fact is possible since session types are collected by starting from axioms and moving towards the leaves of the typing proof tree. Vice versa assumptions in the standard environment are collected from the leaves by moving towards the axioms. Consistency of linear assumptions and standard assumptions is checked in the typing rules for service invocation and service definition. In fact in the premise of these rules the session type relative to the service is collected and in the conclusion the standard type assumption relative to the service is collected. Furthermore in the rule for service invocation the relative session type must be dual with respect to the type assumed for the service and this is important. Session types are defined in such a manner that each action has a dual counterpart: input with output and internal choice with external choice. The duality check is the guarantee that communication within a session evolves without errors.

For a detailed review of the main literature available on session typing systems we refer to Sections 6.2.2 and 6.2.3 which can be used as an useful introductory material for those readers not familiar with session types.

As one could have guessed there is a number of interesting problems to solve in order to infer the typability of a process.

1.4 Main contributions

Our main contribution is the development of an algorithm to infer the typability of sessions in a process solving all the sources of non-determinism already outlined. We first study CST as an higher level language than the calculus of Honda et al. since it allows the implicit and disciplinated managing of the session variables.

At this point we propose a type system for a variant of the language of Honda et al. with general recursion in place of process definition and name extrusion, which we call HVK-X with the aim of studying typability of a process. The set of constraints whose solution implies the typability, involves cyclic dependencies due to the recursion, non direct dependencies due to the service name extrusion and free variables due to the delegation. Service name extrusion allows the communication of a service name within a session to a partner. The partner can either invoke such service or define a new replica of the service dynamically.

We propose an automatic way to solve the set of constraints, an algorithm that multiplexes each constraint in other constraints directly depending on it. We also implement this algorithm in a tool called TypSes written in Ocaml (55).

We provide an encoding function from CST to HVK-X in order that each well typed CST process is a well type encoded HVK-X process and vice versa. From this encoding it comes out that one can use TypSes to check the typability of CST processes.

Said in other words we solve for session types a problem that is classically solved in the simply typed π-calculus finding the sort of each channel. While in the simply typed π-calculus, one finds out the type of values exchanged on each channel here we find out the session type associated to each service. However, since we are in presence of subtyping we do not return the type of a service but rather a series of constraints that the type of each service must satisfy.

The good news about this fact is that each type system built on top of

the simply typed π-calculus (25; 41; 45; 48; 49) can be used with session types. These type systems collecting the so called channel usages allow checking a lot of useful properties such as termination, deadlock freedom, resource analysis and correspondence assertion and using a session type system increases the expressivity reducing the total number of channels (since each channel can be used to exchange more values) and the consequent cost of verifying the desiderated property.

From another point of view one can think of this work as a useful introduction to the topic of dyadic sessions with session types and as a useful index of properties of the subtyping relation.

1.5 Outline of this thesis

The thesis regards: **(1)** the consolidation of session type theory, Chapter 3, **(2)** the application of the theory to calculi with different session handling principles, Chapters 4, 5 and 6. More precisely, this thesis in organized as follows:

- In Chapter 2 we introduce an informal background on process calculi with focus on the π-calculus. We also introduce some notions of type systems and the simply typed π-calculus. Incrementally we give the main ideas underlying the π-calculus with sessions and session types, then we specialize the session instantiation mechanism to obtain the session calculus of Honda, Kubo and Vasconcelos. Finally we further specialize the session handling mechanism on which CaSPiS is based.

- In Chapter 3 we present the session types framework. We present an algorithmic way to compute the intersection between two session types and we prove it to be sound and complete (Theorem 3.19) with respect to the algebraic notion of intersection as induced by the subtyping relation between two session types. Subsequently in order to deal with subtyping relation containing type variables we use the notion of syntactic unifier and then we prove in Proposition 3.30 that under suitable assumptions the syntactic unifier of the subtyping relation behaves as the most general unifier.

- In Chapter 4 we introduce CaSPiS for Session Types, or CST for short and a relative type discipline which enjoys Subject Reduction (Theorem 4.17). From this type system we extract a syntax directed

type system which uses sets to store the types of multiple branches and allows inferring the type of the recursion on the fly. Theorem 4.30 shows the soundness of the set-based type system with respect to the original type system. Our `solve` algorithm solves a certain set of constraints and it succeeds if and only if a process is well typed in the set syntax directed type system (Theorem 4.34).

- In Chapter 5 we introduce HVK-X, a calculus stemmed from the session calculus of Honda, Kubo and Vasconcelos and its type discipline with the proof of the typing preservation along reductions (Theorem 5.12). The relative syntax directed type system is sound (Proposition 5.15) and complete (Proposition 5.16) and the `solve` algorithm succeeds if and only if a process without open constraints is typable in the syntax directed type system (Theorem 5.27). We also encode CST in HVK-X in such a manner that each well typed process is a well typed encoded process and vice versa (Theorem 5.37). In the same manner we encode (the simply typed) π-calculus in HVK-X. This fact allows having only one implementation of the typing algorithms for the simply typed π-calculus, CST and HVK-X.

- In Chapter 6 we prove the progress property for CST (Theorem 6.10. We compare in detail our work with the proposal in (9; 33; 37) which are our main sources of inspiration and we discuss related works.

- In Appendix we report some examples that step by step highlight all the functionalities of TypSes.

We prefer to distribute all technical background among the various chapters, so to keep each notion and notation closer to the place where it is needed.

1.5.1 An important remark on mathematical conventions

We shall introduce each needed syntactical convention each time we need, but one such (non-obvious) convention is used extensively in proofs throughout the thesis. This is a special dot notation often used in the proofs for symbol economy. More precisely, in order to reuse some symbol s that appears in the statement of a theorem we write $\dot{s}$ in the proof to refer a different symbol (like those arising from inference rules)

and sym to explicitly refer the symbol appearing in the statement when we need to do so in order to avoid ambiguities and renaming of symbols in inference rules. An example of this is, the proof of Theorem 5.12. In general we push for the economy of symbols so when there are no risks of clashes we tend to overload the same symbol to indicate different things, for example we use always symbols Γ and Θ for typing environments regardless of the calculus: each time it will be the shape of the typing sequents to disambiguate the type of assumptions contained.

1.6 Origins of the Chapters

The results in this thesis are concerned with both the development and the formal study of a tool, called TypSes see `http://www.di.unipi.it/~mezzina`, for checking the typability of a process written in a session calculus. Here the list of pointers of published and submitted paper relative to each chapter:

- The first part of Chapter 3 is adapted from the paper "Meet recursive session types" (submitted to a conference during the writing of this thesis).
- Chapter 4 is an enhancement of the work "How to infer finite session types in a calculus of services and sessions" published in the Proceedings of Coordination Models and Languages (56) and of the work "How to infer session types in a calculus of services and sessions" (currently submitted to a conference during the writing of this thesis).
- Chapter 5 is the result of the development of TypSes.
- Progress for CST reported in Section 6.1 is an enhancement of the work "Types and Deadlock Freedom in a Calculus of Services, Sessions and Pipelines" published in the Proceedings of Algebraic Methodology and Software Technology (15).

All the rest of the contents is original to this thesis.

Chapter 2

Background

2.1 Process calculi

Process algebras or process calculi are mathematically rigorous languages with well defined semantics that permit describing and verifying properties of concurrent communicating systems. They can be seen as mathematical models of processes, regarded as agents that interact continuously with other similar agents and with their common environment. The agents may be real-world objects (even people), or they may be artifacts, embodied perhaps in computer hardware or software systems. Modeling a system by means of a process algebra allows to focus on the aspects of interest such as secure communication, synchronous and asynchronous communication, distribution awareness and resource access. Process algebras provide a number of constructors for system descriptions and are equipped with an operational semantics that describes systems evolution.

There has been a huge amount of research work on process algebras carried out during the last 25 years that started with the introduction of CCS (57; 58) and CSP (13) and maturated with the introduction of the π-calculus (61), expressive enough to capture most of the others proposed models. The capacity to change the connectivity of a network of processes is the crucial differences between the π-calculus and the preceding proposals. This changing of connectivity allows describing *mobility* among processes since with a suitable level of abstraction the location of a process is determined by the links which it has to other processes.

According to this way of thinking, the movement of a process can be represented entirely by the movement of its links. Of course there are many other ways to describe mobility but this choice is economical, flexible and moderately simple. This flavor of mobility is now known as *name mobility*.

The main ingredients of a specific process algebra are:

1. A minimal set of well thought operators capturing the relevant aspects of systems behavior and the way systems are composed.

2. A transition system associated with the algebra via structural operational semantics to describe the evolution of all systems that can be built from the operators.

3. An equivalence notion that permits abstracting from irrelevant details of systems descriptions.

Let us now introduce some notions of the π-calculus. The simpler π-calculus process is $\mathbf{0}$ used for an inactive process, $x\langle\tilde{y}\rangle.P$ (sometimes written also as $x!(\tilde{y})$ or $\overline{x}y$) for the output of a possibly empty tuple $\tilde{y} = y_1, \ldots, y_n$ on the channel x followed by the process P and $x(\tilde{y}).P$ (sometimes written also as $x?(\tilde{y})$) for the input of a possibly empty pairwise distinct tuple $\tilde{y} = y_1, \ldots, y_n$ on the channel x followed by the process P. These basic actions are called prefixes, additionally there are some operators to compose processes such as the parallel composition of $P|Q$ and the choice $P+Q$. The former allows to describe the parallel execution of both P and Q while the latter describes the execution of either P or Q. Additional expressive power is achieved by means of the restriction operator $(\nu x)P$ which allows to hide all the actions relative to x within P so to describe communication restricted to a certain numbers of participants. New names $\tilde{x}$ and x introduced by $y(\tilde{x}).P$ and $(\nu x)P$ are called bound and the respective operator are called binders.

Bound names are allowed to be alpha-renamed and two processes are considered equivalent up-to renaming of the bound names, e.g. $(\nu x)x\langle y\rangle$ and $(\nu z)z\langle y\rangle$ are the same processes. Moreover one can consider two processes equal up-to the so-called structural congruence (pointed with $\equiv$), an equivalence relation preserved by all operators of the process calculus. A typical example are the monoidal laws of the parallel composition operator $P|Q \equiv Q|P$ (commutativity), $P|\mathbf{0} \equiv P$ ($\mathbf{0}$ identity element) and $(P|Q)|R \equiv P|(Q|R)$ (associativity law), then for example $x().(P|\mathbf{0})$ and $x().P$ are considered as the same process. With the help of structural

equivalence it is also possible to describe an unbounded availability of a certain name provided by means of: replication, recursion and process definition. Recursion is written as $\texttt{rec}\ X.P$ where X is a process variable, the respective structural congruence rule is $\texttt{rec}\ X.P \equiv P[^{\texttt{rec}\ X.P}/_X]$. Replication $!P$, sometimes written $*P$, is defined by means of recursion letting $!P = \texttt{rec}\ X.P|X$. Process definitions written as $X(\tilde{x}) \stackrel{\text{def}}{=} P$ with some $X(\tilde{y})$ possibly free in P resembles function calls with the list of actual parameters $\tilde{y}$ and formal parameters $\tilde{x}$. A standard result is that process definition can be encoded by means of only replication (for instance see (60) Section 9.5). In this thesis we prefer using recursion which gives a fine grained control in modeling both recursive protocols and service availability.

Operational semantics of the π-calculus is given by means of either a Labeled Transition System (LTS for short) or by reductions. The former describes the evolution of a process using a relation of the form $P \xrightarrow{\lambda} Q$ where P evolves in Q with the label λ. For example the axiom for output prefix is $x\langle\tilde{y}\rangle.P \xrightarrow{\overline{x}\tilde{y}} P$, the axiom for input prefix is $x(\tilde{z}).P \xrightarrow{x\tilde{y}} P[^{\tilde{y}}/_{\tilde{z}}]$ which given in the early style guesses the received tuples $\tilde{y}$ where $[^{\tilde{y}}/_{\tilde{z}}]$ is the standard capture avoiding substitution of the tuple $\tilde{y}$ for the tuple $\tilde{z}$. The communication rule is $\dfrac{P \xrightarrow{x\tilde{y}} P' \quad Q \xrightarrow{\overline{x}\tilde{y}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$ (COM) where τ represents a silent action due to the synchronization of the two processes.

Giving the operational semantics by means of reductions allows describing the evolution of a process specifying how redexes are evaluated after being grouped thanks to the structural congruence. For example the corresponding communication rule given by reductions is $x(\tilde{z}).P + M \mid x\langle\tilde{y}\rangle.Q + N \rightarrow P[^{\tilde{y}}/_{\tilde{z}}]|Q$ where M and N are arbitrary summations and the label τ disappears. When we give the operation semantics by reductions we are implicitly considering only reduction actions (indicated with τ).

An interesting fact about the π-calculus is that it allows to extend at runtime the scope of a name thanks to the name extrusion. Consider e.g. the process $((\nu y)x\langle y\rangle.P)|x(z).Q$ in which y is private to P. After a communication on x, it evolves to $(\nu y)(P|Q[^y/_z])$ in which both P and Q know y. Name extrusion, for example, allows to describe how the connections in a system evolve dynamically, given that, P is connected with Q if both P and Q know a certain name. There is another vari-

ant of the π-calculus called πI (67) which limits the communication to private names only. Syntactically it restricts outputs to processes of the form $(\nu\tilde{y})(x\langle\tilde{y}\rangle.P)$ written $\overline{x}(\tilde{y}).P$ with the following communication rule $\overline{x}(\tilde{y}).P+M|x(\tilde{y}).Q+N \rightarrow (P|Q)$. The different notation for output makes explicit that it constitutes also a binder, like the restriction and the input prefix. Notice the only form of substitution is the alpha-renaming ahead of communication to make $\tilde{y}$ syntactically match.

2.2 Well behaving processes

One benefit on using process calculi is to prototype large systems in a natural and formal way to study their interaction properties. Formal verification of concurrent systems within the process algebraic approach is performed either by resorting to behavioral equivalences like trace equivalence, bisimilarities and testing equivalence (10; 61; 68) (just to cite few) for proving conformance of processes to specifications that are expressed within the notation of the same algebra or by checking that processes enjoy properties described by logic formulae (18; 19; 36).

Thus for example we have a famous equality $x().y() + y().x() \sim x()|y()$ where $\sim$ is the bisimilarity, that is the parallel execution of x and y corresponds to either executing first the sequence of x and y or executing the sequence of y and x.

Another trend relative to the verification, which we embrace here, is the development of type systems. In a context where type systems are studied it is simpler to consider the operational semantics given by means of reduction contexts by the fact that half-splitted actions (as the rule for output given above) are not present.

Type systems allow, by means of a set of inference rules, to decide if a process is well-typed or not. The good thing about well-typed processes is that each of them automatically satisfies a certain property and when well-typing is proved to be preserved along reductions, then the property of interest is preserved during the entire lifetime of a process. We are interested in type systems for which the well-typing property is decidable. Well-typedness can be checked with the help of the types annotated by the user, in which case the type system checks that the provided types allow building a correct proof tree using the inference rules of the type system. This is called *type checking*. Also well-typedness can be checked without providing any type annotations, and by developing an algorithm that is able to automatically discover a proof tree allow-

ing the input process to be well-typed using the set of inference rules. This is called *type inference*. In this case the challenge is developing type inference systems, where proof trees can capture the most general type (sometimes called *principal type*), so to cover all possible correct instances of the application.

We now start introducing all these concepts by means of general examples. Consider the following π-calculus process:

$$P \ = !ser(r).(r\langle\rangle \mid r\langle\rangle) \mid (\nu r')ser\langle r'\rangle.r'()$$

Here ! indicates the replication, the fact that ser is always available for input, that is the server waits an unbounded number of requests by the clients. Intuitively P models the behavior of a ping server that reads a channel r and send two replies back to signal it is still correctly working. The client creates by means of the restriction operator a new name r' that it subsequently uses to wait only one of the two replies. In order to typecheck the previous process we can annotate each bound name with type information:

$$P \ = !ser(r:[]).(r\langle\rangle \mid r\langle\rangle)|(\nu r':[])ser\langle r'\rangle.r'()$$

in which we indicate that both r and r' have the same type; a channel used only to signal, i.e. [] is the type of channel used to send a null value. Consider that in order to type P we also need an assumption on the type of ser since it is not introduced in P by any binders and thus is a free name. P is well typed in the simply typed π-calculus assuming ser of type [[]] a channel used to send a channel name in turn used to send a null tuple. A type system which makes assumptions only on free names is given á la Curry. In this thesis every type system is presented à la Curry but also one can assume that each name has a given type a priori in this case the type system is said to be à la Church.

Typing inference for the simply typed π-calculus is decidable (70) i.e., there exists an algorithm to discover the most general type of a channel. As a direct consequence of the well typing of a process and the corresponding subject reduction we have the disciplined usage of channels since they are used to send and receive only one type of value. On the top of this type system a series of type systems have been built, like (41; 45; 49), which allow to prove stronger property such as the deadlock freedom of a process. We show how one of such type system works coding P in Typical (43):

```
*ser?r.(r!()|r!())  | (new r1 in ser!r1.r1?x)
```

Running the tool we obtain the following output:

```
TyPiCal 1.6.2: A Type-based static analyzer for
   the Pi-Calculus analyzing ping...
*ser?r.(r!()|r!())  | (new r1 in ser!!r1.r1??x)
Elapsed Time: 0.001999sec
```

In particular the tool signaled with `ser!!r1.r1??x` (the doubled action) that both the operations of invocation of `ser` and the operation of reading the result on `r1` are guaranteed to eventually succeed. The output above is the result of the channel usages analysis. Channel usages describes for each channel how it is used in the process, in particular here, `r1` has type $[]_{!|!|?}$. The additional annotations with respect to the simple type says that `r1` is used in parallel for two outputs and one input. Thus we cannot say who of the two outputs succeeds but we can say that one of these will be available for the input. However collecting channel usages does not account for dependencies among channels and it does not suffice to prove strong guarantees such as deadlock freedom. Consider a slight variant of P in which the body of ser is deadlocked by keeping invoking itself:

```
TyPiCal 1.6.2: A Type-based static analyzer for
     the Pi-Calculus analyzing ping1...
*ser??r.(new r2 in ser!r2.r2?z.(r!()|r!()))
       |(new r1 in ser!!r1.r1?x)
Elapsed Time: 0.002sec
```

The output says that we will not receive any reply back from the server since we have `r1?x` (not `r1??x` as before). Notice how `r1` has the same usage as in the example before but it is deadlocked. Together with usages Typical collects also obligation and capability levels: two integers that describe the constraints imposed by the surrounding context e.g. the fact that before writing on `r` there is a input operation on `r2`.

Consider now we want to code a sequence of exchanges undertaken with ser: first we send to it an integer and then we expect a reply back. Since we are not allowed to send multiple values with different types on

each channel we need an encoding.

$$!ser(x).x(y,z).z\langle\rangle \mid ser\langle k\rangle.(\nu k_1)k\langle 5,k_1\rangle.k_1()$$

As proposed in (44) we encoded the sequence of actions, returning each time a fresh channel to be used for the subsequent action.

We however want to code such a process in a natural way for example in the variant of the π-calculus introduced in (33).

$$SESSION = !ser(y).y(x).y\langle\rangle \mid (\nu r)ser\langle r^+\rangle.r^-\langle 5\rangle.r^-()$$

We have a mechanism to create a new session channel r that accounts for two polarized sides indicated with r^+, r^- each of those assigned to the service and to the client respectively. One can argue that it could suffice a compilation function from the π-calculus with sessions to the standard π-calculus and then we can continue using Typical for analysis. This can be a good solution for sequences of only inputs and outputs. However things get much worst in presence of delegation and choices because such aspects have no correspondences in the π-calculus. Hence we decide to introduce higher level types to account for session channels.

2.3 Session handling

Session types are intended to study how to type a session channel. As expected r^- has type $!(int).?()$, an output followed by an input. However since the type of a session channel evolves together with the process, the subject reduction is guaranteed only if each side of a session channel is used linearly, i.e. in each moment during the evaluation there exists only one copy of the session channel in the syntax tree of the considered process.

Another issue of interest is finding an automatic way to distribute the two session sides between parties. The idea used in (37) is to employ for each invocation (that creates a new session) the bound output as found in πI. The resulting rule is $x(y).P|\overline{x}(y).Q \rightarrow (P|Q)$.

Finally in (8) (even if the authors do not mention session types at all) they studied a way to completely remove y from the syntax, creating a placeholder of the form $r^+ \triangleright P$ that at runtime allows to remember the current session intended as the subject of the communication. For example the process $SESSION$ is coded with

$$*ser.(x).\langle\rangle|ser.\langle 5\rangle.()$$

This time the invocation rule is $x.P|\overline{x}.Q \rightarrow (\nu r)(r^{+} \triangleright P|r^{-} \triangleright Q)$ where two fresh session side placeholders are created. The main difference is that in the formal proposal the conversation with different partners in different sessions can be carried on in interleaving, while in the proposal of (8) this is not possible, although sessions can be nested one into the other and some outputs to the parent session are possible.

The theory described in the next chapters is based on these ideas, in particular the basics of sessions types are recalled in Section 3.2. We will study and develop type systems and type inference algorithms suitable for calculi with sessions.

Chapter 3

Extending the session types framework

3.1 Introduction

The aim of this chapter is to set up the session type framework to be exploited in the rest of the thesis. We start studying the notion of subtyping among session types which is the notion proposed in (33) with only minor differences. First we consider non ordered branches in each choice. Second, to have a clear separation between input/output of ordinary values with input/ouput of sessions, we handle them in different rules. According to the subtyping pre-order, we consider the problem of computing the standard meet and join of two session types. (Often in the text we use term meet or intersection and term join or union interchangeably.) We show that computing the meet and the join can be reduced to compute only one operation, the meet since the duality operation among session types allows switching the operator's order in the subtyping relation (e.g. $T \leq U$ iff $\overline{U} \leq \overline{T}$).

The subtyping relation is given in a co-inductive setting (i.e. the subtyping relation is defined as the greatest typing simulation relation) then we define the intersection co-inductively too. Moreover since the subtyping relation is not defined on each pair of session types and our notion of intersection is derived directly from the subtyping relation then the intersection is not always defined. We think that this way of having partial operations is more natural than adding some element to indicate that the

operation is not defined.

However both the subtyping algorithm and the meet algorithm show how to compute the intersection among closed types, but often these relations appear in a type which involves free type variables. The problem is that we want to know if there exists some substitution for the variables that allows a constraint to be satisfied. The first simplification we made is to restrict our setting, by requiring the syntactic equality of the sent/received values, which is not necessary in the general setting where sessions are contravariant with respect to the output and covariant with respect to the input. With this simplification we introduce an algorithm to compute the most general syntactic unifier which, when the free variables appear only in the communicated types (types as the argument of either an input or an output action), turns out to be sound and complete in the sense that it holds if and only if there exists a substitution that satisfies the constraint. In presence of generic free variables deciding the existence of a satisfying substitution becomes harder. We develop two algorithms that try to find a solving substitution for a set of meet constraints (i.e. constraints that are satisfied if the intersection between two types exists). The first algorithm is rather simple w.r.t. to the computational cost but it is only correct. The second one is (we claim) sound and complete. We do not prove formally this fact since it is not used elsewhere but we think it is interesting to outline the complexity of solving constraints without any assumptions on free variables. Solving a set of meet exists constraints is an important fact since also subtyping constraints can be reduced to a set of meet exists constraints as we shall show.

Background: Session types were first proposed in (37) and the subtyping relation $\leq$ and the proof that $\leq$ is a pre-order are given in (33). In that work authors give the algorithm to compute the subtyping relation. This algorithm and the others we propose for the membership checking of the greatest fixed point using a set of assumptions are inspired to (2) but as outlined in (31) the proofs are simpler in a co-inductive setting rather then limits of sequences of approximations. All remaining contents are introduced here.

3.2 Session Types

We assume an infinite collection $\alpha, \dots$ of type variables and an infinite collection $l, \dots$ of labels. We distinguish session types T and types

$$
\begin{array}{llll}
T, U & ::= & \mathtt{end} & \text{(no action)} \\
 & | & ?(S_1, \ldots, S_n).T & \text{(input of a tuple)} \\
 & | & !(S_1, \ldots, S_n).T & \text{(output of a tuple)} \\
 & | & ?(U).T & \text{(input of a session)} \\
 & | & !(U).T & \text{(output of a session)} \\
 & | & \&\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(external choice)} \\
 & | & \oplus\{l_1 : T_1, \ldots, l_n : T_n\} & \text{(internal choice)} \\
 & | & \mu\alpha.T & \text{(recursive behavior)} \\
 & | & \alpha & \text{(type variable)} \\
 & & & \\
S & ::= & int & \text{(basic type)} \\
 & | & [T] & \text{(service reference)}
\end{array}
$$

Figure 1: Syntax of types

for message contents S, which we often call sort types, defined by the grammar in Figure 1. Type end represents the type of a session in which no further communications are allowed. The types $?(S_1, \ldots, S_n).T$ and $!(S_1, \ldots, S_n).T$ represent respectively an input and an output of tuple of type $S_1, \ldots, S_n$ respectively, followed by the continuation T. The types $!(U).T$ and $?(U).T$ are similar but they allow for session delegation. The types $\&\{l_1 : T_1, \ldots, l_n : T_n\}$ and $\oplus\{l_1 : T_1, \ldots, l_n : T_n\}$ denote respectively an external and an internal choice. The external choice is used to offer a set of options to the partner while the internal choice expresses selections a partner may wish to perform. The recursive expression $\mu\alpha.T$ represents recursive behaviors and μ is a binder that gives rise, in the standard way, to notions of bound (bv) and free variables (fv), closed μ-types, and equivalence of μ-types up to renaming of bound variables. In the proofs we also use the function $\mathsf{unfold}(T)$ defined by recursion on the structure of T: $\mathsf{unfold}(\mu\alpha.T) = \mathsf{unfold}(T[^{\mu\alpha.T}/_{\alpha}])$ and in all other cases $\mathsf{unfold}(T) = T$. Finally, the type of the sent/received type can be both int, a basic data value, and $[T]$, a service name that behaves according to T. We often omit the trailing end and we write $\tilde{T}$ for a sequence $T_1, \ldots, T_n$ of types and $\diamondsuit\{l_i : T_i\}_{i \in I}$ for $\diamondsuit\{l_{i_1} : T_{i_1}, \ldots, l_{i_n} : T_{i_n}\}$ where $I = \{i_1, \ldots, i_n\}$ and $\diamondsuit \in \{\oplus, \&\}$. We shall use sometimes also the (partial) function $;;$ defined for both kinds of choice as:

$$
\diamondsuit\{l : T\} \;;;\; \diamondsuit\{l_1 : T_1, \ldots, l_n : T_n\} = \begin{cases} \diamondsuit\{l_1 : T_1, \ldots, l : T, \ldots, l_n : T_n\} & \text{if } l \notin \{l_1, \ldots, l_n\} \\ \textit{undefined} & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
\overline{?(\tilde{S}).T} = !(\tilde{S}).\overline{T} \qquad \overline{?(U).T} = !(U).\overline{T} \\
\overline{!(\tilde{S}).T'} = ?(\tilde{S}).\overline{T'} \qquad \overline{!(U).T'} = ?(U).\overline{T'} \\
\overline{\&\{l_1 : T_1, \ldots, l_n : T_n\}} = \oplus\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\} \\
\overline{\oplus\{l_1 : T_1, \ldots, l_n : T_n\}} = \&\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\} \\
\overline{\mu\alpha.T} = \mu\alpha.\overline{T} \qquad \overline{\alpha} = \alpha \qquad \overline{\texttt{end}} = \texttt{end}
\end{array}
$$

Figure 2: The syntactic dual of a session types

inductively extended as expected when the first operator is a general choice $\diamondsuit\{l_j : T'_j\}_{j\in J}$.

Definition 3.1 (Well formedness)**.** A type T is *well formed* if it is contractive (i.e., in any subexpression $\mu\alpha.\mu\alpha_1 \ldots \mu\alpha_n.T'$ the body T' is not α, see (31)) and if the same label does not appear twice in a choice (i.e., in any $\diamondsuit\{l_i : T_i\}_{i\in I}$ if $i \neq j$ then $l_i \neq l_j$).

For simplicity we call types only well formed types and we denote with $TYPE$ the set of all types. Each session type T has a syntactic dual type $\overline{T}$, inductively defined by the equations in Figure 2, notice that the communicated types are unchanged by duality e.g. $\overline{?(?(int).\texttt{end}).\texttt{end}} = !(?(int).\texttt{end}).\texttt{end}$.

The following is a direct consequence of the duality.

Lemma 3.2. *$T = \overline{\overline{T}}$ i.e. the duality is involutive.*

Example 3.3. We now present a few session types that will serve as a running example:

$$
\begin{array}{ll}
T = \mu\alpha.\&\{\mathbf{quit} : \texttt{end}, & V = \mu\alpha.\&\{\mathbf{quit} : \texttt{end}, \\
\quad \mathbf{play} : \mu\beta.\&\{\mathbf{prev} : \alpha, & \quad \mathbf{play} : \mu\beta.\&\{\mathbf{prev} : \alpha, \\
\qquad \mathbf{vol} : ?(int).\beta\}\} & \qquad \mathbf{bright} : ?(int).\beta\}\}
\end{array}
$$

$$
\begin{array}{ll}
U = \mu\alpha.\&\{\mathbf{quit} : \texttt{end}, & U' = \mu\alpha.\&\{\mathbf{play} : \&\{\mathbf{prev} : \alpha\}\} \\
\quad \mathbf{play} : \&\{\mathbf{prev} : \alpha\}\} &
\end{array}
$$

Type T represents the type of a menu that offers two choices: **quit** which allows to exit from the menu and **play** which allows accessing a submenu. The submenu is composed of two choices: **prev** permits returning to the main menu and **vol** that after communicating an integer returns to the submenu. Type V is similar but for the branch **bright** in

place of **vol**. Now it is clear that U has common behaviors of both menus T and V and it is a possible candidate for their meet. In fact, U can offer either the option **quit** or the two consecutive options **play** and **prev**. U' it is also a possible candidate for the meet since it has the common behaviors of T and V, but it lacks the option for **quit**. Intuitively we want U to be the meet of T and V, not U'.

3.3 Intersection of Session Types

3.3.1 Algebraic meet of session types

In this section we discuss the algebraic notion of intersection with respect to the pre-order $\leq$ given in (33).

Definition 3.4. A relation $R \subseteq TYPE \times TYPE$ is a *type simulation relation* if $(T, V) \in R$ implies the following conditions:

1. if $\mathsf{unfold}(T) = !(S_1, \ldots, S_n).T'$ then $\mathsf{unfold}(V) = !(S_1, \ldots, S_n).V'$ and $(T', V') \in R$
2. if $\mathsf{unfold}(T) = !(U).T'$ then $\mathsf{unfold}(V) = !(U').V'$ and $(U', U) \in R$ and $(T', V') \in R$
3. if $\mathsf{unfold}(T) = ?(S_1, \ldots, S_n).T'$ then $\mathsf{unfold}(V) = ?(S_1, \ldots, S_n).V'$ and $(T', V') \in R$
4. if $\mathsf{unfold}(T) = ?(U).T'$ then $\mathsf{unfold}(V) = ?(U').V'$ and $(U, U') \in R$ and $(T', V') \in R$
5. if $\mathsf{unfold}(T) = \&\{l_1 : T_1, \ldots, l_m : T_m\}$ then $\mathsf{unfold}(V) = \&\{l'_1 : V_1, \ldots, l'_n : V_n\}$ where $m \leq n$ and for all $i \in \{1, \ldots, m\}$ there exists one $j \in \{1, \ldots, n\}$ such that $l_i = l'_j$ and $(T_i, V_j) \in R$
6. if $\mathsf{unfold}(T) = \oplus\{l_1 : T_1, \ldots, l_m : T_m\}$ then $\mathsf{unfold}(V) = \oplus\{l'_1 : V_1, \ldots, l'_n : V_n\}$ where $n \leq m$ and for all $i \in \{1, \ldots, n\}$ there exists one $j \in \{1, \ldots, m\}$ such that $l'_i = l_j$ and $(T_j, V_i) \in R$
7. if $\mathsf{unfold}(T) = \mathsf{end}$ then $\mathsf{unfold}(V) = \mathsf{end}$

The co-inductive $\leq$ relation is defined as $T \leq V$ if there exists a type simulation relation R such that $(T, V) \in R$, i.e. it is the largest type simulation relation.

Roughly $T \leq V$ holds if T refines the behavior of V. Each case of a type simulation relation is in an or-relation with the others. Cases for input/ouput prefixes for sort types S are simple: they require the syntactic equality of the argument and they constrain the continuation to be also in the relation. Input/output prefixes for session types require the subtyping relation computed in depth, thus they require contravariance for output case and covariance for the input case respectively (63). Instead, an external choice is in subtype relation with another external choice if it contains a subset of the labels. This, for example, allows replacing an external choice when we are sure that only a subset of the real available labels is offered. On the other hand internal choice behaves dually: we can chose more labels than the actual ones. The greatest fixed point of the subtyping definition exists since the associate function is monotone. The following is a support lemma used in the proof of Proposition 3.6.

Lemma 3.5. *Let* $T \leq V$.

1. *if* $\mathsf{unfold}(V) = !(S_1, \ldots, S_n).V'$ *then* $\mathsf{unfold}(T) = !(S_1, \ldots, S_n).T'$ *and* $T' \leq V'$
2. *if* $\mathsf{unfold}(V) = !(U').V'$ *then* $\mathsf{unfold}(T) = !(U).T'$ *and* $U' \leq U$ *and* $T' \leq V'$
3. *if* $\mathsf{unfold}(V) = ?(S_1, \ldots, S_n).V'$ *then* $\mathsf{unfold}(T) = ?(S_1, \ldots, S_n).T'$ *and* $T' \leq V'$
4. *if* $\mathsf{unfold}(V) = ?(U').V'$ *then* $\mathsf{unfold}(T) = ?(U).T$ *and* $U \leq U'$ *and* $T' \leq V'$
5. *if* $\mathsf{unfold}(V) = \&\{l'_1 : V_1, \ldots, l'_m : V_m\}$ *then* $\mathsf{unfold}(T) = \&\{l_1 : T_1, \ldots, l_n : T_n\}$ *where* $m \leq n$ *and for all* $i \in \{1, \ldots, m\}$ *there exists one* $j \in \{1, \ldots, n\}$ *such that* $l_i = l'_j$ *and* $T_i \leq V_j$
6. *if* $\mathsf{unfold}(V) = \oplus\{l_1 : V_1, \ldots, l_m : V_m\}$ *then* $\mathsf{unfold}(T) = \oplus\{l'_1 : T_1, \ldots, l'_n : T_n\}$ *where* $n \leq m$ *and for all* $i \in \{1, \ldots, n\}$ *there exists one* $j \in \{1, \ldots, m\}$ *such that* $l'_i = l_j$ *and* $T_i \leq V_j$
7. *if* $\mathsf{unfold}(V) = \mathsf{end}$ *then* $\mathsf{unfold}(T) = \mathsf{end}$

Proof. The proof is first by inspection and then by contradiction, we show the first case the others are similar. If $\mathsf{unfold}(T)$ has not the form $!(S_1, \ldots, S_n).T'$ then $\mathsf{unfold}(V)$ cannot have the form $!(S_1, \ldots, S_n).V'$, contradicting the hypothesis. If $\mathsf{unfold}(T) \;=!(S_1, \ldots, S_n).T'$ and $\mathsf{unfold}(V) =!(S_1, \ldots, S_n).V'$ then it must be the case that $T' \leq U'$. □

$$\frac{T \leq U \in \Sigma}{\Sigma \vdash T \leq U} \;(\text{AS-Assump}) \qquad \texttt{end} \leq \texttt{end} \;(\text{AS-End})$$

$$\frac{\Sigma, \mu\alpha.T \leq U \vdash T[^{\mu\alpha.T}/_{\alpha}] \leq U}{\Sigma \vdash \mu\alpha.T \leq U} \;(\text{AS-RecL}) \qquad \frac{\Sigma, T \leq \mu\alpha.U \vdash T \leq U[^{\mu\alpha.U}/_{\alpha}]}{\Sigma \vdash T \leq \mu\alpha.U} \;(\text{AS-RecR})$$

$$\frac{\Sigma \vdash T \leq U}{\Sigma \vdash ?(\tilde{S}).T \leq ?(\tilde{S}).U} \;(\text{AS-In}) \qquad \frac{\Sigma \vdash T \leq U}{\Sigma \vdash !(\tilde{S}).T \leq !(\tilde{S}).U} \;(\text{AS-Out})$$

$$\frac{\Sigma \vdash V \leq V' \quad \Sigma \vdash T \leq U}{\Sigma \vdash ?(V).T \leq ?(V').U} \;(\text{AS-Catch}) \qquad \frac{\Sigma \vdash V' \leq V \quad \Sigma \vdash T \leq U}{\Sigma \vdash !(V).T \leq !(V').U} \;(\text{AS-Throw})$$

$$\frac{|I| \leq |J| \quad \forall i \in I \; \exists j \in J \Rightarrow l_i = l'_j \,,\, \Sigma \vdash T_i \leq T'_j}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \leq \&\{l'_j : T'_j\}_{j \in J}} \;(\text{AS-Select})$$

$$\frac{|J| \leq |I| \quad \forall j \in J \; \exists i \in I \Rightarrow l_i = l'_j \,,\, \Sigma \vdash T_i \leq T'_j}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \leq \oplus\{l'_j : T'_j\}_{j \in J}} \;(\text{AS-Choice})$$

Figure 3: The subtyping algorithm

This co-inductive characterization of the subtyping relation is useful since we have a complete and sound algorithm to compute it. The subtyping algorithm reported in Figure 3 is used to verify that two types are in subtyping relation or said in other words if there exists a type simulation that contains them. As usual for algorithms that check the greatest fixed point membership, the subtype algorithm takes a set Σ of assumptions avoiding unfold indefinitely the recursion. Often paying the cost of being slightly imprecise we use the word algorithm referencing a set of inferences rules to indicate the derived algorithm obtained applying rules from the conclusion to the premises. Of course, each time should exist only one possible rule that matches the algorithm input and if no such rule exists then the algorithm is intended to terminate with failure. Thus imposing that rule (AS-Assump) is applied in place of rules (AS-RecL) and (AS-RecR) whenever is possible, we disambiguate the rule application. Hence the inference rules in Figure 3 constitute in fact an algorithm. In rules (AS-Choice) and (AS-Select) we use the set notation and $|I|$, $|J|$ stay for the cardinality of I and J respectively.

We shall need an additional proposition which relates the $\leq$ relation with the duality relation:

Proposition 3.6. *If $T \leq V$ then $\overline{V} \leq \overline{T}$.*

Proof. It suffices to prove that $R = \{(\overline{V}, \overline{T}) | T \leq V\} \cup \{(T, V) | T \leq V\}$ is a type simulation relation. We sketch the session input and external choice cases. If $\mathsf{unfold}(\overline{V}) = ?(U).\overline{V'}$ we have to show that $\mathsf{unfold}(\overline{T}) = ?(U').\overline{T'}$ with $(U, U') \in R$ and $(\overline{V'}, \overline{T'}) \in R$. Since $(\overline{V}, \overline{T}) \in R$ we know that $T \leq V$. Since $\mathsf{unfold}(\overline{V}) = ?(U).\overline{V'}$ we know by Lemma 3.5 that, $\mathsf{unfold}(\overline{T}) = ?(U').\overline{T'}$ for suitable U' and T'. Therefore $\mathsf{unfold}(\overline{V}) = !(U).V'$ and $\mathsf{unfold}(\overline{T}) = !(U').T'$ and since $T \leq V$ it must be the case that $U \leq U'$ and $T' \leq V'$, hence $(U, U') \in R$ and $(\overline{V'}, \overline{T'}) \in R$. Similarly, if $\mathsf{unfold}(\overline{V}) = \&\{l_1 : \overline{V_1}, \ldots, l_m : \overline{V_m}\}$ then $\mathsf{unfold}(\overline{T}) = \&\{l'_1 : \overline{T_1}, \ldots, l'_n : \overline{T_n}\}$ and $m \leq n$ and for all $i \in \{1, \ldots, m\}$ there exists one $j \in \{1, \ldots, n\}$ such that $l_i = l'_j$ and $(\overline{V_i}, \overline{T_j}) \in R$ since $(T_j \leq V_i)$. ☐

Corollary 3.7. *$T \leq V$ iff $\overline{V} \leq \overline{T}$*

Proof. By the previous proposition and by Lemma 3.2. ☐

The above corollary is proved also in (69) but by induction on the algorithm rather than co-inductively. In the following we shall use this property of subtyping and duality relation naturally, without any reference.

The three following propositions are proved in (33). Apart from the algorithm termination the second one states the soundness and the completeness of the algorithm w.r.t. the co-inductive definition and the third one states that $\leq$ is in fact a preorder; that is $\leq$ is reflexive and transitive.

Proposition 3.8 (see Lemma 10 in (33))**.** *The subtyping algorithm always terminates.*

Proof. The idea of the proof is to define the set $Sub(T)$ to be the set of all subterms of T, with free type variables replaced by their recursive definitions. For any T, $Sub(T)$ is finite because its size is bounded by the number of distinct subterms of T. We refer the original work for further details. ☐

Proposition 3.9 (see Corollary 2 in (33))**.** *$\emptyset \vdash T \leq U$ iff $T \leq U$.*

Proposition 3.10 (see Propositions 2 and 3 in (33))**.** *$\leq$ is a pre-order.*

We let $\lesseqgtr$ be the equivalence relation induced by the pre-order $\leq$, e.g. $!(int).\mu\alpha.!(int).\alpha \lesseqgtr \mu\alpha.!(int).\alpha$.

Definition 3.11. We write $T \lesseqgtr V$ iff $T \leq V$ and $V \leq T$ hold.

Consequently, $\leq$ can be considered a partial order up to $\lesseqgtr$. We write $\mathsf{greatest}(T, U, V)$ for the predicate $\forall\ V'.V' \leq T$ and $V' \leq U \Rightarrow\ V' \leq V$. In the same manner $\mathsf{least}(T, U, V)$ is the predicate $\forall\ V'.T \leq V'$ and $U \leq V' \Rightarrow\ V \leq V'$. Since the relation $\leq$ is a partial order up to $\lesseqgtr$ (the anti-symmetry holds only if we consider $\lesseqgtr$ as equivalence relation), defining the meet of two types requires a bit of attention. The problem is due to the fact that given two types there exists a (possibly infinite) set of greatest lower bounds. In particular all the elements of such a set belong to the equivalence class generated by the relation $\lesseqgtr$. In few words, if V is the greatest lower bound of T and U then all the elements of $[V]_{\lesseqgtr}$ (the equivalence class of V w.r.t. $\lesseqgtr$) also are. Thereby the uniqueness of the meet is intended with respect to the notion of $\lesseqgtr$. Also since the subtyping relation is not defined on every pairs of session types then also the intersection is not always defined.

Definition 3.12 (Type meet and type join)**.** The meet of two session types $T \wedge U$ is the (unique w.r.t. $\lesseqgtr$ when it exists) type V such that $V \leq T$ and $V \leq U$ and $\mathsf{greatest}(T, U, V)$. Similarly the join of two session types $T \vee U$ is the (unique w.r.t. $\lesseqgtr$ when it exists) type V such that $T \leq V$ and $U \leq V$ and $\mathsf{least}(T, U, V)$.

The operation of duality allows switching the meet with join.

Lemma 3.13. $T \wedge U \lesseqgtr \overline{\overline{T} \vee \overline{U}}$

Proof. It suffice to prove that $\overline{T \wedge U} \lesseqgtr \overline{T} \vee \overline{U}$ then by definition $T \wedge U$ is the type V s.t. $V \leq T$ and $V \leq U$ and $\forall\ V'.V' \leq T$ and $V' \leq U \Rightarrow\ V' \leq V$ hence $\overline{T} \leq \overline{V}$ and $\overline{U} \leq \overline{V}$ and $\forall\ V'.\overline{T} \leq \overline{V'}$ and $\overline{U} \leq \overline{V'} \Rightarrow\ \overline{V} \leq \overline{V'}$ which concludes. □

It is also easy to see that the following holds:

Lemma 3.14. *For any T, U, V:*

- *Idempotency:* $(T \wedge T) \lesseqgtr T$
- *Commutativity:* $(T \wedge U) \lesseqgtr (U \wedge T)$
- *Associativity:* $T \wedge (U \wedge V) \lesseqgtr (T \wedge U) \wedge V$

Proof. **Idempotency**: By definition $T \wedge T$ is the type V such that $V \leq T$ and $V \leq T$ and $\mathsf{greatest}(T, T, V)$ and the result follows taking $V \lesseqgtr T$.
Commutativity: By definition $T \wedge U$ is the type V such that $V \leq T$ and $V \leq U$ and $\mathsf{greatest}(T, U, V)$ or equivalently is the type V such that $T \leq V$ and $U \leq V$ and $\mathsf{greatest}(U, T, V)$.
Associativity: By definition $U \wedge V$ is the type V_1 such that $V_1 \leq U$ and $V_1 \leq V$ and $\mathsf{greatest}(U, V, V_1)$ and $T \wedge V_1$ is the type V_2 such that $V_2 \leq T$ and $V_2 \leq V_1$ and $\mathsf{greatest}(T, V_1, V_2)$. Also $T \wedge U$ is the type V_3 such that $V_3 \leq T$ and $V_3 \leq U$ and $\mathsf{greatest}(T, U, V_3)$ and $V_3 \wedge V$ is the type V_4 such that $V_4 \leq V_3$ and $V_4 \leq V$ and $\mathsf{greatest}(V_3, V, V_4)$. The result follows since $V_2 \lesseqgtr V_4$ exploiting the transitivity of $\leq$.

□

Example 3.15. Consider the types given in Section 1. We have both $U \leq T$, $U \leq V$ and $U' \leq T$, $U' \leq V$ but since $U' \leq U$ then U is more suitable than U' to be the meet of T and V, but for now we do not have an automatic manner to verify that $\mathsf{greatest}(T, V, U)$ holds since we have an universal quantification over types.

3.3.2 Inference Relations for the Meet

In this section we overcome the problem related to the universal quantification in the definition of the predicate greatest. To this aim we introduce two new relations: one that checks whether a type is effectively the meet of two given types and another one that checks the existence of the meet for two given types.

Definition 3.16. A relation $R \subseteq TYPE \times TYPE \times TYPE$ is a *meet relation* if $(T, U, V) \in R$ implies the following conditions:

1. if $\mathsf{unfold}(T) = !(S_1, \ldots, S_n).T'$ then $\mathsf{unfold}(U) = !(S_1, \ldots, S_n).U'$ and $\mathsf{unfold}(V) = !(S_1, \ldots, S_n).V'$ and $(T', U', V') \in R$

2. if $\mathsf{unfold}(T) = !(T_1).T'$ then $\mathsf{unfold}(U) = !(U_1).U'$ and $\mathsf{unfold}(V) = !(V_1).V'$ and $(\overline{T_1}, \overline{U_1}, \overline{V_1}) \in R$ and $(T', U', V') \in R$

3. if $\mathsf{unfold}(T) = ?(S_1, \ldots, S_n).T'$ then $\mathsf{unfold}(U) = ?(S_1, \ldots, S_n).U'$ and $\mathsf{unfold}(V) = ?(S_1, \ldots, S_n).V'$ and $(T', U', V') \in R$

4. if $\mathsf{unfold}(T) = ?(T_1).T'$ then $\mathsf{unfold}(U) = ?(U_1).U'$ and $\mathsf{unfold}(V) = ?(V_1).V'$ and $(T_1, U_1, V_1) \in R$ and $(T', U', V') \in R$

5. if $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i\in I}$ then $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j\in J}$ and letting $K = \{(i,j) \mid l_i = l'_j\}$ with $|K| \geq 1$ we have $\mathsf{unfold}(V) = \&\{l_i : U_{(i,j)}\}_{(i,j)\in K}$ and $\forall i \forall j (i,j) \in K \Rightarrow (T_i, T'_j, U_{(i,j)}) \in R$

6. if $\mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i\in I}$ then $\mathsf{unfold}(U) = \oplus\{l'_j : T'_j\}_{j\in J}$ and letting $K = \{(i,j) \mid l_i = l'_j\}$, $I' = I \setminus \mathsf{fst}(K)$,$J' = J \setminus \mathsf{snd}(K)$ we have $\mathsf{unfold}(V) = \oplus\{l_i : U_{(i,j)}\}_{(i,j)\in K} \;;;\; \oplus\{l_i : T_{\mu i}\}_{i\in I'} \;;;\; \oplus\{l'_j : T'_{\mu j}\}_{j\in J'}$ and $\forall i \in I'\ T_{\mu i} \lesseqgtr T_i$ and $\forall j \in J'\ T'_{\mu j} \lesseqgtr T'_j$ and $\forall i \forall j (i,j) \in K \Rightarrow (T_i, T'_j, U_{(i,j)}) \in R$

7. if $\mathsf{unfold}(T) = \mathtt{end}$ then $\mathsf{unfold}(U) = \mathtt{end}$ and $\mathsf{unfold}(V) = \mathtt{end}$

The co-inductive $_\overset{c}{\wedge}_ = _$ relation is defined as $T\overset{c}{\wedge}U = V$ if there exists a meet relation R such that $(T, U, V) \in R$.

Once again it is simple to see that the function associated to the meet relation is monotonic. Since the output of a session is contravariant, the respective rule (point 2) requires the syntactic dual of each sent type to be co-inductively in the relation. The duality relation is exploited here to use the same algorithm but to compute the union (or the join). The rule in fact would require to compute the least upper bound of T_1 and U_1. Rules for choices use the intersection set K which contains pairs of indexes with equal labels in each choice. Remember that we are considering only well formed types, hence labels in choices are distinct and consequently each component in a pair belonging to K is distinct too. Accordingly, the type $\diamondsuit\{l_i : U_{(i,j)}\}_{(i,j)\in K}$ has all the labels in the intersection where l_i are some of the labels indexed by I (here we used l_i the labels of the first type, but l'_j would do too). Moreover, the rule for external choice uses the $;;$ operation (as a destructor) to exhibit the type V and both functions fst and snd to project each component of K. The definition of these functions is straightforward, i.e. $\mathsf{fst}(K) = \{i|(i,j) \in K\}$ and $\mathsf{snd}(K) = \{j|(i,j) \in K\}$. The resulting type contains branches in the intersection (indexed by K) which are constrained to be co-inductively in the relation together with both $T_{\mu i}$ equal to the remaining branches of the type T (indexed by I') and $T'_{\mu i}$ equal to the remaining branches of the type V (indexed by J'). There is a subtlety in the rule for external choice, as the relation is defined only if K contains at least one element. A similar check is not necessary for the internal choice since the requirement is automatically fulfilled as T and V contain at least one branch each.

We first prove that each representant of the algebraic intersection is also captured by the greatest meet relation.

Proposition 3.17. *If $T \wedge U \lesseqgtr V$ then $T \overset{c}{\wedge} U = V$.*

Proof. We show that $R = \{(T, U, V) | V \leq T$ and $V \leq U$ and $\mathsf{greatest}(T, U, V)\}$ is a meet relation. We prove the theorem by case analysis on $\mathsf{unfold}(V)$:

- If $\mathsf{unfold}(V) = !(V_1).V'$ by definition of $\leq$ we must have $\mathsf{unfold}(T) = !(T_1).T'$ with $T_1 \leq V_1$ and $V' \leq T'$. Similarly for U, $\mathsf{unfold}(U) = !(U_1).U'$ with $U_1 \leq V_1$ and $V' \leq U'$. By Proposition 3.6 we have $\overline{V_1} \leq \overline{T_1}$ and $\overline{V_1} \leq \overline{U_1}$. To conclude the proof, note that $\mathsf{greatest}(\overline{T_1}, \overline{U_1}, \overline{V_1})$ and $\mathsf{greatest}(T', U', V')$ trivially hold by $\mathsf{greatest}(T, U, V)$. In fact, by definition of $\mathsf{greatest}(T, U, V)$ we have $\forall W$ s.t. $\mathsf{unfold}(W) = !(W_1).W'$ and $W \leq T$ and $W \leq U \Rightarrow W \leq V$. By definition of $\leq$ we have $W' \leq T'$ and $W' \leq U' \Rightarrow W' \leq V'$ or in other words $\mathsf{greatest}(T', U', V')$. In the same manner we have $\overline{W_1} \leq \overline{T_1}$ and $\overline{W_1} \leq \overline{U_1} \Rightarrow \overline{W_1} \leq \overline{V_1}$, i.e. $\mathsf{greatest}(\overline{T_1}, \overline{U_1}, \overline{V_1})$. In brief we proved: $V' \leq T'$, $\overline{V_1} \leq \overline{T_1}$, $V' \leq U'$, $\overline{V_1} \leq \overline{U_1}$, $\mathsf{greatest}(T', U', V')$ and $\mathsf{greatest}(\overline{T_1}, \overline{U_1}, \overline{V_1})$ which allow to conclude.

- If $\mathsf{unfold}(V) = \oplus\{l_i : V_i\}_{i \in I}$ by definition of $\leq$ we must have $\mathsf{unfold}(T) = \oplus\{l'_j : T_j\}_{j \in J}$ and $|J| \leq |I|$ and for all $j \in J$ there exists $i \in I$ s.t. $l_i = l'_j$ and $V_i \leq T_j$. Similarly for U, $\mathsf{unfold}(U) = \oplus\{l''_i : U_i\}_{i \in J'}$ and $|J'| \leq |I|$ and for all $j \in J'$ there exists $i \in I$ s.t. $l_i = l''_j$ and $V_i \leq U_j$. Take $K = \{(i, j) | l'_i = l''_j$ and $i \in J$ and $j \in J'\}$ it holds that for some $(i, j) \in K$, $V_{(i,j)} \leq T_i$ and $V_{(i,j)} \leq U_j$ and since $\mathsf{greatest}(T, U, V)$ the previous inequalities hold for all $(i, j) \in K$. Take now any i s.t. $i \in J \setminus \mathsf{fst}(K)$ then it holds for some $k' \in I$ that $V_{k'} \leq T_i$ but $\mathsf{greatest}(T, U, V)$ implies $V_{k'} \lesseqgtr T_i$ and the same for any i s.t. $i \in J' \setminus \mathsf{snd}(K)$ it holds for some $k'' \in I$ that $V_{k''} \lesseqgtr U_i$. Since $\mathsf{greatest}(T, U, V)$ then I does not contain any additional branch. To finish we must prove that greatest holds for the elements indexed by K. By definition of $\mathsf{greatest}(T, U, V)$ we have $\forall W$ s.t. $\mathsf{unfold}(W) = \oplus\{l'''_i : W_i\}_{i \in I'}$ and $W \leq T$ and $W \leq U \Rightarrow W \leq V$. By definition of $\leq$ we have $|J| \leq |I'|$ and for all $j \in J$ there exists $i \in I'$ s.t. $l'''_i = l'_j$ and $W_i \leq T_j$ and $|J'| \leq |I'|$ and for all $j \in J'$ there exists $i \in I'$ s.t. $l'''_i = l''_j$ and $W_i \leq U_j \Rightarrow |I| \leq |I'|$ and for all $i \in I$ there exists $j \in I'$ s.t. $l_i = l'''_j$ and $W_j \leq V_i$ and in particular $\forall (i, j) \in K$ $\mathsf{greatest}(T_i, U_j, V_{(i,j)})$.

- If $\mathsf{unfold}(V) = \&\{l_i : V_i\}_{i \in I}$ by definition of $\leq$ we must have $\mathsf{unfold}(T) = \&\{l'_j : T_j\}_{j \in J}$ and $|I| \leq |J|$ and for all $i \in I$ there exists $J \in J$ s.t. $l_i = l'_j$ and $V_i \leq T_j$. Similarly for U, $\mathsf{unfold}(U) = \&\{l''_i :$

$U_i\}_{i\in J'}$ and $|I| \leq |J'|$ and for all $i \in I$ there exists $j \in J'$ s.t. $l_i = l''_j$ and $V_i \leq U_j$. Take $K = \{(i,j) | l'_i = l''_j$ and $i \in J$ and $j \in J'\}$, it is obvious that $|K| \geq 1$ and since greatest(T,U,V) we have $|K| = |I|$. The remaining part that greatest holds for the elements of K is similar to the previous case.

- The remaining cases are analogous but simpler.

□

In order to prove that the greatest meet relation captures only correct representatives of the intersection we need to define co-inductively the notions of minorant.

Definition 3.18. A relation $R \subseteq TYPE \times TYPE \times TYPE$ is a *minorant relation* if $(T,U,V) \in R$ implies the following conditions:

1. if unfold$(T) = !(S_1,\ldots,S_n).T'$ then unfold$(U) = !(S_1,\ldots,S_n).U'$ and unfold$(V) = !(S_1,\ldots,S_n).V'$ and $(T',U',V') \in R$
2. if unfold$(T) = !(T_1).T'$ then unfold$(U) = !(U_1).U'$ and unfold$(V) = !(V_1).V'$ and $(\overline{T_1},\overline{U_1},\overline{V_1}) \in R$ and $(T',U',V') \in R$
3. if unfold$(T) = ?(S_1,\ldots,S_n).T'$ then unfold$(U) = ?(S_1,\ldots,S_n).U'$ and unfold$(V) = ?(S_1,\ldots,S_n).V'$ and $(T',U',V') \in R$
4. if unfold$(T) = ?(T_1).T'$ then unfold$(U) = ?(U_1).U'$ and unfold$(V) = ?(V_1).V'$ and $(T_1,U_1,V_1) \in R$ and $(T',U',V') \in R$
5. if unfold$(T) = \&\{l_i : T_i\}_{i\in I}$ then unfold$(U) = \&\{l'_j : T'_j\}_{j\in J}$ and letting $K = \{(i,j) \mid l_i = l'_j\}$ for any $K' \subseteq K$ and $|K'| \geq 1$ we have unfold$(V) = \&\{l_i : U_{(i,j)}\}_{(i,j)\in K'}$ and $\forall i \forall j (i,j) \in K' \Rightarrow (T_i, T'_j, U_{(i,j)}) \in R$
6. if unfold$(T) = \oplus\{l_i : T_i\}_{i\in I}$ then unfold$(U) = \oplus\{l'_j : T'_j\}_{j\in J}$ and letting $K = \{(i,j) \mid l_i = l'_j\}$, $I' = I \setminus$ fst(K) ,$J' = J \setminus$ snd(K) we have unfold$(V) = \oplus\{l_i : U_{(i,j)}\}_{(i,j)\in K}$;; $\oplus\{l_i : T_{\mu i}\}_{i\in I'}$;; $\oplus\{l'_j : T'_{\mu j}\}_{j\in J'}$;; $\oplus\{l''_k : T''_k\}_{k\in K'}$ for any $\oplus\{l''_k : T''_k\}_{k\in K'}$ and $\forall i \in I'\ T_{\mu i} \leq T_i$ and $\forall j \in J'\ T'_{\mu j} \leq T'_j$ and $\forall i \forall j (i,j) \in K \Rightarrow (T_i, T'_j, U_{(i,j)}) \in R$
7. if unfold$(T) =$ end then unfold$(U) =$ end and unfold$(V) =$ end

The co-inductive minorant relation is defined as minorant(T,U,V) if there exists a minorant relation R such that $(T,U,V) \in R$.

The rules are similar to the corresponding ones in the definition of a meet relation but for cases 5 and 6. In case 5 we allow in the resulting type a subset K' of K; that is, having only some of the common branches of T and U characterizes all minorants of T and U. Case 6 allows the type subscripted with μ to be also less than the type in the corresponding branch (either indexed by I' or J') but also it allows an arbitrary internal choice indexed by K' to be part of the final type. In this manner we characterize all minorants of an external choice. Notice the monotonicity also holds for the function associated to a minorant relation, in particular rules 5 and 6 have an universal quantification that keeps the monotonicity.

The next theorem shows that indeed $_\overset{c}{\wedge}_ = _$ relates two types with their respective meet.

Theorem 3.19. *$T \wedge U \leqslant V$ iff $T \overset{c}{\wedge} U = V$*

Proof. The if part follows by the previous Proposition, for the only if part it suffices to show that:

1. $R_T = \{(V,T) | \exists U \; T \overset{c}{\wedge} U = V\} \cup \{(T_1, T_2) | T_1 \leqslant T_2\} \cup \{(T,V) | \exists U \; \overline{T} \overset{c}{\wedge} \overline{U} = \overline{V}\}$ is a type simulation relation.
2. $R_U = \{(V,U) | \exists T \; T \overset{c}{\wedge} U = V\} \cup \{(T_1, T_2) | T_1 \leqslant T_2\} \cup \{(U,V) | \exists T \; \overline{T} \overset{c}{\wedge} \overline{U} = \overline{V}\}$ is a type simulation relation.
3. $T \overset{c}{\wedge} U = V$ implies $\mathsf{greatest}(T, U, V)$.

We partially prove point 1, point 2 is similar. Relatively to point 1 we prove only the case for $\{(V,T) | \exists U \; T \overset{c}{\wedge} U = V\}$, the third relation is exactly the same while the second relation is straightforward. We proceed by inspection on $\mathsf{unfold}(T)$.

- if $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$ then by definition of $_\overset{c}{\wedge}_ = _$, $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ and letting $K = \{(i,j) | l_i = l'_j \text{ and } i \in I \text{ and } j \in J\}$, $\mathsf{unfold}(V) = \&\{l_i : V_{(i,j)}\}_{(i,j) \in K}$. The result follows since by definition $|K| \leq |I|$ and for all $(i,j) \in K$ there exists $i' \in I$ s.t. $l_i = l_{i'}$ (i.e. the labels of K are a subset of the labels in I) and $T_{i'} \overset{c}{\wedge} U_j = V_{(i,j)}$.
- if $\mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i \in I}$ then by definition of $_\overset{c}{\wedge}_ = _$, $\mathsf{unfold}(U) = \oplus\{l'_j : T'_j\}_{j \in J}$ and letting $K = \{(i,j) | l_i = l'_j \text{ and } i \in$

I and $j \in J\}$, $I' = I \setminus \mathsf{fst}(K)$, $J' = J \setminus \mathsf{snd}(K)$, $\mathsf{unfold}(V) = \oplus\{l_i : V_{(i,j)}\}_{(i,j)\in K}$;; $\oplus\{l_i : T_{\mu i}\}_{i\in I'}$;; $\oplus\{l_j : T'_{\mu j}\}_{j\in J'}$ and for all $i \in I'$, $T_{\mu i} \lesseqgtr T_i$ and for all $j \in J'$, $T_{\mu j} \lesseqgtr T_j$. In particular holds that $|I| \leq |K|$ and for all $i' \in I$ there exists $(i,j) \in K$ s.t. $l_{i'} = l_i$ and then either there exists $T_{\mu i}$ s.t. $T_{\mu i} \lesseqgtr T_{i'}$ or $T_{i'} \overset{c}{\wedge} U_i = V_{(i,j)}$.

- if $\mathsf{unfold}(T) = !(T_1).T'$ then by definition of $_\overset{c}{\wedge}_ = _$, $\mathsf{unfold}(U) = !(U_1).U'$ and $\mathsf{unfold}(V) = !(V_1).V'$. We have $\overline{T_1}\overset{c}{\wedge}\overline{U_1} = \overline{V_1}$ and $T'\overset{c}{\wedge}U' = V'$ which allow to conclude.
- Remaining cases are simpler.

To show point 3 we must prove that in fact minorant does his job showing that $R = \{(T,U,V) | V \leq T \text{ and } V \leq U\}$ is a minorant relation and both $R_T = \{(V,T) | \exists U \ \mathsf{minorant}(T,U,V)\}$ and $R_U = \{(V,U) | \exists T \ \mathsf{minorant}(T,U,V)\}$ are type simulation relations. We conclude by proving that $R_1 = \{(V_1,V) | T\overset{c}{\wedge}U = V \text{ and } \mathsf{minorant}(T,U,V_1)\}$ is a type simulation relation which in fact proves that $T\overset{c}{\wedge}U = V$ implies $\mathsf{greatest}(T,U,V)$. □

Example 3.20. The type U' in the Example 3.3 is not the meet of the two types T and V, since does not exist a meet relation which contains the tuple (T,V,U). In particular, the test fails due to the lack in U' of the label **quit** in a choice together with **play**. However U' is a minorant of T and V since it exists a minorant relation that contains (T,V,U').

Now we introduce another relation which contains all pairs of types such that the meet is defined.

Definition 3.21. A relation $R \subseteq TYPE \times TYPE$ is a *meet exists relation* if $(T,U) \in R$ implies the following conditions:

1. if $\mathsf{unfold}(T) = !(S_1,\ldots,S_n).T'$ then $\mathsf{unfold}(U) = !(S_1,\ldots,S_n).U'$ and $(T',U') \in R$
2. if $\mathsf{unfold}(T) = !(T_1).T'$ then $\mathsf{unfold}(U) = !(U_1).U'$, $(\overline{T_1},\overline{U_1}) \in R$ and $(T',U') \in R$
3. if $\mathsf{unfold}(T) = ?(S_1,\ldots,S_n).T'$ then $\mathsf{unfold}(U) = ?(S_1,\ldots,S_n).U'$ and $(T',U') \in R$

4. if $\mathsf{unfold}(T) = ?(T_1).T'$ then $\mathsf{unfold}(U) = ?(U_1).U'$ and $(T_1, U_1) \in R$ and $(T', U') \in R$
5. if $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ and $K = \{(i,j) \mid l_i = l'_j\}$ and $|K| \geq 1$ and $\forall i \forall j (i,j) \in K \Rightarrow (T_i, T'_j) \in R$
6. if $\mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(U) = \oplus\{l'_j : T'_j\}_{j \in J}$ and $K = \{(i,j) \mid l_i = l'_j\}$, $\forall i \forall j (i,j) \in K \Rightarrow (T_i, T'_j) \in R$
7. if $\mathsf{unfold}(T) = \mathtt{end}$ then $\mathsf{unfold}(U) = \mathtt{end}$

The co-inductive $\overset{c}{\wedge}$ relation is defined as $T \overset{c}{\wedge} U$ if there exists a *meet exists* relation R such that $(T, U) \in R$.

Simply, the previous definition ignores the shape of the meet, checking only the consistency of T and U. The next theorem formally prove the fact that the relation $\overset{c}{\wedge}$ is defined only if the meet of two types exists.

Theorem 3.22. *$T \overset{c}{\wedge} U$ iff $\exists V$ such that $T \overset{c}{\wedge} U = V$.*

Proof. $\Leftarrow$) It suffices to show that $R = \{(T, U) | \exists V \; T \overset{c}{\wedge} U = V\}$ is a meet exists relation. The proof follows directly from the two definitions.
$\Rightarrow$) It suffices to show that $R = \{(T, U, V) | T \overset{c}{\wedge} U$ and $V \leq T$ and $V \leq U$ and $\mathsf{greatest}(T, U, V)\}$ is a meet relation. All cases are simple. For example, if $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i \in I}$ then $\mathsf{unfold}(U) = \&\{l'_j : T'_j\}_{j \in J}$ and the result follows since $V = \&\{l_i : U_{(i,j)}\}_{(i,j) \in K}$ with $K = \{(i,j) \mid l_i = l'_j\}$ and $|K| \geq 1$ and $\forall i \forall j (i,j) \in K$, $U_{(i,j)} \leq T_i$ and $U_{(i,j)} \leq T'_j$ and $\mathsf{greatest}(T_i, T'_j, U_{(i,j)})$. □

One can prove that if there exists a lower bound then there exists also the greatest lower bound.

Lemma 3.23. *If $V \leq T$ and $V \leq U$ then there exists V' s.t. $T \overset{c}{\wedge} U = V'$.*

Proof. Simply we prove that $R = \{(T, U) | \exists \; V \; V \leq T$ and $V \leq U\}$ is a meet exists relation then we can conclude with the previous theorem. □

3.3.3 Algorithmic Meet

In this section we discuss our algorithm for computing the meet. First we present the algorithm for the membership checking of both relations $\overset{c}{\wedge}$ and $_\overset{c}{\wedge}_ = _$. The algorithmic sequents for $\overset{c}{\wedge}$ take the form $\Sigma \vdash T \overset{c}{\wedge} U$

where Σ represents the set of assumptions of the form $T' \overset{c}{\wedge} U'$ (later we will introduce the set of inverse assumptions Ξ with elements $T \overset{c}{\wedge} U$ too, see Figure 6). The inference rules in Figure 4 faithfully follows the definition of a meet exists relation but rules (A2-REC1), (A2-REC2) and (A2-ASSUMP) are used to handle the unfolding of a recursion with the usual convention that rule (A2-ASSUMP) should be applied if possible. The algorithmic sequents for $_ \overset{c}{\wedge} _ = _$ are $\Sigma \vdash T \overset{c}{\wedge} U = V$ where Σ represents the set of assumptions of the form $T' \overset{c}{\wedge} U' = V'$. (Note that we use the same symbol Σ used for algorithmic subtyping without any risk of clash) The inference rules in Figure 5 follows the definition of a meet relation but this time since we are handling a triple we have 3 different rules for recursion unfolding (rules (A3-REC1), (A3-REC2) and (A3-REC3)) and the relative assumption axiom (A3-ASSUMP). In rule (A3-CHOICE), the $\lesseqgtr$ relation can be verified with the set of inference rules in Figure 3.

The proof of soundness and completeness of the two algorithms with respect to the co-inductive definition and the proof of termination are similar to the proofs of Propositions 3.8 and 3.9, here the latter relation is only trivially extended with triple.

Moreover rule (A3-Choice) use the $\lesseqgtr$-relation which can be verified with the help of $\leq$ algorithm.

The problem in computing the meet of two infinite types is that the resulting structure can be different from the two types given in input. The relation $_ \overset{c}{\wedge} _ = _$ gives a hint on how to build the infinite type but we need a way to represent this type finitely by means of a regular infinite session type. As discussed in the introduction, it is not obvious that such representation exists, namely is the meet of two regular session types regular? In fact at this point we only know that if the meet has a finite representation (since the set of session types are the least fixed point of the generating function derived by the relative grammar) then we have a way of checking its goodness.

The idea of computing such representation comes from the algorithmic membership checking of the greatest fixed point for recursive regular types. Simply, the algorithm acts as follows: consider for example any generic type $\mu\alpha.\mathbf{A}.\mathbf{B}.\mathbf{C}.\alpha$ to be checked against another generic type $\mu\beta.\mathbf{D}.\mathbf{E}.\beta$, where $\mathbf{A}, ..., \mathbf{E}$ are generic actions. Then when unfolding recursion to perform the check, it is important to maintain a set of assumptions, so to prevent checking twice the same relation. After the first unfolding the algorithm matches $\mathbf{A}$ with $\mathbf{D}$, then $\mathbf{B}$ with $\mathbf{E}$, then $\mathbf{C}$ with

$$\text{(A2-END)}\quad \Sigma \vdash \texttt{end} \overset{c}{\wedge} \texttt{end} \qquad \text{(A2-ASSUMP)}\quad \Sigma, T \overset{c}{\wedge} U \vdash T \overset{c}{\wedge} U$$

$$\text{(A2-THROW)}\ \frac{\Sigma \vdash \overline{T_1} \overset{c}{\wedge} \overline{U_1} \quad \Sigma \vdash T \overset{c}{\wedge} U}{\Sigma \vdash !(T_1).T \overset{c}{\wedge} !(U_1).U} \qquad \text{(A2-OUT)}\ \frac{\Sigma \vdash T \overset{c}{\wedge} U}{\Sigma \vdash !(\tilde{S}).T \overset{c}{\wedge} !(\tilde{S}).U}$$

$$\text{(A2-CATCH)}\ \frac{\Sigma \vdash T_1 \overset{c}{\wedge} U_1 \quad \Sigma \vdash T \overset{c}{\wedge} U}{\Sigma \vdash ?(T_1).T \overset{c}{\wedge} ?(U_1).U} \qquad \text{(A2-IN)}\ \frac{\Sigma \vdash T \overset{c}{\wedge} U}{\Sigma \vdash ?(\tilde{S}).T \overset{c}{\wedge} ?(\tilde{S}).U}$$

$$\text{(A2-SELECT)}\ \frac{K = \{(i,j) \mid l_i = l'_j\} \quad |K| \geq 1 \quad \forall (i,j) \in K. \Sigma \vdash T_i \overset{c}{\wedge} T'_j}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \&\{l'_j : T'_j\}_{j \in J}}$$

$$\text{(A2-CHOICE)}\ \frac{K = \{(i,j) \mid l_i = l'_j\} \quad \forall (i,j) \in K. \Sigma \vdash T_i \overset{c}{\wedge} T'_j}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \oplus \{l'_j : T'_j\}_{j \in J}}$$

$$\text{(A2-REC1)}\ \frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U} \qquad \text{(A2-REC2)}\ \frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} U[^{\mu\alpha.U}/_{\alpha}]}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U}$$

Figure 4: Algorithmic membership checking for $\overset{c}{\wedge}$

D, then **A** with **E**, then **B** with **D** and finally **C** with **E**, for a total of six matches. After that, the algorithm ends since it re-encounters the original pair of types, already present in the assumptions. All checks made by the algorithm cover the entire range of possibilities for the first type and for the second type, this is also the desired behavior of the meet, namely the meet of the two previous types is composed by six components.

(A3-END)
$$\Sigma \vdash \mathtt{end} \overset{c}{\wedge} \mathtt{end} = \mathtt{end}$$

(A3-ASSUMP)
$$\Sigma, T \overset{c}{\wedge} U = V \vdash T \overset{c}{\wedge} U = V$$

(A3-OUT)
$$\frac{\Sigma \vdash T \overset{c}{\wedge} U = V}{\Sigma \vdash !(\tilde{S}).T \overset{c}{\wedge} !(\tilde{S}).U = !(\tilde{S}).V}$$

(A3-THROW)
$$\frac{\Sigma \vdash \overline{T_1} \overset{c}{\wedge} \overline{U_1} = \overline{V_1} \quad \Sigma \vdash T \overset{c}{\wedge} U = V}{\Sigma \vdash !(T_1).T \overset{c}{\wedge} !(U_1).U = !(V_1).V}$$

(A3-IN)
$$\frac{\Sigma \vdash T \overset{c}{\wedge} U = V}{\Sigma \vdash ?(\tilde{S}).T \overset{c}{\wedge} ?(\tilde{S}).U = ?(\tilde{S}).V}$$

(A3-CATCH)
$$\frac{\Sigma \vdash T_1 \overset{c}{\wedge} U_1 = V_1 \quad \Sigma \vdash T \overset{c}{\wedge} U = V}{\Sigma \vdash ?(T_1).T \overset{c}{\wedge} ?(U_1).U = ?(V_1).V}$$

(A3-SELECT)
$$\frac{K = \{(i,j) \mid l_i = l'_j\} \quad |K| \geq 1 \quad \forall (i,j) \in K.\Sigma \vdash T_i \overset{c}{\wedge} T'_j = U_{(i,j)}}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \&\{l'_j : T'_j\}_{j \in J} = \&\{l_i : U_{(i,j)}\}_{(i,j) \in K}}$$

(A3-CHOICE)
$$\frac{\begin{array}{cc} K = \{(i,j) \mid l_i = l'_j\}, I' = I \setminus \mathsf{fst}(K), & \forall (i,j) \in K.\Sigma \vdash T_i \overset{c}{\wedge} T'_j = U_{(i,j)} \\ J' = J \setminus \mathsf{snd}(K) & \forall i \in I'.T_{\mu i} \lesseqgtr T_i \quad \forall j \in J'.T'_{\mu j} \lesseqgtr T'_j \end{array}}{\begin{array}{cc} \Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \oplus \{l'_j : T'_j\}_{j \in J} = & \begin{array}{c} \oplus\{l_i : U_{(i,j)}\}_{(i,j) \in K} \;;\; \\ \oplus\{l_i : T_{\mu i}\}_{i \in I'} \;;\; \oplus\{l'_j : T'_{\mu j}\}_{j \in J'} \end{array} \end{array}}$$

(A3-REC1)
$$\frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U = V \vdash T[{}^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U = V}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U = V}$$

(A3-REC2)
$$\frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U = V \vdash T \overset{c}{\wedge} U[{}^{\mu\alpha.U}/_{\alpha}] = V}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U = V}$$

(A3-REC3)
$$\frac{\Sigma, T \overset{c}{\wedge} U = \mu\alpha.V \vdash T \overset{c}{\wedge} U = V[{}^{\mu\alpha.V}/_{\alpha}]}{\Sigma \vdash T \overset{c}{\wedge} U = \mu\alpha.V}$$

Figure 5: Algorithmic membership checking for $_\overset{c}{\wedge}_ = _$

The set of rules in Figure 6, with sequents of the form $\Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd Ʒ$ exploits this characteristic of the membership checking algorithms to compute the full range of possible behaviors of the meet. (Notice that the values contained in $Ʒ$ are not related to those in Σ: the symbol has just been chosen to remember the fact that this set of assumptions is built from top to bottom as opposed to Σ.) The set of inverse assumptions collects only the pairs of types which allow to apply the assumption axioms; rules (ALG-ASSUMP1) and (ALG-ASSUMP2). In these cases the algorithm returns (after the $\rightarrow$) a type variable whose name is dependent on the assumption (and on the input types as well). Notice that all possible assumptions are μ-guarded since they contain in either the first component or the second component a type starting with a recursion. In this manner, if before the unfolding of a recursion the pair is already present in the set of inverse assumptions the algorithm knows that this is the right time to place a recursion in the result. The recursion added by rules (ALG-REC1) and (ALG-REC2) is μ-guarded w.r.t. the same type variable that later allows application of axioms (ALG-ASSUMP1) and (ALG-ASSUMP2). If instead before the unfolding of a recursive type the pair is not present in the set of inverse assumptions, then the rules (ALG-SKIP1) and (ALG-SKIP2) are applied. The remaining rules simply compute locally the values of the meet, alike those for $_ \overset{c}{\wedge} _ = _$.

However, using a free type variable in axioms is a tricky operation and requires additional work to avoid capturing the wrong type variable with recursion. For this reason we need an auxiliary convention in order to guarantee that all type variables returned by (ALG-REC1) and (ALG-REC2) are different. We fix a standard naming for the variables to be introduced by (ALG-ASSUMP1) and (ALG-ASSUMP2), which are indeed determined by the exploited assumption present in $Ʒ$. Finally, we define the notation for $\mu\mathsf{var}$ to denote the set of variables that can arise from $Ʒ$: we let $\mu\mathsf{var}(\emptyset) = \emptyset$ and $\mu\mathsf{var}(T \overset{c}{\wedge} U, Ʒ) = \{\alpha_{T,U}\} \cup \mu\mathsf{var}(Ʒ)$.

The following lemma accounts two simple invariants of $\Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd Ʒ$.

Lemma 3.24. *If $\Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd Ʒ$ then the number of assumptions in $Ʒ$ is less then the number of assumptions in Σ and $\mathsf{fv}(V) = \mu\mathsf{var}(Ʒ)$.*

Proof. The proof is by straightforward rule induction on the rules for $\Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd Ʒ$. □

Now we prove the soundness and the completeness of the algorithm.

(ALG-END)
$$\Sigma \vdash \texttt{end} \overset{c}{\wedge} \texttt{end} \rightarrow \texttt{end} \rhd \emptyset$$

(ALG-ASSUMP1)
$$\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow \alpha_{\mu\alpha.T,U} \rhd \mu\alpha.T \overset{c}{\wedge} U$$

(ALG-ASSUMP2)
$$\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} \mu\alpha.U \rightarrow \alpha_{T,\mu\alpha.U} \rhd T \overset{c}{\wedge} \mu\alpha.U$$

(ALG-REC1)
$$\frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U \rightarrow V \rhd \mathcal{Z}, \mu\alpha.T \overset{c}{\wedge} U}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow \mu\alpha_{\mu\alpha.T,U}.V \rhd \mathcal{Z}}$$

(ALG-SKIP1)
$$\frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U \rightarrow V \rhd \mathcal{Z} \quad \mu\alpha.T \overset{c}{\wedge} U \notin \mathcal{Z}}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow V \rhd \mathcal{Z}}$$

(ALG-REC2)
$$\frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} U[^{\mu\alpha.U}/_{\alpha}] \rightarrow V \rhd \mathcal{Z}, T \overset{c}{\wedge} \mu\alpha.U}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U \rightarrow \mu\alpha_{T,\mu\alpha.U}.V \rhd \mathcal{Z}}$$

(ALG-SKIP2)
$$\frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} U[^{\mu\alpha.U}/_{\alpha}] \rightarrow V \rhd \mathcal{Z} \quad T \overset{c}{\wedge} \mu\alpha.U \notin \mathcal{Z}}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U \rightarrow V \rhd \mathcal{Z}}$$

(ALG-SELECT)
$$\frac{K = \{(i,j) \mid l_i = l'_j\} \quad |K| \geq 1 \quad \forall i \forall j (i,j) \in K. \Sigma \vdash T_i \overset{c}{\wedge} T'_j \rightarrow U_{(i,j)} \rhd \mathcal{Z}_{(i,j)}}{\Sigma \vdash \&\{l_i : T_i\}_{i\in I} \overset{c}{\wedge} \&\{l'_j : T'_j\}_{j \in J} \rightarrow \&\{l_i : U_{(i,j)}\}_{(i,j)\in K} \rhd \bigcup_{(i,j)\in K} \mathcal{Z}_{(i,j)}}$$

(ALG-CHOICE)
$$\frac{\begin{array}{c} K = \{(i,j) \mid l_i = l'_j\}, I' = I \setminus \mathsf{fst}(K), \\ J' = J \setminus \mathsf{snd}(K) \end{array} \qquad \forall (i,j) \in K. \Sigma \vdash T_i \overset{c}{\wedge} T'_j \rightarrow U_{(i,j) \rhd} \mathcal{Z}_{(i,j)}}{\begin{array}{c} \Sigma \vdash \oplus\{l_i : T_i\}_{i\in I} \overset{c}{\wedge} \\ \oplus\{l'_j : T'_j\}_{j\in J} \end{array} \rightarrow \begin{array}{l} \oplus\{l_i : U_{(i,j)}\}_{(i,j)\in K} \;;\; \\ \oplus\{l_i : T_i\}_{i \in I'} \;;\; \oplus\{l'_j : T'_j\}_{j\in J'} \end{array} \rhd \bigcup_{(i,j)\in K} \mathcal{Z}_{(i,j)}}$$

(ALG-OUT)
$$\frac{\Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd \mathcal{Z}}{\Sigma \vdash !(\tilde{S}).T \overset{c}{\wedge} !(\tilde{S}).U \rightarrow !(\tilde{S}).V \rhd \mathcal{Z}}$$

(ALG-THROW)
$$\frac{\Sigma \vdash \overline{T_1} \overset{c}{\wedge} \overline{U_1} \rightarrow \overline{V_1} \rhd \mathcal{Z}_1 \quad \Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd \mathcal{Z}}{\Sigma \vdash !(T_1).T \overset{c}{\wedge} !(U_1).U \rightarrow !(V_1).V \rhd \mathcal{Z}_1 \cup \mathcal{Z}}$$

(ALG-IN)
$$\frac{\Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd \mathcal{Z}}{\Sigma \vdash ?(\tilde{S}).T \overset{c}{\wedge} ?(\tilde{S}).U \rightarrow ?(\tilde{S}).V \rhd \mathcal{Z}}$$

(ALG-CATCH)
$$\frac{\Sigma \vdash T_1 \overset{c}{\wedge} U_1 \rightarrow V_1 \rhd \mathcal{Z}_1 \quad \Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd \mathcal{Z}}{\Sigma \vdash ?(T_1).T \overset{c}{\wedge} ?(U_1).U \rightarrow ?(V_1).V \rhd \mathcal{Z}_1 \cup \mathcal{Z}}$$

Figure 6: The algorithmic meet

All states of the algorithm we consider (with state we mean the instance of a particular conclusion) are of the form $\Sigma \vdash T\overset{c}{\wedge}U \to V \rhd \text{Ʒ}$. Moreover *we say that a state is completed if the set of inverse assumptions is the empty set*. To fix the correspondence between different proof trees, we define the function forget from a set of assumptions Σ (set of triples of types), to a set of assumptions Σ (set of pairs of types) as:

$$\mathsf{forget}(\Sigma) = \begin{cases} \mathsf{forget}(\emptyset) = \emptyset \\ \mathsf{forget}(\Sigma', T\overset{c}{\wedge}U = V) = \mathsf{forget}(\Sigma'), T\overset{c}{\wedge}U \end{cases}$$

where $\Sigma, T\overset{c}{\wedge}U = V$ (resp. $\Sigma, T\overset{c}{\wedge}U$, resp. $\text{Ʒ}, T\overset{c}{\wedge}U$) is $\Sigma \cup \{T\overset{c}{\wedge}U = V\}$ (resp. $\Sigma \cup \{T\overset{c}{\wedge}U\}$, resp. $\text{Ʒ} \cup \{T\overset{c}{\wedge}U\}$). The next proposition simply says that the algorithm in Figure 6 computes a correct intersection, all the additional hypotheses are used to give a correspondence between the algorithm in Figure 6 and the algorithm in Figure 5.

Proposition 3.25. *Let* $\Sigma \vdash T\overset{c}{\wedge}U \to V \rhd \text{Ʒ}$ *a state in the proof tree of a completed state with* $\{\alpha_{T_1,U_1}, \ldots, \alpha_{T_n,U_n}\} = \mu\mathsf{var}(\text{Ʒ})$. *Then there exists a tuple* $\mu\alpha_{T_1,U_1}.V'_1, \ldots, \mu\alpha_{T_n,U_n}.V'_n$ *s.t.* $\Sigma \vdash T\overset{c}{\wedge}U = V[^{\mu\alpha_{T_1,U_1}.V'_1}/_{\alpha_{T_1,U_1}}] \ldots [^{\mu\alpha_{T_n,U_n}.V'_n}/_{\alpha_{T_n,U_n}}]$ *where* $\mathsf{forget}(\Sigma) = \Sigma$ *and each of the substitution* $[^{\mu\alpha_{T_i,U_i}.V'_i}/_{\alpha_{T_i,U_i}}]$ *is introduced orderly by successive applications of either the rule* (ALG-REC1) *or the rule* (ALG-REC2) *with premise* $T_i \wedge U_i \to \mu\alpha_{T_i,U_i}.V_i$.

Proof. The proof is by induction on the height of the proof tree of completed state. Notice that we consider a state in the proof tree of a completed state then the order of the substitutions for the free variables in V is uniquely fixed by the considered proof tree.

The base cases are when the last applied rule is (ALG-END), and the statement trivially holds and when the last applied rule are (ALG-ASSUMP1) and (ALG-ASSUMP2). The last cases are both similar. For instance in the case of (ALG-ASSUMP1) we have $\mathsf{forget}(\Sigma), \mu\alpha.T\overset{c}{\wedge}U \vdash \mu\alpha.T\overset{c}{\wedge}U \to \alpha_{\mu\alpha.T,U} \rhd \mu\alpha.T\overset{c}{\wedge}U$ and we can conclude with rule (A3-ASSUMP) $\Sigma, \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V' \vdash T\overset{c}{\wedge}U = \alpha_{\mu\alpha.T,U}[^{\mu\alpha_{\mu\alpha.T,U}.V'}/_{\alpha_{\mu\alpha.T,U}}]$ for some V'. In the inductive cases when the last applied rule is

- (ALG-Rec1) we have $\mathsf{forget}(\Sigma) \vdash T[^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U \to V \rhd \text{Ʒ}', \mu\alpha.T\overset{c}{\wedge}U$ and by inductive hypothesis letting

$[\widetilde{\mu\alpha.V'}/\alpha] = [{}^{\mu\alpha_{T_1,U_1}.V_1'}/_{\alpha_{T_1,U_1}}]\dots[{}^{\mu\alpha T_{n-1},U_{n-1}.V_{n-1}'}/_{\alpha_{T_{n-1},U_{n-1}}}]$, $\Sigma \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = V[{}^{\mu\alpha_{\mu\alpha.T,U}.V_n'}/_{\alpha_{\mu\alpha.T,U}}][\widetilde{\mu\alpha.V'}/\alpha]$ where $\{\alpha_{T_1,U_1},\dots,\alpha_{T_{n-1},U_{n-1}}\} = \mu\mathsf{var}(\Xi')$. Since $[{}^{\mu\alpha_{\mu\alpha.T,U}.V_n'}/_{\alpha_{\mu\alpha.T,U}}]$ is introduced here by hypothesis from the other side we must have $\Sigma \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = V''$ with $V'' = V[\widetilde{\mu\alpha.V'}/\alpha][{}^{V[\widetilde{\mu\alpha.V'}/\alpha]}/_{\alpha_{\mu\alpha.T,U}}]$. The thesis follows applying (A3-Rec1) and (A3-Rec3) since we set $\Sigma = \Sigma', \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha.V'', T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = \mu\alpha.V''$. The following proof trees help understanding this case.

$$\dfrac{\dfrac{\dfrac{\mathsf{forget}(\Sigma''), \mu\alpha.T\overset{c}{\wedge}U \vdash \mu\alpha.T\overset{c}{\wedge}U \to \alpha_{\mu\alpha.T,U} \rhd \mu\alpha.T\overset{c}{\wedge}U}{\vdots}}{\mathsf{forget}(\Sigma), \mu\alpha.T\overset{c}{\wedge}U \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U \to V \rhd \mu\alpha.T\overset{c}{\wedge}U}}{\mathsf{forget}(\Sigma) \vdash \mu\alpha.T\overset{c}{\wedge}U \to \mu\alpha_{\mu\alpha.T,U}.V \rhd \emptyset}\ (\text{ALG-Rec1})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma'', \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V \vdash \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V}{\vdots}}{\Sigma', T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = V[{}^{\mu\alpha_{\mu\alpha.T,U}.V}/_{\alpha}]}}{\Sigma, \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V \vdash T[{}^{\mu\alpha.T}/_{\alpha}]\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V}\ (\text{A3-Rec3})}{\Sigma \vdash \mu\alpha.T\overset{c}{\wedge}U = \mu\alpha_{\mu\alpha.T,U}.V}\ (\text{A3-Rec1})$$

- (ALG-Throw) we have $\mathsf{forget}(\Sigma) \vdash \overline{T_1}\overset{c}{\wedge}\overline{U_1} \to \overline{V_1} \rhd \Xi_1$ and $\mathsf{forget}(\Sigma) \vdash \overline{T}\overset{c}{\wedge}\overline{U} \to \overline{V} \rhd \Xi$ and by induction $\Sigma \vdash \overline{T_1}\overset{c}{\wedge}\overline{U_1} = \overline{V_1}[{}^{\mu\alpha_{T_1,U_1}.V_1'}/_{\alpha_{T_1,U_1}}]\dots[{}^{\mu\alpha_{T_n,U_n}.V_n'}/_{\alpha_{T_n,U_n}}]$ where $\{\alpha_{T_1,U_1},\dots,\alpha_{T_n,U_n}\} = \mu\mathsf{var}(\Xi_1)$ and $\Sigma \vdash T\overset{c}{\wedge}U = V[{}^{\mu\alpha_{T_1',U_1'}.V_1''}/_{\alpha_{T_1',U_1'}}]\dots[{}^{\mu\alpha_{T_m',U_m'}.V_m''}/_{\alpha_{T_m',U_m'}}]$ where $\{\alpha_{T_1',U_1'},\dots,\alpha_{T_m',U_m'}\} = \mu\mathsf{var}(\Xi)$. By hypothesis we are inducting on a completed tree then it must exist a sequence of applications of either the rule (ALG-Rec1) or (ALG-Rec2) which consume all the substitutions and fixes their application ordering. Consequently we can conclude by applying (A3-Throw) $\Sigma \vdash !(T_1).T\overset{c}{\wedge}!(U_1).U = !(V_1).V[{}^{\mu\alpha_{T_1'',U_1''}.V_1'''}/_{\alpha_{T_1'',U_1''}}]\dots[{}^{\mu\alpha_{T_k'',U_m''}.V_k'''}/_{\alpha_{T_k'',U_k''}}]$ where $\{\alpha_{T_1'',U_1''},\dots,\alpha_{T_k'',U_k''}\} = \mu\mathsf{var}(\Xi \cup \Xi_1)$.

- remaining rules are similar e.g. (ALG-REC2), (ALG-SELECT) or follow directly by induction e.g. (ALG-SKIP1). In the case of rule (ALG-CHOICE) we use the reflexivity of $\leqslant$.

$\square$

The inverse of the previous Lemma is simpler since we only make assumptions on the existence of the meet (using the respective relation).

Proposition 3.26. *If* $\Sigma \vdash T \overset{c}{\wedge} U$ *then* $\exists Ʒ$ *and* U *s.t.* $\Sigma \vdash T \overset{c}{\wedge} U \rightarrow V \rhd Ʒ$*.*

Proof. The proof is by straightforward induction on the depth of the proof tree. The following is an example tree where the recursion is applied.

$$\dfrac{\dfrac{\dfrac{\Sigma', \mu\alpha.T \overset{c}{\wedge} U \vdash \mu\alpha.T \overset{c}{\wedge} U}{\vdots}}{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U}}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U} \ (\text{A2-REC1})$$

$$\dfrac{\dfrac{\dfrac{\Sigma', \mu\alpha.T \overset{c}{\wedge} U \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow \alpha_{\mu\alpha.T,U} \rhd \mu\alpha.T \overset{c}{\wedge} U}{\vdots}}{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U \rightarrow V \rhd \mu\alpha.T \overset{c}{\wedge} U, Ʒ}}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U \rightarrow \mu\alpha_{\mu\alpha.T,U}.V \rhd Ʒ} \ (\text{ALG-REC1})$$

$\square$

The final theorem states soundness and completeness of the algorithmic intersection.

Theorem 3.27. *If* $\emptyset \vdash T \overset{c}{\wedge} U \rightarrow V \rhd \emptyset$ *then* V *is the meet of* T *and* U*.*

Proof. By Proposition 3.25 we know that each value computed by the algorithm belongs to a meet relation. Notice that the proposition is applied since the set of inverse assumptions ($Ʒ$) is empty and thus a completed state. Again, since $Ʒ$ is empty, there exists a meet relation that contains (T, U, V). By the Proposition 3.26 we know that the algorithm exhibits a result V for each T and U for which the meet is defined and by Lemma 3.24, $Ʒ = \emptyset$ and $\mathsf{fv}(V) = \emptyset$. Then in order not to contradict Theorem 3.19 each value V in a meet relation, with T and U must be in $\leqslant$-relation with the value computed by the algorithm. $\square$

$$\dfrac{\dfrac{\dfrac{\dfrac{T\stackrel{c}{\wedge}U, \ldots \vdash T\stackrel{c}{\wedge}U \rightarrow \alpha_{T,U} \rhd T\stackrel{c}{\wedge}U}{cT\stackrel{c}{\wedge}U, \ldots \vdash \begin{array}{c}\&\{s1 : \alpha; s2 : \beta\}[^{T}/_{\alpha}][^{T_1}/_{\beta}]\stackrel{c}{\wedge} \\ \&\{s1 : \alpha; s3 : \beta\}[^{U}/_{\alpha}][^{U_1}/_{\beta}]\end{array} \rightarrow \&\{s1 : \alpha_{T,U}\} \rhd T\stackrel{c}{\wedge}U}\ (\text{ALG-ASSUMP1})}{\vdots}}{T\stackrel{c}{\wedge}U, \ldots \vdash \begin{array}{c}\&\{s1 : \alpha; s2 : \beta\}[^{T}/_{\alpha}][^{T_1}/_{\beta}]\stackrel{c}{\wedge} \\ \mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\}\end{array} \rightarrow \&\{s1 : \alpha_{T,U}\} \rhd T\stackrel{c}{\wedge}U}}{T\stackrel{c}{\wedge}U \vdash \begin{array}{c}\mu\beta.\&\{s1 : \alpha; s2 : \beta\}[^{T}/_{\alpha}]\stackrel{c}{\wedge} \\ \mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\}\end{array} \rightarrow \&\{s1 : \alpha_{T,U}\} \rhd T\stackrel{c}{\wedge}U}\ (\text{ALG-SKIP1})$$

$$\dfrac{T\stackrel{c}{\wedge}U \vdash \begin{array}{c}\mu\beta.\&\{s1 : \alpha; s2 : \beta\}[^{T}/_{\alpha}]\stackrel{c}{\wedge} \\ \mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\}\end{array} \rightarrow \&\{s1 : \alpha_{T,U}\} \rhd T\stackrel{c}{\wedge}U}{\emptyset \vdash \begin{array}{c}\mu\alpha.\mu\beta.\&\{s1 : \alpha; s2 : \beta\}\stackrel{c}{\wedge} \\ \mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\}\end{array} \rightarrow \mu\alpha_{T,U}.\&\{s1 : \alpha_{T,U}\} \rhd \emptyset}\ (\text{ALG-REC1})$$

Figure 7: Algorithmic meet at work

Example 3.28. We show how the algorithm computes the meet of $T = \mu\alpha.\mu\beta.\&\{s1 : \alpha; s2 : \beta\}$ and $U = \mu\alpha.\mu\beta.\&\{s1 : \alpha; s3 : \beta\}$. The intersection computed in Figure 7 is μ-equal to $\mu\alpha_{T,U}.\&\{s1 : \alpha_{T,U}\}$ with T_1 and U_1, in Figure, resulting from the unfolding and $\ldots$ in Figure, indicating any set of assumptions.

Notice that the given algorithm does not require backtracking (i.e. it is syntax directed) and the set of inference rules reported in Figure 6 constitutes in fact a real algorithm (applied from the conclusion to the premises). The algorithm is intended to input a triple Σ, T, U and return a pair V, Σ and if no rule conclusion matches the input triple the algorithm fails returning an error. When the algorithm fails it means that no intersection of two types exists, because Theorem 3.27 together with Theorem 3.19 logically imply the soundness and the completeness of our algorithm.

3.4 Types with free type variables

We now tackle the problem of computing the subtyping and the meet relations in presence of types with free type variables. First of all we introduce a new category of type variables ranged over β for sort type S and we continue using α for a type variable relative to a session type. The relative notion of free variables is defined as expected extended also to all type variables β. However we need to be more precise and identify also those free variables appearing as argument of either an input

$$\mathsf{fcv}(?(S_1,\ldots,S_n).T) = \mathsf{fcv}(T) \cup \bigcup_{i=1}^{n} \mathsf{fv}(S_i) \qquad \mathsf{fcv}(?(U).T) = \mathsf{fcv}(T) \cup \mathsf{fv}(U)$$
$$\mathsf{fcv}(!(S_1,\ldots,S_n).T) = \mathsf{fcv}(T) \cup \bigcup_{i=1}^{n} \mathsf{fv}(S_i) \qquad \mathsf{fcv}(!(U).T) = \mathsf{fcv}(T) \cup \mathsf{fv}(U)$$
$$\mathsf{fcv}(\&\{l_1 : T_1,\ldots,l_n : T_n\}) = \bigcup_{i=1}^{n} \mathsf{fcv}(T_i)$$
$$\mathsf{fcv}(\oplus\{l_1 : T_1,\ldots,l_n : T_n\}) = \bigcup_{i=1}^{n} \mathsf{fcv}(T_i)$$
$$\mathsf{fcv}(\mu\alpha.T) = \mathsf{fcv}(T) \setminus \{\alpha\} \qquad \mathsf{fcv}(\alpha) = \{\alpha\} \qquad \mathsf{fcv}(\beta) = \{\beta\} \qquad \mathsf{fcv}(\mathtt{end}) = \emptyset$$

Figure 8: The set of free communicated variables

or an output action. We call these variables free communicated variables written as fcv defined in Figure 8.

We anticipate that we will deal with type T such that $\mathsf{fv}(T) = \mathsf{fcv}(T)$ e.g. $!(\alpha).!(\beta)$ represents a type that sends both an unknown session α and an unknown value of type β and its free variables coincide with the free communicated variables.

We use σ and ρ to range over substitutions with domain containing both sort type variables and session type variables and with $\mathsf{dom}(\sigma)$ we indicate the set of both type variables and sort variables in the domain of σ, so for example $\sigma_1 = [^{\mathsf{end}}/\alpha][^{int}/\beta]$ is the substitution of α with $\mathtt{end}$ and of β with int and $\mathsf{dom}(\sigma) = \{\alpha, \beta\}$. The substitution application to a type T is pointed with σT thus for example $\sigma_1!(\beta).\alpha =!(int)$ holds. Often we extend substitution application to a constraint and to set of constraints as expected e.g. $\sigma(T_1 \leq T_2)$ corresponds to $\sigma T_1 \leq \sigma T_2$ and $\sigma\Sigma$ is the pointwise application of σ to each constraint in Σ. In the same way we indicate the substitution composition with the juxtaposition of substitutions, for example $\sigma\rho$ is the composition of ρ with σ. Since we use juxtaposition for both substitution composition and substitution application we fix that composition has higher priority than application so for example $\sigma\rho T$ is $(\sigma\rho)T$. Formally we can define composition of substitutions as:

Definition 3.29 (Composition of substitutions)**.** Let $\sigma = [^{T_1}/_{\alpha_1}]\ldots[^{T_n}/_{\alpha_n}][^{S_1}/_{\beta_1}]\ldots[^{S_m}/_{\beta_m}]$ and $\rho = [^{T'_1}/_{\alpha'_1}]\ldots[^{T'_{n'}}/_{\alpha'_{n'}}][^{S'_1}/_{\beta'_1}]\ldots[^{S'_{m'}}/_{\beta_{m'}}]$ s.t. $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\rho) = \emptyset$. The composition of σ and ρ, $\sigma\rho$ is the substitution $[^{\rho T_1}/_{\alpha_1}]\ldots[^{\rho T_n}/_{\alpha_n}][^{\rho S_1}/_{\beta_1}]\ldots[^{\rho S_n}/_{\beta_n}] \cup \rho$.

We employ the standard algorithm to find the most general unifier (66) and we use the notation $\sigma_{T=U}$ and $\sigma_{S_1=S_2}$ for the most general unifier s.t. respectively $\sigma T = \sigma U$ and $\sigma S_1 = \sigma S_2$. The idea is to rely on the unification algorithm to solve a constraint with the set of free variables equal to the set of free communicated variables. In order to do so we modify the definition of $\leq$ in cases of session input/output as follows:

$$\frac{T \leq U \in \Sigma}{\Sigma \vdash T \leq U \blacktriangleright \emptyset}\ (\text{AS}\mathcal{C}\text{-Assump}) \qquad \Sigma \vdash \texttt{end} \leq \texttt{end} \blacktriangleright \emptyset\ (\text{AS}\mathcal{C}\text{-End})$$

$$\Sigma \vdash \alpha \leq T \blacktriangleright \{\alpha = T\}\ (\text{AS}\mathcal{C}\text{-VarL}) \qquad \Sigma \vdash T \leq \alpha \blacktriangleright \{T = \alpha\}\ (\text{AS}\mathcal{C}\text{-VarR})$$

$$\frac{\Sigma, \mu\alpha.T \leq U \vdash T[^{\mu\alpha.T}/_{\alpha}] \leq U \blacktriangleright \mathcal{C}}{\Sigma \vdash \mu\alpha.T \leq U \blacktriangleright \mathcal{C}}\ (\text{AS}\mathcal{C}\text{-RecL}) \qquad \frac{\Sigma, T \leq \mu\alpha.U \vdash T \leq U[^{\mu\alpha.U}/_{\alpha}] \blacktriangleright \mathcal{C}}{\Sigma \vdash T \leq \mu\alpha.U \blacktriangleright \mathcal{C}}\ (\text{AS}\mathcal{C}\text{-RecR})$$

$$\frac{\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}}{\Sigma \vdash ?(\tilde{S}').T \leq ?(\tilde{S}'').U \blacktriangleright \{\tilde{S}' = \tilde{S}''\} \cup \mathcal{C}}\ (\text{AS}\mathcal{C}\text{-In}) \qquad \frac{\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}}{\Sigma \vdash !(\tilde{S}').T \leq !(\tilde{S}'').U \blacktriangleright \mathcal{C} \cup \{\tilde{S}' = \tilde{S}''\}}\ (\text{AS}\mathcal{C}\text{-Out})$$

$$\frac{\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}}{\Sigma \vdash ?(V).T \leq ?(V').U \blacktriangleright \{V = V'\} \cup \mathcal{C}}\ (\text{AS}\mathcal{C}\text{-Catch}) \qquad \frac{\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}}{\Sigma \vdash !(V).T \leq !(V').U \blacktriangleright \mathcal{C} \cup \{V = V'\}}\ (\text{AS}\mathcal{C}\text{-Throw})$$

$$\frac{|I| \leq |J| \quad \forall i \in I\ \exists j \in J \Rightarrow l_i = l'_j\ ,\ \Sigma \vdash T_i \leq T'_j \blacktriangleright \mathcal{C}_i}{\Sigma \vdash \&\{l_i : T_i\}_{i\in I} \leq \&\{l'_j : T'_j\}_{j \in J} \blacktriangleright \bigcup_{i\in I} \mathcal{C}_i}\ (\text{AS}\mathcal{C}\text{-Select})$$

$$\frac{|J| \leq |I| \quad \forall i \in I\ \exists j \in J \Rightarrow l_i = l'_j\ ,\ \Sigma \vdash T_i \leq T'_j \blacktriangleright \mathcal{C}_i}{\Sigma \vdash \oplus\{l_j : T_j\}_{j\in J} \leq \oplus\{l'_i : T'_i\}_{i \in I} \blacktriangleright \bigcup_{i\in I} \mathcal{C}_i}\ (\text{AS}\mathcal{C}\text{-Choice})$$

Figure 9: An algorithm to extract constraints for the syntactic unifier relative to the subtyping relation

- if $\mathsf{unfold}(T) = !(U).T'$ then $\mathsf{unfold}(V) = !(U).V'$ and $(T', V') \in R$
- if $\mathsf{unfold}(T) = ?(U).T'$ then $\mathsf{unfold}(V) = ?(U).V'$ and $(T', V') \in R$

From now on we use $\leq$ to indicate this new relation and with $\overset{c}{\wedge}$ and $_\overset{c}{\wedge}_ = _$ the straightforward adaptation of meet relations to this new definition of $\leq$. We think that removing the depth subtyping for sessions is not an issue in fact we recover it in rules for session delegation (see Figure 33).

In figure 9 we report the algorithm $\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}$ that extracts a set of unification constraints $\mathcal{C}$ whose the most general unifier is a substitution that we are going to call the syntactic unifier relative to the subtyping relation. The algorithm is simple, it faithfully follows the subtyping algorithm to collect a set of constraints. For example in rule (AS$\mathcal{C}$-Catch) we allow V and V' to be different as long as there exists a unifier that

solves $V = V'$; in rules (ASC-OUT) and (ASC-IN) we shorten the pointwise equality with $\tilde{S} = \tilde{S}'$.

As the next proposition shows the syntactic unifier computed by the algorithm is in fact the most general unifier of $T \leq U$ if the free variables of both T and U coincide with the free communicated variables (indicated with $\mathsf{fv}(T_1 \leq T_2) = \mathsf{fcv}(T_1 \leq T_2)$). Moreover we prove that if the free variables that are not also free communicated variables have unique occurrence then the syntactic unifier exists if and only if there exists a substitution ρ that satisfies the constraint. This second part of the proposition will be used to check the satisfiability of constraints generated by the session delegation which satisfy the hypothesis of linear occurrence of type variables.

Proposition 3.30. *Let $T_1 \leq T_2$ a constraint and $\Sigma \vdash T_1 \leq T_2 \blacktriangleright \mathcal{C}$. If σ is the most general unifier of $\mathcal{C}$ then for any $\rho = \sigma\sigma'$ we have $\rho\Sigma \vdash \rho T_1 \leq \rho T_2$. Vice versa*

1. *if $\mathsf{fv}(T_1 \leq T_2) = \mathsf{fcv}(T_1 \leq T_2)$ and $\rho\Sigma \vdash \rho T_1 \leq \rho T_2$ then there exists σ as the mgu of $\mathcal{C}$ and $\rho = \sigma\sigma'$ for some σ'.*

2. *if each occurrence of a free variable in the set $\mathsf{fv}(T_1 \leq T_2) \setminus \mathsf{fcv}(T_1 \leq T_2)$ is unique in $T_1 \leq T_2$ and $\rho\Sigma \vdash \rho T_1 \leq \rho T_2$ then there exist σ as the mgu of $\mathcal{C}$ and ρ_* s.t. $\rho_*\Sigma \vdash \rho_* T_1 \leq \rho_* T_2$ and $\rho_* = \sigma\sigma_*$ for some σ_* and $\mathsf{dom}(\rho) = \mathsf{dom}(\rho_*)$ and for all $\alpha \in \mathsf{dom}(\rho)$ if $\alpha \in \mathsf{fcv}(T_1 \leq T_2)$ then $\rho(\alpha) = \rho_*(\alpha)$.*

Proof. $\Rightarrow$) The proof is by induction on the derivation of $\Sigma \vdash T_1 \leq T_2 \blacktriangleright \mathcal{C}$ with case analysis on the last applied rule. In the base cases we have:

- (ASC-ASSUMP) and $\dfrac{T \leq U \in \Sigma}{\Sigma \vdash T \leq U \blacktriangleright \emptyset}$ (ASC-ASSUMP) and $\dfrac{\rho T \leq \rho U \in \rho\Sigma}{\rho\Sigma \vdash \rho T \leq \rho U}$ (AS-ASSUMP) for any ρ.

- (ASC-VARL) and (ASC-VARL) $\Sigma \vdash \alpha \leq T \blacktriangleright \{\alpha = T\}$ and the mgu is the substitution $\sigma = [^T/_\alpha]$ and (AS-VARL) $(\sigma\rho)\Sigma \vdash (\sigma\rho)\alpha \leq (\sigma\rho)T$ by reflexivity for any ρ. Notice that it cannot be the case that α is in the syntax tree of T otherwise the mgu is not defined.

- (ASC-VARR) similar to the previous case and (ASC-END) is trivial.

In the inductive cases we have:

- (AS$\mathcal{C}$-CATCH) and $\dfrac{\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}}{\Sigma \vdash ?(V).T \leq ?(V').U \blacktriangleright \{V = V'\} \cup \mathcal{C}}$ (AS$\mathcal{C}$-CATCH) and σ is the mgu of $\{V = V'\} \cup \mathcal{C}$. Let $\rho = \sigma\sigma'$ (for some σ'). Since σ solves $\{V = V'\} \cup \mathcal{C}$ we have that ρ solves $\mathcal{C}$ and $\rho V = \rho V'$. If we let σ_1 be the mgu of $\mathcal{C}$ then $\rho = \sigma_1\sigma''$. By inductive hypothesis $\rho\Sigma \vdash \rho T \leq \rho U$ (because $\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}$ and σ_1 is the mgu of $\mathcal{C}$ and $\rho = \sigma_1\sigma''$) hence $\rho\Sigma \vdash \rho?(V).T \leq \rho?(V').U$ by applying rule (AS-CATCH).

- (AS$\mathcal{C}$-SELECT) and $\dfrac{|I| \leq |J| \quad \forall i \in I\ \exists j \in J \Rightarrow l_i = l'_j\ ,\ \Sigma \vdash T_i \leq T'_j \blacktriangleright \mathcal{C}_i}{\Sigma \vdash \&\{l_i : T_i\}_{i\in I} \leq \&\{l'_j : T'_j\}_{j\in J} \blacktriangleright \bigcup_{i\in I} \mathcal{C}_i}$ (AS$\mathcal{C}$-SELECT) and σ is the mgu of $\bigcup_{i\in I} \mathcal{C}_i$. Since σ solves $\bigcup_{i\in I} \mathcal{C}_i$ we have for all i, σ solves $\mathcal{C}_i$. Let $\rho = \sigma\sigma'$ for some σ'. Then, $\forall i$ also ρ solves $\mathcal{C}_i$. If we let σ_{1i} be the mgu of $\mathcal{C}_i$ then $\rho = \sigma_{1i}\sigma''_i$ for all $i \in I$. By inductive hypothesis $\rho\Sigma \vdash \rho T_i \leq \rho T'_j$ hence $\rho\Sigma \vdash \rho\&\{l_i : T_i\}_{i\in I} \leq \rho\&\{l'_j : T'_j\}_{j\in J}$ by applying rule (AS-SELECT).

- the remaining case are either similar or follows directly by induction.

$\Leftarrow$ **Sub-Case 1**) Follows directly from the Sub-Case 2 since by hypothesis $\mathsf{fv}(T_1 \leq T_2) \setminus \mathsf{fcv}(T_1 \leq T_2) = \emptyset$ implies $\dot{\rho_*} = \dot{\rho}$.
$\Leftarrow$ **Sub-Case 2**) The proof is by induction on the derivation of $\rho\Sigma \vdash \rho T_1 \leq \rho T_2$ with case analysis on the last applied rule. In both rules (AS-ASSUMP) and (AS-END) the empty substitution holds for the mgu of the empty set.

In the inductive cases we have:

- every rule which has a premise of the form $\rho\alpha \leq \rho T$ or of the form $\rho T \leq \rho\alpha$. We discuss the first case the other is similar. We have $\rho\Sigma \vdash \rho\alpha \leq \rho T$ and $\rho = [^{T'}/_{\alpha}]\rho'$ for some T', ρ' and the rule (AS$\mathcal{C}$-VARL) is applied, then we take the substitution $\rho_* = [^{T}/_{\alpha}]\rho'$ to conclude since $[^{T}/_{\alpha}]$ is the mgu of $\{\alpha = T\}$.

- (AS-CATCH) and $\dfrac{\rho\Sigma \vdash \rho T \leq \rho U}{\rho\Sigma \vdash \rho?(V).T \leq \rho?(V').U}$ (AS-CATCH) and by induction $\Sigma \vdash T \leq U \blacktriangleright \mathcal{C}$ and $\rho_* = \sigma'\sigma'_*$ for some σ'_* and σ' is the mgu of $\mathcal{C}$. Now since $\rho V = \rho V'$ and since by hypothesis the variables in V and V' belong to set of free communicated variables then the information about the substitution are already those contained in σ'_* thus we can specialize the mgu taking a different composition $\rho_* = \sigma\sigma_*$ to conclude.

- (AS-SELECT) and $\dfrac{|I| \leq |J| \quad \forall i \in I\ \exists j \in J \Rightarrow l_i = l'_j\ ,\ \rho\Sigma \vdash \rho T_i \leq \rho T'_j}{\rho\Sigma \vdash \rho\&\{l_i : T_i\}_{i \in I} \leq \rho\&\{l'_j : T'_j\}_{j \in J}}$ (AS-SELECT) and by induction $\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \leq \&\{l'_j : T'_j\}_{j \in J} \blacktriangleright \mathcal{C}_i$ and $\rho_{*i} = \sigma_i\sigma_*$ for some σ_* and σ_i is the mgu of $\mathcal{C}_i$ and $i \in I$. By hypothesis ρ may differ from each ρ_{*i} in the value assigned to the free variables that are not free communicated variables and since each of these variables has an unique occurrence we can built the resulting ρ_* taking the substitution for the free variables from the respective branch they belong to.

□

In the same manner we define an algorithm $\Sigma \vdash T \overset{c}{\wedge} U \blacktriangleright \mathcal{C}$ reported in Figure 10 that extracts a set of unification constraints in order to find the syntactic unifier of a constraint of the form $T \overset{c}{\wedge} U$. The following proposition shows the property of this unifier.

Proposition 3.31. *Let T_1, T_2 two types from a constraint and $\Sigma \vdash T_1 \overset{c}{\wedge} T_2 \blacktriangleright \mathcal{C}$. If σ is the most general unifier of $\mathcal{C}$ then for any $\rho = \sigma\sigma'$ we have $\rho\Sigma \vdash \rho T_1 \overset{c}{\wedge} \rho T_2$. Vice versa*

1. *if $\mathsf{fv}(T_1 \overset{c}{\wedge} T_2) = \mathsf{fcv}(T_1 \overset{c}{\wedge} T_2)$ and $\rho\Sigma \vdash \rho T_1 \overset{c}{\wedge} \rho T_2$ then there exists σ as the mgu of $\mathcal{C}$ and $\rho = \sigma\sigma'$ for some σ'.*

2. *if each occurrence of a free variable in the set $\mathsf{fv}(T_1 \overset{c}{\wedge} T_2) \setminus \mathsf{fcv}(T_1 \overset{c}{\wedge} T_2)$ is unique in $T_1 \overset{c}{\wedge} T_2$ and $\rho\Sigma \vdash \rho T_1 \overset{c}{\wedge} \rho T_2$ then there exist σ as the mgu of $\mathcal{C}$ and ρ_* s.t. $\rho_*\Sigma \vdash \rho_* T_1 \leq \rho_* T_2$ and $\rho_* = \sigma\sigma_*$ for some σ_* and $\mathsf{dom}(\rho) = \mathsf{dom}(\rho_*)$ and for all $\alpha \in \mathsf{dom}(\rho)$ if $\alpha \in \mathsf{fcv}(T_1 \overset{c}{\wedge} T_2)$ then $\rho(\alpha) = \rho_*(\alpha)$.*

Proof. The proof is almost similar to the previous one. □

From now on when $\emptyset \vdash T_1 \leq T_2 \blacktriangleright \mathcal{C}$ we write $\sigma_{T_1 \leq T_2}$ for the most general unifier σ, if it exists, of $\mathcal{C}$ and when $\emptyset \vdash T_1 \overset{c}{\wedge} T_2 \blacktriangleright \mathcal{C}$ we write $\sigma_{T_1 \overset{c}{\wedge} T_2}$ for the most general unifier σ, if it exists, of $\mathcal{C}$.

3.4.1 Beyond syntactic unifiers

In the previous section we have introduced syntactic unifiers for both the subtyping algorithm and the meet exists relation and we have proved

$$\text{(A2}\mathcal{C}\text{-End)}\quad \Sigma \vdash \texttt{end} \overset{c}{\wedge} \texttt{end} \blacktriangleright \emptyset \qquad \text{(A2}\mathcal{C}\text{-Assump)}\quad \Sigma, T \overset{c}{\wedge} U \vdash T \overset{c}{\wedge} U \blacktriangleright \emptyset$$

$$\text{(A2}\mathcal{C}\text{-VarR)}\quad \Sigma \vdash T \overset{c}{\wedge} \alpha \blacktriangleright \{T = \alpha\} \qquad \text{(A2}\mathcal{C}\text{-VarL)}\quad \Sigma \vdash \alpha \overset{c}{\wedge} T \blacktriangleright \{\alpha = T\}$$

$$\text{(A2}\mathcal{C}\text{-Throw)}\quad \frac{\Sigma \vdash T \overset{c}{\wedge} U \blacktriangleright \mathcal{C}}{\Sigma \vdash !(V).T \overset{c}{\wedge} !(V').U \blacktriangleright \{V = V'\} \cup \mathcal{C}} \qquad \text{(A2}\mathcal{C}\text{-Out)}\quad \frac{\Sigma \vdash T \overset{c}{\wedge} U \blacktriangleright \mathcal{C}}{\Sigma \vdash !(\tilde{S}').T \overset{c}{\wedge} !(\tilde{S''}).U \blacktriangleright \{\tilde{S}' = \tilde{S''}\} \cup \mathcal{C}}$$

$$\text{(A2}\mathcal{C}\text{-Catch)}\quad \frac{\Sigma \vdash T \overset{c}{\wedge} U \blacktriangleright \mathcal{C}}{\Sigma \vdash ?(V).T \overset{c}{\wedge} ?(V').U \blacktriangleright \{V = V'\} \cup \mathcal{C}} \qquad \text{(A2}\mathcal{C}\text{-In)}\quad \frac{\Sigma \vdash T \overset{c}{\wedge} U \blacktriangleright \mathcal{C}}{\Sigma \vdash ?(\tilde{S}').T \overset{c}{\wedge} ?(\tilde{S''}).U \blacktriangleright \{\tilde{S}' = \tilde{S''}\} \cup \mathcal{C}}$$

$$\text{(A2}\mathcal{C}\text{-Select)}\quad \frac{K = \{(i,j) \mid l_i = l'_j\} \quad |K| \geq 1 \quad \forall (i,j) \in K.(\Sigma \vdash T_i \overset{c}{\wedge} T'_j \blacktriangleright \mathcal{C}_{(i,j)})}{\Sigma \vdash \&\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \&\{l'_j : T'_j\}_{j \in J} \blacktriangleright \bigcup_{(i,j) \in K} \mathcal{C}_{(i,j)}}$$

$$\text{(A2}\mathcal{C}\text{-Choice)}\quad \frac{K = \{(i,j) \mid l_i = l'_j\} \quad \forall (i,j) \in K.(\Sigma \vdash T_i \overset{c}{\wedge} T'_j \blacktriangleright \mathcal{C}_{(i,j)})}{\Sigma \vdash \oplus\{l_i : T_i\}_{i \in I} \overset{c}{\wedge} \oplus \{l'_j : T'_j\}_{j \in J} \blacktriangleright \bigcup_{(i,j) \in K} \mathcal{C}_{(i,j)}}$$

$$\text{(A2}\mathcal{C}\text{-Rec1)}\quad \frac{\Sigma, \mu\alpha.T \overset{c}{\wedge} U \vdash T[^{\mu\alpha.T}/_{\alpha}] \overset{c}{\wedge} U \blacktriangleright \mathcal{C}}{\Sigma \vdash \mu\alpha.T \overset{c}{\wedge} U \blacktriangleright \mathcal{C}} \qquad \text{(A2}\mathcal{C}\text{-Rec2)}\quad \frac{\Sigma, T \overset{c}{\wedge} \mu\alpha.U \vdash T \overset{c}{\wedge} U[^{\mu\alpha.U}/_{\alpha}] = V \blacktriangleright \mathcal{C}}{\Sigma \vdash T \overset{c}{\wedge} \mu\alpha.U \blacktriangleright \mathcal{C}}$$

Figure 10: An algorithm to extract constraints for the syntactic unifier relative to the meet exists relation

some useful properties in particular that in presence of types where free variables coincide with free communicated variables, syntactic unifiers are able to discover the most general substitution that satisfies the considered relation. Solving the problem in general when types present free variables is not easy. Here we discuss an algorithm to discover a substitution for a set of meet exists constraints which is important since also solving a subtyping constraint involves discovering a substitution for a set of meet exists constraints.

First of all consider the algorithm in Figure 10 with these rules:

$$\text{(A2}\mathcal{C}\text{-VarR)}\quad \Sigma \vdash T \overset{c}{\wedge} \alpha \blacktriangleright\!\!\blacktriangleright \{T \overset{c}{\wedge} \alpha\} \qquad \text{(A2}\mathcal{C}\text{-VarL)}\quad \Sigma \vdash \alpha \overset{c}{\wedge} T \blacktriangleright\!\!\blacktriangleright \{\alpha \overset{c}{\wedge} T\}$$

localsolve

$\underline{\text{localsolve}}(\mathcal{C})$

1. given $\mathcal{C}$ of the form $T = T_1, \mathcal{C}'$ (resp. $S = S_1, \mathcal{C}'$) then σ=$\underline{\text{localsolve}}(\sigma_{T=T_1}\mathcal{C}')$ (resp. $\underline{\text{localsolve}}(\sigma_{S=S_1}\mathcal{C}')$) and return $\sigma_{T=T_1}\sigma$ (resp. $\sigma_{S=S_1}\sigma$)

2. given $\mathcal{C}$ of the form $\alpha \overset{c}{\wedge} T, \mathcal{C}'$ (resp. $U \overset{c}{\wedge} \alpha, \mathcal{C}'$) then σ=$\underline{\text{localsolve}}([^T/_\alpha]\mathcal{C}')$ (resp. $\underline{\text{localsolve}}([^U/_\alpha]\mathcal{C}')$) and return $[^T/_\alpha]\sigma$ (resp. $[^U/_\alpha]\sigma$)

3. given $\mathcal{C}$ of the form $T \overset{c}{\wedge} U, \mathcal{C}'$ with $T \neq \alpha$ and $U \neq \alpha$ compute $\emptyset \vdash T \overset{c}{\wedge} U \blacktriangleright\!\!\blacktriangleright \mathcal{C}''$ then return $\underline{\text{localsolve}}(\mathcal{C}' \cup \mathcal{C}'')$.

4. if $\mathcal{C} = \emptyset$ return the empty substitution.

Figure 11: localsolve algorithm

It is obvious that a solution to the set of constraints generated by $\blacktriangleright\!\!\blacktriangleright$ holds if and only if the substitution satisfies the considered relation.

Lemma 3.32. *Let* $\Sigma \vdash T_1 \overset{c}{\wedge} T_2 \blacktriangleright\!\!\blacktriangleright \mathcal{C}$. $\sigma \vDash \mathcal{C}$ *iff* $\sigma\Sigma \vdash \sigma T_1 \overset{c}{\wedge} \sigma T_2$.

Proof. By straightforward induction on either the derivation of $\Sigma \vdash T_1 \leq T_2 \blacktriangleright\!\!\blacktriangleright \mathcal{C}$ or the derivation of $\sigma\Sigma \vdash \sigma T_1 \leq \sigma T_2$. $\square$

However we want to solve a set of such constraints and now we give, in Figure 11, an algorithm that discovers and returns a solving substitution. For ease of notation in the algorithm we use $\mathcal{C}, \mathcal{C}'$ as an abbreviation of $\mathcal{C} \cup \mathcal{C}'$ and when either $\mathcal{C}$ or $\mathcal{C}'$ is given specifying the list of elements we omit the usual usage of curly brackets. In line 1 the algorithm solves the unification constraints using the most general unifier for both sort types S and session types T. In line 2 we approximate α with T (resp. U) exploiting the symmetry of the meet. In the penultimate line we generate the set of constraints for the meet of T and U. Obviously we lost the completeness of the algorithm in line 2 but this algorithm is enough under the assumption of unique occurrence of type variables (see Proposition 5.30). We now prove the soundness of the algorithm, we use the notation $\sigma \vDash \mathcal{C}$ to mean that $\sigma\mathcal{C}$ holds. This notation comes together with this useful lemma, it allows to switch substitutions from the right to $\vDash$ to its left.

Lemma 3.33 (Switching Lemma). *Let σ, σ' two substitutions s.t.* $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \emptyset$. $\sigma \vDash \sigma'\mathcal{C}$ *iff* $\sigma'\sigma \vDash \mathcal{C}$.

Proof. The proof follows directly by the definition of substitution composition and the definition of $\sigma \vDash \mathcal{C}$. □

The function $\mathsf{var}(\mathcal{C})$ returns the set of both sort and session type variables in $\mathcal{C}$. The condition on the domain of σ says that we want a minimal substitution w.r.t. to the free variables in the set of constraints, moreover this allows the application of Lemma 3.33.

Lemma 3.34 (localsolve soundness). *If σ=localsolve($\mathcal{C}$) then $\sigma \vDash \mathcal{C}$ and* $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(\mathcal{C})$.

Proof. We first define a measure of termination for localsolve then we use it to induct on the recursive structure of the algorithm. The well founded order we are going to define is $\mathcal{C} < \mathcal{C}'$ if $\mathcal{C}$ has less constraints of the form $T \overset{c}{\wedge} U$ where T and U are different from a type variables or if such constraints are equal in both $\mathcal{C}$ and $\mathcal{C}'$ then $\mathcal{C}$ has less constraints of the form $T \overset{c}{\wedge} U$ where either T or U is a type variable otherwise if the equality holds for the previous conditions then $|\mathcal{C}| < |\mathcal{C}'|$. We now proceed to prove lemma by induction on σ=localsolve($\mathcal{C}$) by case analysis on the last applied line. In the base case line 4 is applied and the empty substitution solves the empty set of constraints. In the inductive case when the last applied line is:

- line 1 and σ =localsolve($\sigma_{T=T_1}\mathcal{C}'$) implies by induction $\sigma \vDash \sigma_{T=T_1}\mathcal{C}'$ and $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(\sigma_{T=T_1}\mathcal{C}')$. By definition $\sigma \vDash \sigma_{T=T_1}(\mathcal{C}' \cup \{T = T_1\})$ and by Lemma 3.33 (since $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma_{T=T_1}) = \emptyset$) $\sigma_{T=T_1}\sigma \vDash \mathcal{C}' \cup \{T = T_1\}$ which concludes. The case where an unification constraint of the form $S = S_1$ is solved is similar.

- line 2 and σ =localsolve($[^T/_\alpha]\mathcal{C}'$) implies by induction $\sigma \vDash [^T/_\alpha]\mathcal{C}'$. Since the intersection is idempotent it is the case that $[^T/_\alpha]\sigma(T \overset{c}{\wedge} \alpha)$. We can conclude since $[^T/_\alpha]\sigma \vDash \mathcal{C}' \cup \{T \overset{c}{\wedge} \alpha\}$.

- line 3 and σ=localsolve($\mathcal{C}' \cup \mathcal{C}''$) and $\mathcal{C}''$ s.t. $\emptyset \vdash T \overset{c}{\wedge} U \twoheadrightarrow \mathcal{C}''$ and by induction $\sigma \vDash \mathcal{C}' \cup \mathcal{C}''$. By Lemma 3.32, $\sigma \vDash T \overset{c}{\wedge} U$ holds and we can conclude with $\sigma \vDash \mathcal{C}' \cup \mathcal{C}'' \cup \{T \overset{c}{\wedge} U\}$.

□

Let us now make some examples, in particular the first one shows how to solve a subtyping constraint by means of localsolve.

Example 3.35. We want to check if there exists some substitution that satisfies the following inequality.

$$\begin{array}{c}\oplus\{l_1 : \alpha, l_2 : \alpha, l_3 : \alpha, l_4 : \alpha_1\} \\ \leq \\ \oplus\{l_1 : \oplus\{ll_1 : \alpha_2, ll_2 : \alpha_3\}, l_2 : \oplus\{ll_1 : \alpha_3, ll_3 : \alpha_4\}, l_4 : \texttt{end}\}\end{array}$$

It is simple to extract a set of constraints from a subtype constraint (with a similar algorithm to $\twoheadrightarrow$), as we done in the following, in order to exploit localsolve.

$$\begin{array}{l}\alpha \leq \oplus\{ll_1 : \alpha_2, ll_2 : \alpha_3\} \quad \alpha \leq \oplus\{ll_1 : \alpha_3, ll_3 : \alpha_4\} \quad \alpha_1 \leq \texttt{end} \\ \alpha \leq \oplus\{ll_1 : \alpha_2, ll_2 : \alpha_3\} \quad \alpha \leq \oplus\{ll_1 : \alpha_3, ll_3 : \alpha_4\} \\ \underline{\text{localsolve}}(\oplus\{ll_1 : \alpha_2, ll_2 : \alpha_3\} \overset{c}{\wedge} \oplus \{ll_1 : \alpha_3, ll_3 : \alpha_4\}) \\ \quad \alpha_2 \overset{c}{\wedge} \alpha_3 \\ \quad \mathit{success}\end{array}$$

In the first line we safely remove $\alpha_1 \leq \texttt{end}$ since it does not appear elsewhere, in the second line we remain with two constraints relative to α then we use localsolve, to check the existence of the intersection.

Example 3.36. Consider someone give us this specification of α without specifying something else. We want to check locally if there exists someone that is able to behave dually.

$$\begin{array}{l}\alpha \leq \oplus\{l_1 : \alpha_1, l_2 : \alpha_1, l_3 : \alpha_1, l_4 : \alpha_2\} \\ \alpha \leq \oplus\{ \begin{array}{c} l_1 : \oplus\{ll_1 : \&\{lll_1 : \alpha_1\}, ll_2 : \alpha_4\}, \\ l_2 : \oplus\{ll_1 : \&\{lll_2 : \alpha_1\}, ll_2 : \alpha_2\}, l_4 : \texttt{end} \end{array} \}\end{array}$$

We proceed using $\emptyset \vdash T_1 \overset{c}{\wedge} T_2 \twoheadrightarrow \mathcal{C}$ (where T_1 and T_2 are the two types at the right of α) to extract the set of constraints and then localsolve to solve it.

$$\begin{array}{ll} 2 & \begin{array}{c} \alpha_2 \overset{c}{\wedge} \texttt{end} \qquad \alpha_1 \overset{c}{\wedge} \oplus \{ll_1 : \&\{lll_1 : \alpha_1\}, ll_2 : \alpha_4\} \\ \alpha_1 \overset{c}{\wedge} \oplus \{ll_1 : \&\{lll_2 : \alpha_1\}, ll_2 : \alpha_2\} \end{array} \\ 2 & \alpha_1 \overset{c}{\wedge} \oplus \{ll_1 : \&\{lll_1 : \alpha_1\}, ll_2 : \alpha_4\} \quad \alpha_1 \overset{c}{\wedge} \oplus \{ll_1 : \&\{lll_2 : \alpha_1\}, ll_2 : \texttt{end}\} \\ 3 & \begin{array}{c} \oplus\{ll_1 : \&\{lll_1 : \alpha_1\}, ll_2 : \alpha_4\} \overset{c}{\wedge} \\ \oplus\{ll_1 : \&\{lll_2 : \oplus\{ll_1 : \&\{lll_2 : \alpha_1\}, ll_2 : \texttt{end}\}\}, ll_2 : \texttt{end}\} \end{array} \end{array}$$

which fails in the last step.

In spite of the <u>localsolve</u> failure in the previous example the two constraints are satisfiable. Consider for example the substitution of α_1 with $\oplus\{ll_3 : \texttt{end}\}$, α_2 and α_4 both with end and consequently α with $\oplus\{l_1 : \oplus\{ll_3 : \texttt{end}\}, l_2 : \oplus\{ll_3 : \texttt{end}\}, l_3 : \oplus\{ll_3 : \texttt{end}\}, l_4 : \texttt{end}\}$. The problem is in the line 2 of <u>localsolve</u> which approximates the behavior of α in $\alpha \overset{c}{\wedge} T$ with T itself. We have been smarter than the algorithm, choosing the right substitution with a *fresh* label ll_3 that does not lead to a contradiction (the fact that does not exist the intersection of $\&\{lll_1 : \alpha_1\}$ and $\&\{lll_2 : \alpha_1\}$). What we are going to do now is to refine line 2 of <u>localsolve</u> to achieve the completeness.

2a given $\mathcal{C}$ of the form $\alpha \overset{c}{\wedge} T, \mathcal{C}'$ then:

1 if $\mathsf{unfold}(T) = ?(\tilde{S}).T'$ (resp. $?(V).T$) then σ=<u>localsolve</u>($[^{?(\tilde{S}).\alpha'}/_{\alpha}]\mathcal{C}$) (resp. σ=<u>localsolve</u>($[^{?(V).\alpha'}/_{\alpha}]\mathcal{C}$) with α' fresh and return $[^{?(\tilde{S}).\alpha'}/_{\alpha}]\sigma$ (resp. $[^{?(V).\alpha'}/_{\alpha}]\sigma$)

2 if $\mathsf{unfold}(T) = !(\tilde{S}).T'$ (resp. $!(V).T$) then σ=<u>localsolve</u>($[^{!(\tilde{S}).\alpha'}/_{\alpha}]\mathcal{C}$) (resp. <u>localsolve</u>($[^{!(V).\alpha'}/_{\alpha}]\mathcal{C}$)) with α' fresh and return $[^{!(\tilde{S}).\alpha'}/_{\alpha}]\sigma$ (resp. $[^{!(V).\alpha'}/_{\alpha}]\sigma$)

3 if $\mathsf{unfold}(T) = \oplus\{l_i : T_i\}_{i\in I}$ then generate a new fresh label l and σ=<u>localsolve</u>($[^{\oplus\{l:\texttt{end}\}}/_{\alpha}]\mathcal{C}'$) and return $[^{\oplus\{l:\texttt{end}\}}/_{\alpha}]\sigma$

4 if $\mathsf{unfold}(T) = \&\{l_i : T_i\}_{i\in I}$ then select a label l_j try σ=<u>localsolve</u>($[^{\&\{l_j:T_j\}}/_{\alpha}]\mathcal{C}\}$) if it fails select another label and retry until either all labels are chosen or the call succeeds and return $[^{\&\{l_j:T_j\}}/_{\alpha}]\sigma$.

5 if $\mathsf{unfold}(T) = \alpha'$ or $\mathsf{unfold}(T) = \texttt{end}$ then σ=<u>localsolve</u>($[^{T}/_{\alpha}]\mathcal{C}\}$) and return $[^{T}/_{\alpha}]\sigma$

2b given $\mathcal{C}$ of the form $U \overset{c}{\wedge} \alpha, \mathcal{C}'$ we proceed in a similar manner as before.

The idea behind this refinement lays on the fact that we specialize each time one operator without instantiating too much in such a manner to lose the completeness. Lines for input/output prefixes (lines 2a-1, 2b-1

and 2a-2,2b-2) instantiate α with the same prefix and a fresh variable in the continuation. Lines relative to the external choice (2a-3,2b-3) creates a new fresh branch in such a manner to not interfere with the already available labels. Internal choice requires care, according to rule (A2-SELECT) (since $|K| \geq 1$) the instantiated choice must have at the least one common branch with the other choice then we backtrack until a compatible branch is found. Even if we do not formally prove the following claim it should hold.

Claim. *σ=localsolve($\mathcal{C}$) iff there exists ρ and $\rho \vDash \mathcal{C}$.*

Let us now show some examples of how the algorithm works.

Example 3.37. Take again the constraints from example 3.36 we have:

$$
\begin{array}{ll}
2a-5 & \alpha_2 \overset{c}{\wedge} \mathtt{end} \quad \alpha_1 \overset{c}{\wedge} \oplus\{ll_1 : \&\{lll_1 : \alpha_1\}, ll_2 : \alpha_4\} \\
 & \alpha_1 \overset{c}{\wedge} \oplus\{ll_1 : \&\{lll_2 : \alpha_1\}, ll_2 : \alpha_2\} \\
2a-3 & \alpha_1 \overset{c}{\wedge} \oplus\{ll_1 : \&\{lll_1 : \alpha_1\}, ll_2 : \alpha_4\} \\
 & \alpha_1 \overset{c}{\wedge} \oplus\{ll_1 : \&\{lll_2 : \alpha_1\}, ll_2 : \mathtt{end}\} \\
3 & \oplus\{ll_3 : \mathtt{end}\} \overset{c}{\wedge} \oplus\{ll_1 : \&\{lll_2 : \oplus\{ll_3 : \mathtt{end}\}\}, ll_2 : \mathtt{end}\} \\
4 & \textit{success}
\end{array}
$$

Example 3.38. Consider we have a set of meet constraints and we want to check if there exists some substitution that satisfies them all together, we can run localsolve directly on the set of constraints. For instance takes the constraints in the first line we have:

$$
\begin{array}{l}
\alpha \overset{c}{\wedge} \&\{l :!(int), l_1 : \mathtt{end}\} \quad \alpha \overset{c}{\wedge} \&\{l :?(int), l_2 : \mathtt{end}\} \quad \alpha \overset{c}{\wedge} \alpha_1 \\
\star \quad \alpha_1 \overset{c}{\wedge} \&\{l :!(int), l_1 : \mathtt{end}\} \quad \alpha_1 \overset{c}{\wedge} \&\{l :?(int), l_2 : \mathtt{end}\} \\
\quad \&\{l : \alpha_2\} \overset{c}{\wedge} \&\{l :!(int), l_1 : \mathtt{end}\} \quad \&\{l : \alpha_2\} \overset{c}{\wedge} \&\{l :?(int), l_2 : \mathtt{end}\} \\
\quad \alpha_2 \overset{c}{\wedge} !(int) \quad \&\{l : \alpha_2\} \overset{c}{\wedge} \&\{l :?(int), l_2 : \mathtt{end}\} \\
\quad \&\{l :!(int)\} \overset{c}{\wedge} \&\{l :?(int), l_2 : \mathtt{end}\} \\
\star \quad \alpha_1 \overset{c}{\wedge} \&\{l :!(int), l_1 : \mathtt{end}\} \quad \alpha_1 \overset{c}{\wedge} \&\{l :?(int), l_2 : \mathtt{end}\} \\
\quad \&\{l_1 : \alpha_3\} \overset{c}{\wedge} \&\{l :!(int), l_1 : \mathtt{end}\} \quad \&\{l_1 : \alpha_3\} \overset{c}{\wedge} \&\{l :?(int), l_2 : \mathtt{end}\}
\end{array}
$$

In particular in the starred lines we backtrack trying to find the correct branch and it turns out that the function fails after trying both branches l and l_1.

3.5 Concluding remarks on the session types framework

In this chapter we have studied the subtyping preorder introduced by Gay and Hole in (33). Taking the definition as originally proposed we have introduced the greatest lower bound of two types. We also have proposed a co-inductive characterization of the meet and we have proved that these two characterizations are equivalent. However, the co-inductive relation is useful to prove properties of the intersection for instance we proved that if there exists a lower bound between two types then there exists the greatest lower bound too. The *meet exist* relation $\overset{c}{\wedge}$ instead, contains all pairs such that the greatest lower bound exists.

Next we have tackled the problem of algorithmically compute the intersection. The co-inductive definition of these two relations gives a hint on how to locally compute the meet but it does not exhibit the representation as a finite regular session type. A further step is that for co-inductive relations the membership checking algorithms based on set of assumptions, is standard (2; 31; 50). The existence of these algorithms says that if the meet is finite then we have a way of checking its appropriateness. These algorithms proceed by exploring all the possible combinations of sub-elements among the input elements until an already explored combination is found and this is exactly the behavior of the meet. Our algorithm $\emptyset \vdash T \overset{c}{\wedge} U \ \rightarrow \ V \triangleright \emptyset$ is built on this key observation. To prove its soundness we have showed that the relation $_ \overset{c}{\wedge} _ = _$ is able to simulate all the input triples in the algorithm. Vice versa the algorithm is capable to exhibit a result for each pair of type that has the greatest lower bound. The soundness result follows from the uniqueness, up to $\leqslant$, of the greatest lower bound.

We have concluded this chapter studying both the subtyping relation and the meet exist relation among types with free variables. We have two kinds of type variables, one kind for sort types and another one for session types respectively. What we want to do is to compute (if it exists) a substitution for the variables of each type such that the substituted types satisfies the considered relation. Since we are interested to the most general such substitution we modify both the subtyping relation and the meet relation to not allow the exploration in depth, that is we require the syntactic equality of sent/receviced sessions. Thanks to this mild modification (since we recover the depth session subtyping modifying the

typing rules for session delegation) we compute the syntactic unifier resulting as the solution of a set of unification constraints obtained miming the algorithmic behavior of each relation. As expected when given types such that free variables appear only in the communicated types, the syntactic unifier is in fact the most general unifier. Next, we have focused our attention to solve the problem of finding the existence of a substitution in the general case where types contain also free variables. We have proved that if each occurrence of a free variable is unique the syntactic unifier exists if and only if there exists a substitution that satisfies the considered constraint. In alternative we have proposed the localsolve algorithm which inputs a set of meet exists constraints and returns a solving substitution. Solving the meet exists constraints is an important fact since also a subtyping constraint can be reduced to a set of meet exists constraint. We have proposed a first version of localsolve which is only correct and a refined one. For this refined version we conjecture that it succeeds if and only if there exists a solving substitution, unfortunately proving it is an issue due also to the backtracking behaviors in the rule for external choice. In fact the algorithm in presence of an external choice tries a branch per time until either the a recursive call succeeds or it fails since all branches are explored.

Chapter 4

CaSPiS for Session Types

4.1 Introduction

In this chapter we introduce a simple language to handle sessions. This language is somewhat a more disciplinated version of the full calculus introduced next, but since it allows reasoning only on two sessions per time, the proofs and the statements of the various theorems are certainly much immediate. The core language is inspired to CaSPiS (8; 9) but it is extended with constructs for intra-session communications and at the same time it is limited removing pattern matching. One can think of this language as a typed variant of CaSPiS with session types, which we call CaSPiS for Session Types or CST for short. As in the original proposal in CST communications are intended within the current session and within the parent session. For this reason communication primitives are not annotated with the session subject since this subject is implicit from the syntactic context. The resulting type system, instead of being endowed with the linear environment Δ, has a couple of types (without session subject) relative to the current and to the parent session, that is, besides standard type environments, type judgments for processes P have the form $P : T; U$.

When we started working with these judgments we noticed that the entire framework is suitable to explore some naive intuitions about recursive processes. For example given that the process P is typed as $P : T; U$ is it true that `rec` $X.P'$ where P' is obtained from P by replacing trailing $\mathbf{0}$ with X is typed as $P : \mu\alpha.T'; \mu\alpha.U'$ where T' and U' are obtained

from T and U respectively by replacing the trailing occurrences of `end` with α? We start introducing a standard non syntax directed type system and then we build a *set based* type system that infers recursive behaviors based on the intuition above. We found that this basic intuition requires a lot of mathematical efforts to prove its validity. For instance we prove that if $T \leq U$ then it holds that infinite sequences of T (obtained by means of the μ operator) are in subtyping relation with infinite sequences of U. In addition, the set based type system uses sets to record the type of multiple branches in a process, for example the type of an if-then-else is the set resulting from the union of two branches.

We prove that the syntax directed type system is correct with respect to the non syntax directed version, that is it produces valid judgments and then we propose an algorithm to extract constraints and another algorithm to solve these constraints. In few words we partition the non-determinism of the initial type system into two blocks: a syntax directed type system from which we generate some constraints and an algorithm to solve these constraints. As we shall see, using sets for multiplexing types of branches and inferring on the fly recursive behaviors, we push all efforts towards the type system. The resulting constraint solver algorithm is very simple and elegant. Such algorithm has a complexity that is quadratic with respect to the set of constraints and the set of constraints have as many elements as the number of service invocations and service declarations in the input process. Unfortunately, this naive type system is not complete with respect to the original type system. At the end we propose another syntax directed type system which is also complete. This time the effort is split among the two blocks but the algorithm of constraint resolution has an exponential growth as well as an increment of the number of generated constraints.

Background. CaSPiS was proposed in (9) built on the proposal of SCC (8). We built CST on the top of CaSPiS adding the communication constructs as proposed in (37) (see Section 6.2.1 for a brief introduction to CaSPiS). We also give the operational semantics by means of reductions much closer to (8). The idea of typing judgments with two components was first proposed in (51), but typing rules does not support session types (only sequence of input/output) nor they allow to check the linearity of sessions. All remaining contents are introduced here.

$$
\begin{array}{llll}
P, Q & ::= & \mathbf{0} & \text{(nil)} \\
 & | & v.P & \text{(service definition)} \\
 & | & r^p \rhd P & \text{(session)} \\
 & | & \overline{v}.Q & \text{(invocation)} \\
 & | & \texttt{if } v = w \texttt{ then } P \texttt{ else } Q & \text{(if-then-else)} \\
 & | & (\tilde{x}).P & \text{(tuple input)} \\
 & | & \langle \tilde{v} \rangle.P & \text{(value output)} \\
 & | & \Sigma_{i=1}^{n}(l_i).P_i & \text{(label guarded sum)} \\
 & | & \langle l \rangle.P & \text{(label choice)} \\
 & | & \texttt{return } \tilde{v}.P & \text{(value return)} \\
 & | & P|Q & \text{(parallel)} \\
 & | & (\nu m)P & \text{(restriction)} \\
 & | & \texttt{rec } X.P & \text{(recursion)} \\
 & | & P > \tilde{x} > Q & \text{(pipeline)} \\
 & | & X & \text{(process variable)} \\
v, w & ::= & x & \text{(variable)} \\
 & | & a & \text{(service)} \\
 & | & \ldots, -1, 0, 1, \ldots & \text{(integers)}
\end{array}
$$

Figure 12: Syntax of our service calculus

4.2 Syntax and operational semantics

We assume an infinite collection $r, s \ldots$ of session names, an infinite collection $a, \ldots$ of service names, an infinite collection of variables $x, y, \ldots$, an infinite collection of process variables $X, Y, \ldots$ and an infinite collection $l, \ldots$ of labels. Additionally we use m, n to range over both service and session names. We let $\tilde{\cdot}$ denote tuples and $|\cdot|$ their cardinality. The syntax of processes $P, Q, \ldots$ is defined in Figure 12, where values $v, w, \ldots$ are either variables, services or integers. For practical reason we assume every variables appearing in a tuple $\tilde{x}$ is different from each other so the variables-tuple for values-tuple substitution $[^{v_1,\ldots,v_n}/_{x_1,\ldots,x_n}]$ stays for the sequence of substitutions $[^{v_1}/_{x_1}] \ldots [^{v_n}/_{x_n}]$.

As usual $\mathbf{0}$ is the nil process (whose trailing is often omitted), parallel composition is denoted by $P|Q$, restriction by $(\nu m)P$ and recursion by `rec` $X.P$. The construct $r^p \rhd P$ indicates a generic session side with polarity $p \in \{+, -\}$. We use p, q as meta variables to range over session polarities and $\overline{p}$ is the dual polarity of p, i.e.; $\overline{+} = -$ and $\overline{-} = +$. A fresh session name r and two polarized session ends $r^- \rhd P$ and $r^+ \rhd Q$ are

generated (on client and service sides, respectively) upon each service invocation $\overline{a}.P$ of the service $a.Q$.

Then, polyadic I/O communications in P and Q are uniquely directed toward the dual side labeled with the same name r but with dual polarity. Communication primitives inside each session are standard for session typed calculi: input/output (i.e. abstractions $(\tilde{x})$ and concretions $\langle\tilde{v}\rangle$ prefixes) and internal/external choices (i.e. a way for expressing a choice $\langle l\rangle.P$ on one side among a set of available options $\Sigma_i(l_i).Q_i$ at the other side). Sessions can be arbitrarily nested (e.g., in $\overline{a_1}.(P|\overline{a_2}.Q)$ the session established with a_2 will run inside the session established with a_1) and the `return` primitive outputs a value to (the partner of) the parent session. Finally $P > \tilde{x} > Q$ allows to output a tuple, bound in $\tilde{x}$, directly to a new instance of Q, that is each time P outputs a value then a new copy of Q is spawned to receive the value.

In few words the calculus allows all the standard in-session communication constructs towards dual sessions and allows only outputs towards the parent and the current sessions with `return` $\tilde{v}$ $.P$ and $P > \tilde{x} > Q$ respectively.

Priority of the operators in order of increasing relevance is: $|$, pipe, $\triangleright$ and ν, so for example $r^p \triangleright P|Q$ means $(r^p \triangleright P)|Q$ and $(\nu r)P|Q$ means $((\nu r)P)|Q$. Binders of the calculus are $(\tilde{x}).P$ for $\tilde{x}$ in P, $P > \tilde{x} > Q$ for $\tilde{x}$ in Q and $(\nu m)P$ for m in P, and `rec` $X.P$ for X in P, with standard notions of free names (fn defined in Figure 13) and bound names (bn) of a process, bound (bpv) and free process variables (fpv) defined in Figure 14 and closed processes (when $\mathsf{fpv}(P) = \emptyset$). Processes are considered equivalent up to alpha renaming of bound names and bound processes variables. Sometimes, especially in proofs, we need to be more specific about session names with polarity, in which case we define the set of free polarized names ($\mathsf{fpn}(P)$) in Figure 13 which is the set of (only) session names with polarity information of P. Notice that $r \notin \mathsf{fn}(P)$ implies $\{r^+, r^-\} \cap \mathsf{fpn}(P) = \emptyset$.

The structural congruence relation $\equiv$ reported in Figure 15 is the standard π-calculus structural congruence, with additional axioms for the floating of restrictions w.r.t. session side and pipe constructs. As common when dealing with concurrency the structural congruence is used to couple redexes so the reduction rules are given specifying redexes, without caring about the other processes in parallel. The presence of nesting, however requires a little care since the reduction can happen at different level of sessions nesting. For example in $r^+ \triangleright r_1^- \triangleright \langle 5\rangle.P \mid r_1^+ \triangleright ((r_2^- \triangleright Q_1) \mid (x).Q_2)$ we have an enabled communication (a delivery of a mes-

$$
\begin{array}{lcl}
\mathsf{fn}(\mathbf{0}) & = & \emptyset \\
\mathsf{fn}(v.P) & = & \mathsf{fn}(v) \cup \mathsf{fn}(P) \\
\mathsf{fn}(r^p \triangleright P) & = & \{r\} \cup \mathsf{fn}(P) \\
\mathsf{fn}(\overline{v}.P) & = & \mathsf{fn}(v) \cup \mathsf{fn}(P) \\
\mathsf{fn}(\ \mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q) & = & \mathsf{fn}(v) \cup \mathsf{fn}(w) \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q) \\
\mathsf{fn}((\tilde{x}).P) & = & \mathsf{fn}(P) \setminus \{\tilde{x}\} \\
\mathsf{fn}(\langle \tilde{v} \rangle.P) & = & \mathsf{fn}(P) \cup \mathsf{fn}(\tilde{v}) \\
\mathsf{fn}(\Sigma_{i=1}^{n}(l_i).P_i) & = & \bigcup_{i=1}^{n} \mathsf{fn}(P_i) \\
\mathsf{fn}(\langle l \rangle.P) & = & \mathsf{fn}(P) \\
\mathsf{fn}(\mathtt{return}\ \tilde{v}.P) & = & \mathsf{fn}(P) \cup \mathsf{fn}(\tilde{v}) \\
\mathsf{fn}(P|Q) & = & \mathsf{fn}(P) \cup \mathsf{fn}(Q) \\
\mathsf{fn}((\nu m)P) & = & \mathsf{fn}(P) \setminus \{m\} \\
\mathsf{fn}(\mathtt{rec}\ X.P) & = & \mathsf{fn}(P) \\
\mathsf{fn}(P > \tilde{x} > Q) & = & \mathsf{fn}(P) \cup (\mathsf{fn}(Q) \setminus \{\tilde{x}\}) \\
\mathsf{fn}(X) & = & \emptyset \\
\mathsf{fn}(a) & = & \{a\} \\
\mathsf{fn}(x) & = & \{x\} \\
\mathsf{fn}(..,0,1,..) & = & \emptyset \\
 & & \\
\mathsf{fpn}(r^p \triangleright P) & = & \{r^p\} \cup \mathsf{fpn}(P) \\
\mathsf{fpn}((\nu r)P) & = & \mathsf{fpn}(P) \setminus \{r^+, r^-\} \\
\mathsf{fpn}((\nu a)P) & = & \mathsf{fpn}(P) \\
\mathsf{fpn}(P) & = & \ldots
\end{array}
$$

Figure 13: Definition of free names and free polarized names

sage by standard message passing) of value 5 to process Q_2, however both the sender and the receiver are inside specific sessions.

We present the operational semantics of the calculus by means of reduction contexts. We have four different types of contexts generated from respective grammars. The one-hole context $\mathbb{C}[\![\cdot]\!]$ is useful to insert a process P, the result being denoted $\mathbb{C}[\![P]\!]$ (process P replaces the hole inside $\mathbb{C}$), into an arbitrary nesting of sessions together with arbitrary processes in parallel. The one-hole context $\mathbb{C}_{r^p}$ allows inserting a process into the session r^p running in parallel with an arbitrary process. Contexts $\mathbb{D}$ and $\mathbb{D}_r$ are the two-holes counterparts of the previous contexts. These contexts are a subclass of syntactic contexts, for instance in each context, neither binders nor service invocation/definition cannot appear above the hole since they are not enabled for immediate reductions. The operational semantics is reported below.

$$
\begin{array}{lcl}
\mathsf{fpv}(\mathbf{0}) & = & \emptyset \\
\mathsf{fpv}(v.P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(r^p \rhd) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\overline{v}.P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\ \texttt{if}\ v = w\ \texttt{then}\ P\ \texttt{else}\ Q) & = & \mathsf{fpv}(P) \cup \mathsf{fpv}(Q) \\
\mathsf{fpv}((\tilde{x}).P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\langle\tilde{v}\rangle.P) & = & \mathsf{fpv}(P)
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{fpv}(\Sigma_{i=1}^{n}(l_i).P_i) & = & \bigcup_{i=1}^{n} \mathsf{fpv}(P_i) \\
\mathsf{fpv}(\langle l\rangle.P) & = & \mathsf{fn}(P) \\
\mathsf{fpv}(\texttt{return}\ \tilde{v}.P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(P|Q) & = & \mathsf{fpv}(P) \cup \mathsf{fpv}(Q) \\
\mathsf{fpv}((\nu m)P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\texttt{rec}\ X.P) & = & \mathsf{fpv}(P) \setminus \{X\} \\
\mathsf{fpv}(P > \tilde{x} > Q) & = & \mathsf{fpv}(P) \cup (\mathsf{fpv}(Q) \\
\mathsf{fpv}(X) & = & \{X\}
\end{array}
$$

Figure 14: Definition of free process variables

$$
\begin{array}{l}
P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R) \\
(\nu m)P|Q \equiv (\nu m)(P|Q) \text{ if } m \notin \mathsf{fn}(Q) \\
(\nu m)\mathbf{0} \equiv \mathbf{0} \qquad (\nu m)P > \tilde{x} > Q \equiv (\nu m)(P > \tilde{x} > Q) \ \text{ if } m \notin \mathsf{fn}(Q)\{\tilde{x}\} \\
(\nu m)(\nu n)P \equiv (\nu n)(\nu m)Q \qquad r^p \rhd (\nu m)P \equiv (\nu m)r^p \rhd P \text{ if } m \neq r
\end{array}
$$

Figure 15: Structural congruence

$$
\begin{array}{lll}
(\textsc{Inv}) & \mathbb{D}[\![\overline{a}.P, a.Q]\!] \ \rightarrow\ (\nu r)\mathbb{D}[\![r^- \rhd P, r^+ \rhd Q]\!] & (r \notin \mathsf{fn}(\mathbb{D}[\![\overline{a}.P, a.Q]\!])) \\
(\textsc{Com}) & \mathbb{D}_r[\![(\tilde{x}).P, \langle\tilde{v}\rangle.Q]\!] \ \rightarrow\ \mathbb{D}_r[\![P[^{\tilde{v}}/_{\tilde{x}}], Q]\!] & (|\tilde{x}| = |\tilde{v}|) \\
(\textsc{LCom}) & \mathbb{D}_r[\![\Sigma_{i=1}^{n}(l_i).P_i, \langle l_k\rangle.Q]\!] \ \rightarrow\ \mathbb{D}_r[\![P_k, Q]\!] & (1 \leq k \leq n) \\
(\textsc{Ret}) & \mathbb{D}_{r_1}[\![(\tilde{x}).P, \mathbb{C}_{r^q}[\![\texttt{return}\ \tilde{v}.Q]\!]]\!] \ \rightarrow\ \mathbb{D}_{r_1}[\![P[^{\tilde{v}}/_{\tilde{x}}], \mathbb{C}_{r^q}[\![Q]\!]]\!] & \\
(\textsc{Pipe}) & \mathbb{C}[\![(\langle\tilde{v}\rangle.P|P') > \tilde{x} > Q]\!] \ \rightarrow\ \mathbb{C}[\![(P|P') > \tilde{x} > Q \mid Q[^{\tilde{v}}/_{\tilde{x}}]]\!] & (|\tilde{x}| = |\tilde{v}|) \\
(\textsc{PipeRet}) & \mathbb{C}[\![(\mathbb{C}_{r^p}[\![\texttt{return}\ \tilde{v}.P]\!]|P') > \tilde{x} > Q]\!] \ \rightarrow\ \begin{array}{l}\mathbb{C}[\![(\mathbb{C}_{r^p}[\![P]\!]|P') > \\ \tilde{x} > Q \mid Q[^{\tilde{v}}/_{\tilde{x}}]]\!]\end{array} & (|\tilde{x}| = |\tilde{v}|) \\
(\textsc{IfT}) & \mathbb{C}[\![\texttt{if}\ v = v\ \texttt{then}\ P\ \texttt{else}\ Q]\!] \rightarrow \mathbb{C}[\![P]\!] & \\
(\textsc{IfF}) & \mathbb{C}[\![\texttt{if}\ v = w\ \texttt{then}\ P\ \texttt{else}\ Q]\!] \rightarrow \mathbb{C}[\![Q]\!] & (v \neq w) \\
(\textsc{Rec}) & \mathbb{C}[\![P[^{\texttt{rec}\ X.P}/_X]]\!] \rightarrow P' \Rightarrow \mathbb{C}[\![\texttt{rec}\ X.P]\!] \rightarrow P' & \\
(\textsc{Scop}) & P \ \rightarrow\ P' \ \Rightarrow\ (\nu m)P \rightarrow (\nu m)P' & \\
(\textsc{Str}) & P \equiv P' \ \wedge\ P' \rightarrow Q' \ \wedge\ Q' \equiv Q \Rightarrow\ P \rightarrow Q &
\end{array}
$$

$$
\begin{array}{lcl}
\mathbb{C} & ::= & [\![\cdot]\!] \mid \mathbb{C}|P \mid r^p \rhd \mathbb{C} \mid \mathbb{C} > \tilde{x} > P \\
\mathbb{D} & ::= & \mathbb{C}[\![\mathbb{C}' \mid \mathbb{C}'']\!]
\end{array}
\qquad
\begin{array}{lcl}
\mathbb{C}_{r^p} & ::= & r^p \rhd ([\![\cdot]\!]|P) \\
\mathbb{D}_r & ::= & \mathbb{D}[\![\mathbb{C}'_{r^p}, \mathbb{C}''_{r^{\overline{p}}}]\!]
\end{array}
$$

Rule (Inv) defines how the invocation of a service creates a new session r that puts in direct communication an instance of the service protocol Q with the client body P. Notice how the definition of a service is consumed once it is used. Rules (Com) and (LCom) show respectively how a tuple is transmitted (by means of message passing) between the two sides of a session and how the dual partner can select one of the options offered by the other. As usual $P[^{\tilde{v}}/_{\tilde{x}}]$ is the simultaneous substitution of

each variable $\tilde{x}$ with values $\tilde{v}$ in P if $|\tilde{x}| = |\tilde{v}|$. Rule (Ret) illustrates how a nested session r^q can output a value upward to the parent session r_1^p, which is read by $(\tilde{x}).P$ in $r_1^{\overline{p}}$. Rules (Pipe) and (PipeRet) describe how pipe works: the former allows the direct delivery (by means of an output) of $\tilde{v}$ to Q while the latter manages the delivery of a value $\tilde{v}$ from a nested session (by means of a return) to Q. In both cases a new instance of Q is spawned and the pipe rests waiting another value. Notice how we used the same output primitive but we assigned a different semantics depending of the surrounding syntactic context. Recursion is given with rule (Rec) in the operational semantics, another solution is to use the rule built-in with the structural congruence but we do not pursue this way since we want to keep distinct the role of the structural congruence from the role of the evaluation function. Remaining rules (IfT), (IfF) locally evaluate the if guard, rule (Scop) adds restrictions and (Str) couples redexes by means of the structural congruence.

We now introduce two running examples which we also continue using in the next sections to highlight some peculiarities of the type system.

Example 4.1. In this first example we show how different replicas of a service are allowed, as long as each replica has a common type even in presence of syntactically different processes. In fact the operational semantics in presence of multiple definitions non-deterministically choose one of them.

$$(\nu a)\big(\overline{a}.\mathtt{rec}\ X.\langle l\rangle.\Sigma_{i=1}^{4}(l_i).X \quad \big|\ \mathtt{rec}\ X_1.\big(a.\mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2 \ \big|\ X_1\big) \\ \big|\ a.\mathtt{rec}\ Y_1.(l).\mathtt{if}\ test\ \begin{array}{l}\mathtt{then}\ \langle l_1\rangle.Y_1\\ \mathtt{else}\ \langle l_3\rangle.Y_1\end{array}\ \big)$$

The definition of service a is duplicated (in the sense that there are two different definitions of a) and despite of their different syntactic structure both protocols of the replicas are compatible with the client (in the sense of subtyping relation, see Definition 3.4). They are in fact both able to recursively offer l (the first one by means of a choice which offers l *twice*) and then to select one of the labels offered by the client (either l_1 or l_2 in the case of the first service definition, depending on which l-branch is used; either l_1 or l_3 in the case of the second definition, depending on some internal condition $test$). Also notice that the first definition of service a is replicated by means of recursion and that the well-formedness of session types does not forbid using an external choice guarded with the same label.

Example 4.2. In this example we show recursivity using mutual references to different services.

$$P = a.(x).\overline{x}.\langle a\rangle \mid \overline{a}.\langle b\rangle \mid b.(x)$$

Service a inputs a service x that in turn is invoked again with a service of type a. We also show the evaluation steps:

$$\begin{array}{l} P \rightarrow (\nu r)(r^+ \rhd (x).\overline{x}.\langle a\rangle \mid r^- \rhd \langle b\rangle) \mid b.(x) \rightarrow \\ (\nu r)(r^+ \rhd \overline{b}.\langle a\rangle \mid r^- \rhd \mathbf{0}) \mid b.(x) \rightarrow \\ (\nu r)(\nu s)(r^+ \rhd s^- \rhd \langle a\rangle \mid r^- \rhd \mathbf{0} \mid s^+ \rhd (x)) \rightarrow \\ (\nu r)(\nu s)(r^+ \rhd s^- \rhd \mathbf{0} \mid r^- \rhd \mathbf{0} \mid s^+ \rhd \mathbf{0}) \end{array}$$

4.3 Type system

4.3.1 A type system for CST

The set of types we consider is defined in Figure 1 we retain same notions of well formedness, of bound (bv) and free type variables (fv), closed μ-types, and equivalence of μ-types up to alpha-renaming of bound variables. We start introducing the type discipline of CST. We consider two typing environments Γ and Θ, which are finite partial mappings. Γ is the *standard typing environment* and maps service/polarized session names/variables to sort types S and Θ is the *processes typing environment* and maps process variables to couples of session types of the form $T;U$. We write $\mathsf{dom}(\Gamma)$ (resp. $\mathsf{dom}(\Theta)$) for the set of elements in the domain of Γ (resp. Θ) and $\emptyset$ is the function with empty domain. With $\Gamma, x : S$ we point the partial function Γ' s.t. $\Gamma'(y) = \Gamma(y)$ if $x \neq y$ and $\Gamma'(y) = S$ otherwise, in the same manner we define $\Gamma, r^p : [T]$ and $\Gamma, a : [T]$ and $\Theta, X : T; U$. For example $\emptyset, x : int, y : int = \emptyset, y : int, x : int$ is the partial function with domain $\{x, y\}$ that maps both x and y to an integer type. The initial $\emptyset$, is often omitted when the function is defined for some values. With the help of these definitions we also define a syntactic notion of well formedness for typing environments; for simplicity we require that if a session $\Gamma(r^p) = [T]$ then it must be also that $\Gamma(r^{\overline{p}}) = [\overline{T}]$.

Definition 4.3 (Well-formed typing environments)**.** Γ and Θ are *well-formed* if they are generated from the following grammar:

$$\begin{array}{l} \Gamma ::= \emptyset \mid \Gamma, a : [T] \mid \Gamma, x : S \mid \Gamma, r^p : [T], r^{\overline{p}} : [\overline{T}] \\ \Theta ::= \emptyset \mid \Theta, X : T; U \end{array}$$

We consider only well-formed environments and we write $\Gamma, r : [T]$ (r without polarity annotation) as a shorthand for $\Gamma, r^+ : [T], r^- : [\overline{T}]$.

We have two kinds of typing judgments reported in Figure 16, $\Gamma \vdash m : S$ for values and $\Gamma; \Theta \vdash P : T; U; L$ for processes. The first four typing rules are for values and are standard, in rule (IntV) $\underline{n}$ stays for a generic integer. We sometimes use $\Gamma \vdash \tilde{m} : \tilde{S}$ where $\tilde{m}$=$m_1, \ldots, m_n$ and $\tilde{S} = S_1, \ldots, S_n$ to mean that $\Gamma \vdash m_1 : S_1$,.., $\Gamma \vdash m_n : S_n$ hold. In the typing judgments for processes, T is the type of the behavior of P w.r.t. the current session, U is the type of the behavior of P w.r.t. the parent session and L is a set of polarized session names, which is used to check the linearity of sessions. We call the triple $T; U; L$ *the linear typing environment* and we refer to T (resp. to U, L) with T-component (resp. to U-component, L-component). The presence of L allows the type system to check the linear usage of each session in a process. Since the type of each session evolves during reductions, linearity is fundamental in order to keep typing preservation along reductions, namely the Subject Reduction. Consider for example the process $r^+ \triangleright (x) \mid r^+ \triangleright (x) \mid r^- \triangleright \langle 5 \rangle$, the process would be well typed with assumptions $r^+ : [?(int)], r^- : [?(int)]$, but after a reduction step it is not well typed anymore.

Some comments about the typing rules are in order. Rule (TNewR) guesses two dual types for each session sides whilst (TNew) guesses only the type of the service a. This choice reflects the asymmetry between client and service and the symmetry between two dual session sides. The side condition on the cardinality of $|L \cap \{r^+, r^-\}|$ allows $\{r^+, r^-\}$ optional in L. Rule (TDef) constrains the protocol of the service to be the same as the body type of the process P and (TInv), instead checks that the dual type of the client protocol is the same as the service protocol. Rule (TSes) is similar to (TDef) but in addition it handles the linearity check on r^p w.r.t. the set L. We require, in fact with the help of the disjoint union $L \uplus \{r^p\}$ that in $r^p \triangleright P$ there is no sub-process of P of the form $r^p \triangleright P'$. Rules for parallel composition (TParL) and (TParR) do the same check with the two set of labels L_1 and L_2. Besides, they check that at least one process among P and Q has type `end` for both the current and the parent session. This check is important to prevent parallel actions inside a session, because session types are not powerful enough to model parallel composition of two session types. Rules (TIn), (TOut) and (TRet) insert either the input or the output in the correct place. Rule (TBranch) allows to insert in the result type only a subset J of the total choices, while rule (TChoice) adds arbitrarily branches in the type together with the branch

(SER) $\Gamma, a : [T] \vdash a : [T]$ (VAR) $\Gamma, x : S \vdash x : S$ (SES) $\Gamma, r^p : [T] \vdash r^p : [T]$ (INTV) $\Gamma \vdash \underline{n} : int$

(TNIL) $\Gamma; \Theta \vdash \mathbf{0} : \mathsf{end}; \mathsf{end}; \emptyset$

(TNEW)
$$\frac{\Gamma, a : S; \Theta \vdash P : T; U; L}{\Gamma; \Theta \vdash (\nu a)P : T; U; L}$$

(TNEWR)
$$\frac{\Gamma, r^+ : [T], r^- : [\overline{T}]; \Theta \vdash P : T'; U; L \quad |L \cap \{r^+, r^-\}| \neq 1}{\Gamma; \Theta \vdash (\nu r)P : T'; U; L \setminus \{r^+, r^-\}}$$

(TIF)
$$\frac{\Gamma \vdash v_i : S \quad i = 1,2 \quad \Gamma; \Theta \vdash P : T; U; L \quad \Gamma; \Theta \vdash Q : T; U; L}{\Gamma; \Theta \vdash \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q : T; U; L}$$

(TREC)
$$\frac{\Gamma; \Theta, X : T; U \vdash (P : T; U; \emptyset)^\star}{\Gamma; \Theta \vdash \mathtt{rec}\ X.P : T; U; \emptyset}$$

(TWEAK)
$$\frac{\Gamma; \Theta \vdash P : T; U; L \quad T' \leq T \quad U' \leq U}{\Gamma; \Theta \vdash P : T'; U'; L}$$

(TPVAR) $\Gamma; \Theta, X : T; U \vdash X : T; U; \emptyset$

(TSES)
$$\frac{\Gamma; \Theta \vdash P : T; U; L \quad \Gamma \vdash r^p : [T]}{\Gamma; \Theta \vdash r^p \triangleright P : U; \mathsf{end}; L \uplus \{r^p\}}$$

(TDEF)
$$\frac{\Gamma; \Theta \vdash P : T; U; L \quad \Gamma \vdash v : [T]}{\Gamma; \Theta \vdash v.P : U; \mathsf{end}; L}$$

(TINV)
$$\frac{\Gamma; \Theta \vdash P : \overline{T}; U; L \quad \Gamma \vdash v : [T]}{\Gamma; \Theta \vdash \overline{v}.P : U; \mathsf{end}; L}$$

(TIN)
$$\frac{\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P : T; U; L}{\Gamma; \Theta \vdash (\tilde{x}).P : ?(\tilde{S}).T; U; L}$$

(TOUT)
$$\frac{\Gamma; \Theta \vdash P : T; U; L \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma; \Theta \vdash \langle \tilde{v} \rangle.P : !(\tilde{S}).T; U; L}$$

(TRET)
$$\frac{\Gamma; \Theta \vdash P : T; U; L \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma; \Theta \vdash \mathtt{return}\ \tilde{v}.P : T; !(\tilde{S}).U; L}$$

(TBRANCH)
$$\frac{\emptyset \subset J \subseteq I = \{1, \ldots, n\} \quad \forall i \in I,\ \Gamma; \Theta \vdash P_i : T_i; U; L}{\Gamma; \Theta \vdash \Sigma_{i=1}^n (l_i).P_i : \&\{l_j : T_j\}_{j \in J}; U; L}$$

(TCHOICE)
$$\frac{l = l_i \in \{l_1, \ldots, l_n\} \quad \Gamma; \Theta \vdash P : T_i; U; L}{\Gamma; \Theta \vdash \langle l \rangle.P : \oplus\{l_1 : T_1, \ldots, l_n : T_n\}; U; L}$$

(TPARL)
$$\frac{\Gamma; \Theta \vdash P : T; U; L_1 \quad \Gamma; \Theta \vdash Q : \mathsf{end}; \mathsf{end}; L_2}{\Gamma; \Theta \vdash P | Q : T; U; L_1 \uplus L_2}$$

(TPARR)
$$\frac{\Gamma; \Theta \vdash P : \mathsf{end}; \mathsf{end}; L_1 \quad \Gamma; \Theta \vdash Q : T; U; L_2}{\Gamma; \Theta \vdash P | Q : T; U; L_1 \uplus L_2}$$

(TPIPE)
$$\frac{\Gamma; \emptyset \vdash P : T_1; \mathsf{end}; L \quad \Gamma, \tilde{x} : \tilde{S}; \Theta \vdash Q : T_2; U_2; \emptyset \quad (T, U) = \mathsf{pipe}(T_1, T_2, U_2, \tilde{S})}{\Gamma; \Theta \vdash P > \tilde{x} > Q : T; U; L}$$

Figure 16: Typing rules: rule (TREC) for recursion requires an additional consistency condition

$$
\begin{array}{lcl}
nocuract(\mathbf{0}) & & \\
nocuract(v.P) & = & noretact(P) \\
nocuract(\texttt{return } \tilde{v}.P) & = & nocuract(P) \\
nocuract(\overline{v}.P) & = & noretact(P) \\
nocuract(r^p \triangleright P) & = & noretact(P) \\
nocuract(\texttt{if then } P \texttt{ else } Q) & = & nocuract(P) \text{ and } nocuract(Q) \\
nocuract(P|Q) & = & nocuract(P) \text{ and } nocuract(Q) \\
nocuract(P > \tilde{x} > Q) & = & nocuract(Q) \\
nocuract((\nu m)P) & = & nocuract(P) \\
nocuract(\texttt{rec } X.P) & = & nocuract(P) \\
nocuract(X) & &
\end{array}
\qquad
\begin{array}{lcl}
noretact(\mathbf{0}) & & \\
noretact(v.P) & & \\
noretact(\overline{v}.P) & & \\
noretact(r^p \triangleright P) & & \\
noretact(\texttt{if then } P \texttt{ else } Q) & = & noretact(P) \text{ and } noretact(Q) \\
noretact((\tilde{x}).P) & = & noretact(P) \\
noretact(\Sigma_{i=1}^{n}(l_i).P_i) & = & noretact(P_1) \ldots \text{ and } noretact(P_n) \\
noretact(\langle \tilde{v} \rangle.P) & = & noretact(P) \\
noretact(\langle l \rangle.P) & = & noretact(P) \\
noretact(P|Q) & = & noretact(P) \text{ and } noretact(Q) \\
noretact(P > \tilde{x} > Q) & = & noretact(Q) \\
noretact((\nu m)P) & = & noretact(P) \\
noretact(\texttt{rec } X.P) & = & noretact(P) \\
noretact(X) & &
\end{array}
$$

Figure 17: Two predicates to check the presence of active actions

labeled with l. Both these rules are not syntax directed and together with (TWEAK) (which in turn relaxes the type of T and U by substituting them with a subtype) allow some flexibility in the protocol specification of each side, for example we can correctly type $\mu\alpha.\&\{l : \texttt{end}, l_1 : \alpha\}$ against both $\overline{\oplus\{l_1 : \oplus\{l : \texttt{end}\}\}}$ and $\overline{\mu\alpha. \oplus \{l_1 : \alpha\}}$ (i.e. they are in subtyping relation). Rule (TPIPE) uses the pipe function defined as

$$
\begin{array}{l}
\mathsf{pipe}(\texttt{end}, T, U, \tilde{S}) = \texttt{end}, \texttt{end} \\
\mathsf{pipe}(!(\tilde{S}), T, U, \tilde{S}) = T, U
\end{array}
$$

This function allows a pipe to be used only as a sequencing operator since replication of multiple instances of Q can cause parallel actions inside a session. In detail, consider $P > \tilde{x} > Q$ if P does not produce any output then the total type of the process is end for both the current and the parent sessions, if instead P produces only one output then the type of the entire process is the type of Q, since only one copy of Q will be spawned. We studied in (15) a way to relax the typing rules for pipe and parallel composition which can be easily accommodated here with minimal effort.

Finally rule (TREC) is the standard rule to type recursion but we need an additional condition to type the unguarded recursion. Consider for example the process $\texttt{rec } X.X$, using the above rules the process is typed as $\emptyset; \emptyset \vdash \texttt{rec } X.X : T; U; \emptyset$ for any T and U. To characterize all such cases we introduce two predicates (Figure 17) $nocuract(P)$ which holds if there is in P at least one active action in the current sessions and $noretact(P)$

which holds if in P there is at least one return action. The only non trivial case in both the two definitions is the case of a pipe, where we check only actions in the continuation Q, and it is strictly related to the function pipe. The consistency condition we impose in rule (TREC) pointed with $(P : T; U; \emptyset)^\star$ is "$nocuract(P)$ implies $T = \texttt{end}$ and $noretact(P)$ implies $U = \texttt{end}$". Take again the process $\texttt{rec}\ X.X$ with this new condition the *only* possible type for T and U is end.

Example 4.4. In Figure 18 we report the proof tree of the two definitions of service a in Example 4.1 where the set L is omitted for conciseness (it is equal to $\emptyset$) and $T_\alpha = \mu\alpha.\&\{l : \oplus\{l_1 : \alpha, l_2 : \alpha, l_3 : \alpha\}\}$, $\Gamma = a : [T_\alpha]$, $\Theta = Y_2 : T_\alpha; \texttt{end}; X_1 : \texttt{end}; \texttt{end}$ and $\Theta_1 = Y_1 : T_\alpha; \texttt{end}$. In rule (TPAR)$\star$ we omit the typing derivation for X_1.

Example 4.5. In figure 19 we report the proof tree of each of the three processes of Example 4.2 where the set L is omitted for conciseness (it is equal to $\emptyset$) and $T_a = [\mu\alpha.?([?([\alpha])]$, $\Gamma = a : [T_a], b : [?([T_a])]$. Notice that we can apply (TWEAK) since $\mu\alpha.?([?([\alpha])]) \leq ?([?(\mu\alpha.?([?([\alpha])]))])$.

4.3.2 Subject reduction and safety

The aim of this subsection is to prove that the typing is preserved along reduction steps which directly implies the so-called session safety. In Chapter 6, we prove a stronger property of this type system almost a direct consequence of the subject reduction, called progress in literature (26). Next three lemmas are standard: they allow respectively to add assumptions into each environment, to remove assumptions from environments and to collapse two assumptions.

Lemma 4.6 (Weakening). *If* $\Gamma; \Theta \vdash P : T; U; L$ *and* $m \notin \mathsf{fn}(P)$ *then* $\Gamma, m : S; \Theta \vdash P : T; U; L$ *for any* S*. If* $\Gamma; \Theta \vdash P : T; U; L$ *and* $X \notin \mathsf{fpv}(P)$ *then* $\Gamma; \Theta, X : T'; U' \vdash P : T; U; L$ *for any* T', U'*.*

Proof. The proof is by straightforward induction on the derivation of $\Gamma; \Theta \vdash P : T; U; L$. □

Lemma 4.7 (Strengthening). *If* $\Gamma, m : S; \Theta \vdash P : T; U; L$ *and* $m \notin \mathsf{fn}(P)$ *then* $\Gamma; \Theta \vdash P : T; U; L$*. If* $\Gamma; \Theta, X : T'; U' \vdash P : T; U; L$ *and* $X \notin \mathsf{fpv}(P)$ *then* $\Gamma; \Theta \vdash P : T; U; L$*.*

Proof. The proof of the two statements are by straightforward induction on the derivation of $\Gamma, m : S; \Theta \vdash P : T; U; L$ and $\Gamma; \Theta, X : T'; U' \vdash P : T; U; L$ respectively. □

Lemma 4.8 (Substitution lemma). *If* $\Gamma, x : S; \Theta \vdash P : T; U; L$ *and* $\Gamma \vdash v : S$ *then* $\Gamma; \Theta \vdash P[^v/_x] : T; U; L$*. If* $\Gamma; \Theta, X : T'; U' \vdash Q : T; U; L$ *and* $\Gamma; \Theta \vdash P : T'; U'; \emptyset$ *then* $\Gamma; \Theta \vdash Q[^P/_X] : T; U; L$*.*

Proof. Without loss of generality we prove the theorem for the case when the tuple length in input, output (and pipe) actions is one. The proof of the first statement is by induction on the derivation of $\Gamma, x : S; \Theta \vdash P : T; U; L$ with case analysis on the last applied rule. We sketch the interesting cases. When the last applied rule is (TOUT) and $\dot{P} = \langle v \rangle.P$ the not obvious case is when $\dot{x} = v$ otherwise the theorem follows by induction. We have $\dfrac{\Gamma, v : S, \dot{v} : S; \Theta \vdash P : T; U; L}{\Gamma, v : S, \dot{v} : S; \Theta \vdash \langle v \rangle.P :!(S).T; U; L}\,(\text{TOUT})$ and by induction $\Gamma, \dot{v} : S; \Theta \vdash P[^{\dot{v}}/_v] : T; U; L$ holds. Applying rule (TOUT) to the last judgment we have $\Gamma, \dot{v} : S; \Theta \vdash \langle \dot{v} \rangle.P[^{\dot{v}}/_v] :!(S).T; U; L$ and the thesis follows since $\langle \dot{v} \rangle.P[^{\dot{v}}/_v] = (\langle v \rangle.P)[^{\dot{v}}/_v]$. The other non-trivial cases are when the last applied rule is (TDEF) or (TINV). We have $\dfrac{\Gamma, v : [T], \dot{v} : [T]; \Theta \vdash P : \overline{T}; U; L}{\Gamma, v : S, \dot{v} : S; \Theta \vdash \overline{v}.P : U; \mathsf{end}; L}\,(\text{TINV})$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma;\Theta\vdash Y_2 : T_\alpha;\mathsf{end}}{\Gamma;\Theta\vdash\langle l_2\rangle.Y_2 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha\};\mathsf{end}}}{\Gamma;\Theta\vdash(l).\langle l_2\rangle.Y_2 : \mathsf{unfold}(T_\alpha);\mathsf{end}}
\qquad
\dfrac{\dfrac{\Gamma;\Theta\vdash Y_2 : T_\alpha;\mathsf{end}}{\Gamma;\Theta\vdash\langle l_1\rangle.Y_2 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha\};\mathsf{end}}}{\Gamma;\Theta\vdash(l).\langle l_1\rangle.Y_2 : \mathsf{unfold}(T_\alpha);\mathsf{end}}}{\Gamma;\Theta\vdash\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2 : T_\alpha;\mathsf{end}}\ (\text{TBRANCH})}{\Gamma; X_1 : \mathsf{end};\mathsf{end}\vdash\mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2 : T_\alpha;\mathsf{end}}\ (\text{TWEAK—TREC})}{\Gamma; X_1 : \mathsf{end};\mathsf{end}\vdash a.\mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2;\mathsf{end};\mathsf{end}}\ (\text{TWEAK—TDEF})}{\Gamma; X_1 : \mathsf{end};\mathsf{end}\vdash a.\mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2|X_1;\mathsf{end};\mathsf{end}}\ (\text{TPAR})\star}{\Gamma;\emptyset\vdash\mathtt{rec}\ X_1.(a.\mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2|X_1);\mathsf{end};\mathsf{end}}\ (\text{TREC})
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma;\Theta_1\vdash Y_1 : T_\alpha;\mathsf{end}}{\Gamma;\Theta_1\vdash\langle l_3\rangle.Y_1 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\};\mathsf{end}}\qquad\vdots}{\Gamma;\Theta_1\vdash\mathtt{if}\ test\ \mathtt{then}\ \langle l_1\rangle.Y_1\ \mathtt{else}\ \langle l_3\rangle.Y_1 : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\};\mathsf{end}}}{\Gamma;\Theta_1\vdash(l).\mathtt{if}\ test\ \mathtt{then}\ \langle l_1\rangle.Y_1\ \mathtt{else}\ \langle l_3\rangle.Y_1 : \&\{l : \oplus\{l_1 : T_\alpha, l_2 : T_\alpha, l_3 : T_\alpha\}\};\mathsf{end}}}{\Gamma;\emptyset\vdash(l).\mathtt{if}\ test\ \mathtt{then}\ \langle l_1\rangle.Y_1\ \mathtt{else}\ \langle l_3\rangle.Y_1 : \mathsf{unfold}(T_\alpha);\mathsf{end}}}{\Gamma;\emptyset\vdash\mathtt{rec}\ Y_1.(l).\mathtt{if}\ test\ \mathtt{then}\ \langle l_1\rangle.Y_1\ \mathtt{else}\ \langle l_3\rangle.Y_1 : T_\alpha;\mathsf{end}}\ (\text{TWEAK—TREC})}{\Gamma;\emptyset\vdash a.\mathtt{rec}\ Y_1.(l).\mathtt{if}\ test\ \mathtt{then}\ \langle l_1\rangle.Y_1\ \mathtt{else}\ \langle l_3\rangle.Y_1 : \mathsf{end};\mathsf{end}}\ (\text{TDEF})
$$

Figure 18: Example of typing

$$
\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x :?([T_\alpha];\emptyset\vdash\langle a\rangle :!([T_a]);\mathsf{end}}{\Gamma, x :?([T_\alpha]);\emptyset\vdash\overline{x}.\langle a\rangle : \mathsf{end};\mathsf{end}}\ (\text{TINV})}{\Gamma;\emptyset\vdash(x).\overline{x}.\langle a\rangle :?([?([T_a])]);\mathsf{end}}\ (\text{TIN})}{\Gamma;\emptyset\vdash(x).\overline{x}.\langle a\rangle : T_a;\mathsf{end}}\ (\text{TWEAK})}{\Gamma;\emptyset\vdash a.(x).\overline{x}.\langle a\rangle : \mathsf{end};\mathsf{end}}\ (\text{TDEF})
\qquad
\dfrac{\Gamma;\emptyset\vdash\langle b\rangle :!([?([T_a])]);\mathsf{end}}{\Gamma;\emptyset\vdash\overline{a}.\langle b\rangle : \mathsf{end};\mathsf{end}}\ (\text{TINV})
$$

$$
\dfrac{\Gamma;\emptyset\vdash(x) :?([T_a]);\mathsf{end}}{\Gamma;\emptyset\vdash b.(x) : \mathsf{end};\mathsf{end}}\ (\text{TDEF})
$$

Figure 19: Example of typing

and by induction $\Gamma, \dot{v} : [T]; \Theta \vdash P[^{\dot{v}}/_{v}] : \overline{T}; U; L$ holds. Applying rule (TINV) to the last judgment we have $\Gamma, \dot{v} : [T]; \Theta \vdash \overline{\dot{v}}.P[^{\dot{v}}/_{v}] : U; \mathtt{end}; L$ and we conclude since $\overline{\dot{v}}.P[^{\dot{v}}/_{v}] = (\overline{v}.P)[^{\dot{v}}/_{v}]$. The case for rule (TDEF) is similar. The proof of the second judgment is by induction on the derivation of $\Gamma; \Theta, X : T'; U' \vdash Q : T; U; L$ with case analysis on the last applied rule. The only non trivial case is the base case, rule (TPVAR), when $Q = \dot{X}$. We have $\Gamma; \Theta, \dot{X} : T'; U' \vdash \dot{X} : T'; U'; \emptyset$ and by hypothesis $\Gamma; \Theta \vdash \dot{P} : \dot{T'}; \dot{U'}; \emptyset$ which concludes since $X[^{\dot{P}}/_{X}] = \dot{P}$ □

We extend the previous lemma with tuples to fit the polyadic communications of the calculus.

Corollary 4.9 (Substitution lemma for tuples). *If* $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P : T; U; L$ *and* $\Gamma \vdash \tilde{v} : \tilde{S}$ *then* $\Gamma; \Theta \vdash P[^{\tilde{v}}/_{\tilde{x}}] : T; U; L$.

Proof. The proof is by induction on the tuple length. The base case with tuple length 1 is the Substitution Lemma. In the inductive case the theorem holds for tuple of length $n-1$ thus $\Gamma, x_1 : S_1 \ldots x_n : S_n; \Theta \vdash P : T; U; L$ and $\Gamma \vdash v_1 : S_1, \ldots, \Gamma \vdash v_n : S_n$ then $\Gamma, x_n : S_n; \Theta \vdash P[^{v_1,\ldots,n_{n-1}}/_{x_1,\ldots,x_{n-1}}]; T; U; L$. The result follows applying the Substitution Lemma to $P_1 = P[^{v_1,\ldots,n_{n-1}}/_{x_1,\ldots,x_{n-1}}]$. □

Sometimes we use the Weakening Lemma to add an assumption of a name (resp. of a process variable) that does not appear in the typing environment (resp. process typing environment) without caring about free names (resp. free process names). The following lemma justifies this fact.

Lemma 4.10. *Let* $\Gamma; \Theta \vdash P : T; U; L$. *If* m *is not a session then* $m \notin \mathsf{dom}(\Gamma)$ *implies* $m \notin \mathsf{fn}(P)$. *If* m *is a session* r *then* $\{r^+, r^-\} \cap \mathsf{dom}(\Gamma) = \emptyset$ *implies* $r \notin \mathsf{fn}(P)$. *If* $X \notin \mathsf{dom}(\Theta)$ *then* $X \notin \mathsf{fpv}(P)$.

Proof. The proof is by straightforward induction on the typing of $\Gamma; \Theta \vdash P : T; U; L$ with case analysis on the last applied rule. □

Subject Congruence proves that the typing is preserved by the structural congruence.

Proposition 4.11 (Subject Congruence). *If* $\Gamma; \Theta \vdash P : T; U; L$ *and* $P \equiv Q$ *then* $\Gamma; \Theta \vdash Q : T; U; L$

Proof. It suffices to prove that the statement holds in both directions for each congruence rule. Closure under arbitrary contexts ($\equiv$ is a congruence relation) is obvious from the typing system.

- $P|\mathbf{0} \equiv P$

$$\frac{\Gamma; \Theta \vdash P : T; U; L \qquad \Gamma; \Theta \vdash \mathbf{0} : \texttt{end}; \texttt{end}; \emptyset}{\Gamma; \Theta \vdash P|\mathbf{0} : T; U; L} \text{(TPARL)}$$

- $P|Q \equiv Q|P$ we have two cases depending of the rule used to derive the judgment of $P|Q$. If the judgment has been derived by rule (TPARL) then

$$\frac{\Gamma; \Theta \vdash P : T; U; L_1 \qquad \Gamma; \Theta \vdash Q : \texttt{end}; \texttt{end}; L_2}{\Gamma; \Theta \vdash P|Q : T; U; L_1 \uplus L_2} \text{(TPARL)}$$

$$\frac{\Gamma; \Theta \vdash Q : \texttt{end}; \texttt{end}; L_2 \qquad \Gamma; \Theta \vdash P : T; U; L_1}{\Gamma; \Theta \vdash Q|P : T; U; L_1 \uplus L_2} \text{(TPARR)}$$

otherwise,

$$\frac{\Gamma; \Theta \vdash P : \texttt{end}; \texttt{end}; L_1 \qquad \Gamma; \Theta \vdash Q : T; U; L_2}{\Gamma; \Theta \vdash P|Q : T; U; L_1 \uplus L_2} \text{(TPARR)}$$

$$\frac{\Gamma; \Theta \vdash Q : T; U; L_2 \qquad \Gamma; \Theta \vdash P : \texttt{end}; \texttt{end}; L_1}{\Gamma; \Theta \vdash Q|P : T; U; L_1 \uplus L_2} \text{(TPARL)}$$

- $(P|Q)|R \equiv P|(Q|R)$ in order to be well-typed two processes among P, Q and R should be typed with a type different from end; end. Consequently we have three possibilities: typing in turn either P or Q or R with a generic type $T; U$ and with end; end the remaining two.

- $(\nu m)P|Q \equiv (\nu m)(P|Q)$ if $m \notin \mathsf{fn}(Q)$ we have two possibilities according if m is a session or a service; the two cases are similar. For example if $m = r$ and $L_1 \cap \{r^+, r^-\} = \emptyset$ or $L_1 \cap \{r^+, r^-\} = \{r^+, r^-\}$ we have:

$$\frac{\dfrac{\Gamma, r : [T]; \Theta \vdash P : T; U; L_1}{\Gamma; \Theta \vdash (\nu r)P : T; U; L_1 \setminus \{r^+, r^-\}} \text{(TNEWR)} \qquad \Gamma; \Theta \vdash Q : \texttt{end}; \texttt{end}; L_2}{\Gamma; \Theta \vdash (\nu r)P|Q : T; U; L_1 \setminus \{r^+, r^-\} \uplus L_2} \text{(TPARL)}$$

$$\frac{\dfrac{\Gamma, r : [T]; \Theta \vdash P : T; U; L_1 \qquad \Gamma, r : [T]; \Theta \vdash Q : \texttt{end}; \texttt{end}; L_2}{\Gamma, r : [T]; \Theta \vdash P|Q : T; U; L_1 \uplus L_2} \text{(TPARL)}}{\Gamma; \Theta \vdash (\nu r)(P|Q) : T; U; L_1 \setminus \{r^+, r^-\} \uplus L_2} \text{(TNEWR)}$$

In particular, when the proof is from $(\nu m)P|Q$ to $(\nu m)(P|Q)$ we use (to judge Q) the Weakening Lemma, in the other direction instead we use the Strengthening Lemma. The case with (TPARR) when P has type end; end is similar.

- $(\nu m)\mathbf{0} \equiv \mathbf{0}$ we might have $\dfrac{\Gamma, r : [T]; \Theta \vdash \mathbf{0} : \mathtt{end}; \mathtt{end}; \emptyset}{\Gamma; \Theta \vdash (\nu r)\mathbf{0} : \mathtt{end}; \mathtt{end}; \emptyset}\ (\textsc{TNewR})$ if $m = r$ or we might have $\dfrac{\Gamma, a : [T]; \Theta \vdash \mathbf{0} : \mathtt{end}; \mathtt{end}; \emptyset}{\Gamma; \Theta \vdash (\nu a)\mathbf{0} : \mathtt{end}; \mathtt{end}; \emptyset}\ (\textsc{TNew})$ if $m = a$. The direction from $\mathbf{0}$ to $(\nu m)\mathbf{0}$ follows by the Weakening Lemma and the well-formedness of Γ since $\Gamma = \Gamma', r^p : [T]$ iff $\Gamma = \Gamma'', r^{\overline{p}} : [\overline{T}]$.

- $(\nu m)P > \tilde{x} > Q \equiv (\nu m)(P > \tilde{x} > Q)$ if $m \notin \mathsf{fn}(Q) \setminus \{\tilde{x}\}$

$$\dfrac{\dfrac{\Gamma, r : [T']; \emptyset \vdash P : T_1; U_1; L_1}{\Gamma; \emptyset \vdash (\nu r)P : T_1; U_1; L_1 \setminus \{r^+, r^-\}}\ (\textsc{TNewR}) \qquad \Gamma, \tilde{x} : \tilde{S}; \Theta \vdash Q : T_2 : U_2; \emptyset}{\Gamma; \Theta \vdash (\nu r)P > \tilde{x} > Q : T; U; L}\ (\textsc{TPipe})$$

$$\dfrac{\dfrac{\Gamma, r : [T']; \emptyset \vdash P : T_1; U_1; L_1 \qquad \Gamma, r : [T'], \tilde{x} : \tilde{S}; \Theta \vdash Q : T_2 : U_2; \emptyset}{\Gamma, r : [T']; \Theta \vdash P > \tilde{x} > Q : T; U; L_1}\ (\textsc{TPipe})}{\Gamma; \Theta \vdash (\nu r)(P > \tilde{x} > Q) : T; U; L}\ (\textsc{TNewR})$$

where $(T, U) = \mathsf{pipe}(T_1, U_1, T_2, U_2, \tilde{S})$ and $L = L_1 \setminus \{r^+, r^-\}$ and $L_1 \cap \{r^+, r^-\} = \emptyset$ or $L_1 \cap \{r^+, r^-\} = \{r^+, r^-\}$. We apply Strengthening Lemma and Weakening Lemma to type Q in the respective direction.

- $(\nu m)(\nu n)P \equiv (\nu n)(\nu m)Q$ There are four cases depending of the essence of m and n we prove the case where $m = a$ and $n = r$.

$$\dfrac{\dfrac{\Gamma, r : [T_r], a : [T_a]; \Theta \vdash P : T; U; L}{\Gamma, a : [T_a]; \Theta \vdash (\nu r)P : T; U; L'}\ (\textsc{TNewR})}{\Gamma; \Theta \vdash (\nu a)(\nu r)P : T; U; L'}\ (\textsc{TNew})$$

$$\dfrac{\dfrac{\Gamma, r : [T_r], a : [T_a]; \Theta \vdash P : T; U; L}{\Gamma, r : [T_r]; \Theta \vdash (\nu a)P : T; U; L}\ (\textsc{TNew})}{\Gamma; \Theta \vdash (\nu r)(\nu a)P : T; U; L'}\ (\textsc{TNewR})$$

where $L' = L \setminus \{r^+, r^-\}$ and $L \cap \{r^+, r^-\} = \emptyset$ or $L \cap \{r^+, r^-\} = \{r^+, r^-\}$

- $r^p \triangleright (\nu m)P \equiv (\nu m)r^p \triangleright P$ if $m \neq r$. In the case when $m = a$ we have:

$$\cfrac{\cfrac{\Gamma, r : [T], a : S; \Theta \vdash P : T; U; L}{\Gamma, r : [T]; \Theta \vdash (\nu a)P : T; U; L}\ \text{(TNEW)}}{\Gamma, r : [T]; \Theta \vdash r^p \triangleright (\nu a)P : U; \mathtt{end}; L \uplus \{r^p\}}\ \text{(TSES)}$$

$$\cfrac{\cfrac{\Gamma, a : S, r : [T]; \Theta \vdash P : T; U; L}{\Gamma, a : S, r : [T]; \Theta \vdash r^p \triangleright P : U; \mathtt{end}; L \uplus \{r^p\}}\ \text{(TSES)}}{\Gamma, r : [T]; \Theta \vdash (\nu a)r^p \triangleright P : U; \mathtt{end}; L \uplus \{r^p\}}\ \text{(TNEW)}$$

□

In addition, we need auxiliary lemmas to deal with the presence of contexts. The next lemma allows to take P out from a well-typed process $\mathbb{C}[\![P]\!]$. Not only P is well-typed but it can be replaced with Q, a process with the same typing of P, and conclude $\mathbb{C}[\![Q]\!]$ well-typed. Furthermore Q can have more opened sessions than P provided that it contains all the sessions opened in P. This condition is achieved constraining the set of session names.

Lemma 4.12 (Replacement $\mathbb{C}$)**.** *If $\Gamma; \emptyset \vdash \mathbb{C}[\![P]\!] : T; U; L$ then there exist some T',U',L' s.t. $\Gamma; \emptyset \vdash P : T'; U'; L'$ with $L' \subseteq L$. Moreover for any Q, s.t. $\Gamma; \emptyset \vdash Q : T'; U'; L''$ with $L'' \cap L = L'$ then $\Gamma; \emptyset \vdash \mathbb{C}[\![Q]\!] : T; U; L \cup L''$.*

Proof. The first part of the statement is a direct consequence of the compositionality of the type system plus the fact that the set of labels L increases from the premises to the conclusion of each rule. Second part of the statement can be proved by straightforward induction on the structure of the context $\mathbb{C}$. □

As discussed above the linearity of each session is important in order to keep type preservation during reductions. This fact appears just during the proof of the Replacement Lemma for contexts $\mathbb{D}_r$. In turn proving session linearity requires the following lemma about the set L which contains exactly the set of free polarized session names.

Lemma 4.13. *If $\Gamma; \Theta \vdash P : T; U; L$ then $\mathsf{fpn}(P) = L$*

Proof. The proof is by straightforward induction on the typing of $\Gamma; \Theta \vdash P : T; U; L$ with case analysis on the last applied rule. □

Lemma 4.14 (Session linearity)**.** *If $\Gamma; \Theta \vdash \mathbb{C}[\![r^p \triangleright P]\!] : T; U; L$ then $r^p \notin \mathsf{fpn}(\mathbb{C}[\![P]\!])$*

Proof. The proof is by induction on the structure of $\mathbb{C}$ and then on the structure of P. Base case is when $\mathbb{C} = [\![\cdot]\!]$. We report one interesting inductive case. When $\mathbb{C} = \mathbb{C}'[\![[\![\cdot]\!]|P_c]\!]$ and $P = s^q \triangleright P'$ by induction hypothesis the theorem holds for $\mathbb{C}'[\![r^p \triangleright P']\!]$, then $r^p \notin \mathsf{fpn}(\mathbb{C}'[\![P']\!])$. If $\Gamma; \Theta \vdash \mathbb{C}'[\![P']\!] : T; U; L$ by Lemma 4.13, $r^p \notin L$ and by Lemma 4.12, there exist T', U' and L' s.t $\Gamma; \Theta \vdash P' : T; U'; L'$ and $L' \subseteq L$ and $L' \uplus \{r^p\}$ holds. Since the entire process is well-typed by hypothesis, it must be the case (rule (TSES) applied to $r^p \triangleright s^q \triangleright P'$) that $r^p \neq s^q$ (since $L' \uplus \{s^q\} \uplus \{r^q\}$ holds in turn) and then $r^p \notin \mathsf{fpn}(\mathbb{C}'[\![s^q \triangleright P']\!])$. On the other hand since $\mathbb{C}'[\![(r^p \triangleright s^q \triangleright P')|P_c]\!]$ is well-typed, by Lemma 4.12 there exists $L'' \subseteq L$ that type-checks P_c and by rules for parallel composition $L' \uplus \{s^q, r^q\} \uplus L''$ holds and in order not to contradict Lemma 4.13, $r^p \notin \mathsf{fpn}(P_c)$ which concludes. □

We define a relation that captures the evaluation of session types:

Definition 4.15. Let $\subseteq$ denote the smallest relation on $TYPE \times TYPE$ such that: **(1)** $T \subseteq ?(\tilde{S}).T$, **(2)** $T \subseteq !(\tilde{S}).T$, **(3)** $T_i \subseteq \&\{l_1 : T_1, \ldots, l_n : T_n\}$ and **(4)** $T_i \subseteq \oplus\{l_1 : T_1, \ldots, l_n : T_n\}$.

Lemma 4.16 (Replacement $\mathbb{D}_r$). *Let $\Gamma = \Gamma', r^p : [T_r], r^{\overline{p}} : [\overline{T_r}]$. If $\Gamma; \emptyset \vdash \mathbb{D}_r[P, Q] : T; U; L$ then there exist some U_1, $L_1 \subseteq L$ and U_2, $L_2 \subseteq L$ s.t. $L_1 \cap L_2 = \emptyset$ and $\Gamma; \emptyset \vdash P : T_r; U_1; L_1$ and $\Gamma; \emptyset \vdash Q : \overline{T_r}; U_2; L_2$. Moreover for any P' and Q' s.t. $\Gamma_1; \emptyset \vdash P' : T_r^{\subseteq}; U_1; L_1$ and $\Gamma_1; \emptyset \vdash Q' : \overline{T_r^{\subseteq}}; U_2; L_2$ then $\Gamma_1; \emptyset \vdash \mathbb{D}_r[P', Q'] : T; U; L$ for some $T_r^{\subseteq} \subseteq T_r$ and $\Gamma_1 = \Gamma', r^p : [T_r^{\subseteq}], r^{\overline{p}} : [\overline{T_r^{\subseteq}}]$.*

Proof. First part of the statement is a direct consequence of the compositionality of the type system plus the fact that we use disjoint union to compose two sets of label. Second part of the statement can be proved by straightforward induction on the structure of the context $\mathbb{D}_r$ which turns out to be an induction on the structure of $\mathbb{C}[\![\mathbb{C}'_{r^p}|\mathbb{C}''_{r^{\overline{p}}}]\!]$. We sketch the base case and one inductive case, the others are similar. In the base case we have $\mathbb{D}_r[\![\dot{P}, \dot{Q}]\!] = r^p \triangleright (\dot{P}|P)|r^{\overline{p}} \triangleright (\dot{Q}|Q)$ and by hypothesis (or by the first statement) $\Gamma; \emptyset \vdash \dot{P} : T_r; U_1; L_1$ and $\Gamma; \emptyset \vdash P : \mathtt{end}; \mathtt{end}; L'_1$ and $\Gamma; \emptyset \vdash \dot{Q} : \overline{T_r}; U_2; L_2$ and $\Gamma; \emptyset \vdash Q : \mathtt{end}; \mathtt{end}; L'_2$ with $\Gamma = \Gamma', r^p : [T_r], r^{\overline{p}} : [\overline{T_r}]$. Now consider the case when $\Gamma_1; \emptyset \vdash P' : T_r^{\subseteq}; U_1; L_1$ and $\Gamma_1; \emptyset \vdash Q' : \overline{T_r^{\subseteq}}; U_2; L_2$ where $\Gamma_1 = \Gamma', r^p : T_r^{\subseteq}, r^{\overline{p}} : \overline{T_r^{\subseteq}}$. By Lemma 4.13 both $r \notin \mathsf{fn}(P)$ and $r \notin \mathsf{fn}(Q)$. Applying the Strengthening Lemma first and the Weakening lemma then, we can conclude

$\Gamma_1;\emptyset \vdash P : \texttt{end};\texttt{end};L'_1$ and $\Gamma_1;\emptyset \vdash Q : \texttt{end};\texttt{end};L'_2$. The thesis follows applying rules (TPARR) and (TSES) on both sides and then one of the two rules for parallel composition depending whose between U_1 and U_2 has type `end`. The inductive cases are mixed cases one for each production of contexts $\mathbb{C}$, $\mathbb{C}'_{r^p}$ and $\mathbb{C}''_{r^{\overline{p}}}$ used to build $\mathbb{D}_r$. Consider for example the case where $\mathbb{D}_r[\![\dot{P},\dot{Q}]\!] = (\mathbb{C}'_1[\![s^q \triangleright r^p \triangleright (\dot{P}|P)]\!]|(\mathbb{C}''_1[\![r^{\overline{p}} \triangleright (\dot{Q}|Q)|Q_1]\!]))$ where $\mathbb{C} = [\![\cdot]\!]$, $\mathbb{C}' = \mathbb{C}'_1[\![s^q \triangleright [\![\cdot]\!]]\!]$ and $\mathbb{C}'' = \mathbb{C}''_1[\![[\![\cdot]\!]|Q_1]\!]$. By induction hypothesis the thesis holds for $\mathbb{C}'_1[\![r^p \triangleright (\dot{P}|P)]\!]|\mathbb{C}''_1[\![r^{\overline{p}} \triangleright (\dot{Q}|Q)]\!]$. The result follows since replacing $\dot{P}$ with $\dot{P}'$ and its environment not influence the typing of session s (*in fact* $s \neq r$ *by Lemma 4.14*) and replacing $\dot{Q}$ with $\dot{Q}'$ not influence the applicability of rules for parallel composition provided that we use Lemma 4.14 together with Strengthening and Weakening lemmas to type Q_1 with the new assumptions. □

Finally we are now ready to prove the Subject Reduction Theorem. We prove the Subject reduction for closed processes (i.e. $\Theta = \emptyset$) and we require the set of sessions L to be balanced. A set L is balanced if $r^p \in L$ implies $r^{\overline{p}} \in L$. This condition is necessary since we require that a process must always have either a session with both dual polarities or no session at all.

Theorem 4.17 (Subject reduction)**.** *If* $\Gamma;\emptyset \vdash P : T;U;L$ *with* L *balanced and* $P \to P'$ *then* $\Gamma';\emptyset \vdash P' : T;U;L$ *for some* Γ' *s.t.* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ *and* $a \in \mathsf{dom}(\Gamma)$ *implies* $\Gamma(a) = \Gamma'(a)$.

Proof. The proof is by induction on the derivation of $P \to P'$ with case analysis on the last applied rule.

- Rule (INV) we have that $\mathbb{D}[\![\overline{a}.P, a.Q]\!]$ is well-typed by hypothesis. Applying Lemma 4.12 on $\mathbb{D}$ it must be the case that $\Gamma;\emptyset \vdash \mathbb{C}'[\![\overline{a}.P]\!]|\mathbb{C}''[\![a.Q]\!] : T_1;U_1;L_1^{\subseteq} \uplus L_2^{\subseteq}$. Two other applications of Lemma 4.12 give, with say $\Gamma \vdash a : [T'']$,

$$\text{(TINV)}\ \frac{\Gamma;\emptyset \vdash P : \overline{T''};U'_1;L_1}{\Gamma;\emptyset \vdash \overline{a}.P : U'_1;\texttt{end};L_1} \qquad \text{(TDEF)}\ \frac{\Gamma;\emptyset \vdash Q : T'';U'_2;L_2}{\Gamma;\emptyset \vdash a.Q : U'_2;\texttt{end};L_2}$$

  To type the result obtained after the reduction step (INV), we can apply the Weakening Lemma (thanks to the side condition on r in the rule (INV)).

$$\text{(TSES)}\ \frac{\Gamma'_1, r : [T''];\emptyset \vdash P : \overline{T''};U'_1;L_1}{\Gamma'_1, r : [T''];\emptyset \vdash r^- \triangleright P : U'_1;\texttt{end};L_1 \uplus \{r^-\}}$$

$$\frac{\Gamma'_2, r : [T'']; \emptyset \vdash Q : T''; U'_2; L_2}{\Gamma'_2, r : [T'']; \emptyset \vdash r^+ \rhd Q : U'_2; \mathtt{end}; L_2 \uplus \{r^+\}} \text{(TSES)}$$

The result follows applying Lemma 4.12 twice, to obtain $\Gamma; \emptyset \vdash \mathbb{C}'[\![r^- \rhd P]\!]|\mathbb{C}''[\![r^+ \rhd Q]\!] : T_1; U_1; L_1^{\subseteq} \uplus \{r^-\} \uplus L_2^{\subseteq} \uplus \{r^+\}$ since the condition of freshness of r gives $\{r^+, r^-\} \cap \mathsf{fpn}(\mathbb{C}'[\![\overline{a}.P]\!]|\mathbb{C}''[\![a.Q]\!]) = \emptyset$ and Lemma 4.13 implies sets of label disjoint with r. To conclude it suffices to apply Lemma 4.12 another time and rule (TNEWR).

- (COM) We have that $\mathbb{D}_r[\![(\tilde{x}).P, \langle\tilde{v}\rangle.Q]\!]$ is well-typed by hypothesis. Applying Lemma 4.16 on the context $\mathbb{D}_r$ it must be the case that $\frac{\Gamma, \tilde{x} : \tilde{S}; \emptyset \vdash P : T'_1; U'_1; L_1}{\Gamma; \emptyset \vdash (\tilde{x}).P : ?(\tilde{S}).T'_1; U'_1; L_1}\text{(TIN)}$ and $\frac{\Gamma; \emptyset \vdash Q : \overline{T'_1}; U'_2; L_2}{\Gamma; \emptyset \vdash \langle\tilde{v}\rangle.Q : \overline{!(\tilde{S}).T'_1}; U'_2; L_2}\text{(TOUT)}$. To type the result obtained after the reduction step (COM), we can apply the Substitution Lemma to obtain: $\Gamma; \emptyset \vdash P[^{\tilde{v}}/_{\tilde{x}}] : T'_1; U'_1; L_1$ and another application of Lemma 4.16 to conclude.

- Rule (LCOM) we have that $\mathbb{D}_r[\![\Sigma_{i=1}^n (l_i).P_i, \langle l_k\rangle.Q]\!]$ is well-typed by hypothesis. Applying Lemma 4.16 on the context $\mathbb{D}_r$ it must be the case that:
$$\frac{\emptyset \subset J \subseteq \{1, \ldots, n\} \quad \forall i\ \Gamma; \emptyset \vdash P_i : T_i; U'_1; L_1}{\Gamma; \emptyset \vdash \Sigma_{i=1}^n (l_i).P_i : \&\{l_j : T_j\}_{j \in J}; U'_1; L_1}\text{(TBRANCH)}$$
$$\frac{\Gamma; \emptyset \vdash Q : \overline{T_k}; U'_2; L_2}{\Gamma; \emptyset \vdash \langle l_k\rangle.Q : \oplus\{l_1 : \overline{T_1}, \ldots, l_n : \overline{T_n}\}; U'_2; L_2}\text{(TCHOICE)}$$
The process resulting after the reduction step (LCOM) is the typing judgment obtained from P_k and Q and an application of Lemma 4.16.

- Rule (RET) we have that $\mathbb{D}_{r_1}[\![(\tilde{x}).P, \mathbb{C}_{r^q}[\![\mathtt{return}\ \tilde{v}.Q]\!]]\!]$ is well-typed by hypothesis. Applying Lemma 4.16 on the context $\mathbb{D}_{r_1}$ it must be the case that: $\frac{\Gamma, \tilde{x} : \tilde{S}; \emptyset \vdash P : T'_1; U'_1; L_1}{\Gamma; \emptyset \vdash (\tilde{x}).P : ?(\tilde{S}).T'_1; U'_1; L_1}\text{(TIN)}$ $\frac{\Gamma; \emptyset \vdash \mathbb{C}_{r^q}[\![Q]\!] : \overline{T'_1}; \mathtt{end}; L_2}{\Gamma; \emptyset \vdash \mathbb{C}_{r^q}[\![\mathtt{return}\ \tilde{v}.Q]\!] : \overline{?(\tilde{S}).T'_1}; \mathtt{end}; L_2}\text{(TRET-TPARL-TSES)}$. The result follows applying Lemma 4.16 on both premises.

- Rule (PIPE). The result follows by Lemma 4.12 if we show that $(\langle\tilde{v}\rangle.P|P') > \tilde{x} > Q$ and $(P|P') > \tilde{x} > Q \mid Q[^{\tilde{v}}/_{\tilde{x}}]$ have the same type. Let $L_1 = L'_1 \uplus L''_1$, the former is typed as

$$\dfrac{\dfrac{\dfrac{\Gamma;\emptyset \vdash P : \mathtt{end};\mathtt{end};L'_1}{\Gamma;\emptyset\vdash \langle\tilde v\rangle.P :!(\tilde S);\mathtt{end};L'_1}\,(\textsc{TOut}) \quad \Gamma;\emptyset\vdash P' : \mathtt{end};\mathtt{end};L''_1}{\Gamma;\emptyset\vdash\langle\tilde v\rangle.P|P' :!(\tilde S);\mathtt{end};L_1}\,(\textsc{TParL}) \quad \Gamma,\tilde x:\tilde s;\emptyset\vdash Q : T'_1;U'_2;\emptyset}{\Gamma;\emptyset\vdash(\langle\tilde v\rangle.P|P')>\tilde x>Q : T'_1;U'_2;L_1}\,(\textsc{TPipe})$$

while the latter (with the help of Substitution Lemma) is typed as

$$\dfrac{\dfrac{\dfrac{\Gamma;\emptyset \vdash P : \mathtt{end};\mathtt{end};L'_1\;\Gamma;\emptyset\vdash P' : \mathtt{end};\mathtt{end};L''_1}{\Gamma;\emptyset\vdash P|P' : \mathtt{end};\mathtt{end};L_1} \quad \Gamma,\tilde x:\tilde s;\emptyset\vdash Q : T'_1;U'_2;\emptyset}{\Gamma;\emptyset\vdash(P|P')>\tilde x>Q : \mathtt{end};\mathtt{end};L_1}\,(\textsc{TPipe}) \quad \Gamma;\emptyset\vdash Q[{}^{\tilde v}/_{\tilde x}] : T'_1;U'_2;\emptyset}{\Gamma;\emptyset\vdash(P|P')>\tilde x>Q \mid Q[{}^{\tilde v}/_{\tilde x}] : T'_1;U'_2;L_1}\,(\textsc{TParR})$$

- Rule (PipeRet). Similarly to the previous case we have that both $\mathbb{C}[\![(\mathbb{C}_{rp}[\![\mathtt{return}\;\tilde v.P]\!]|P')>\tilde x>Q]\!]$ and $\mathbb{C}[\![(\mathbb{C}_{rp}[\![P]\!]|P')>\tilde x>Q \mid Q[{}^{\tilde v}/_{\tilde x}]]\!]$ must have equal type. Let $L_1 = L'_1 \uplus L''_1$, the former is typed as

$$\dfrac{\dfrac{\dfrac{\Gamma;\emptyset \vdash P : \mathtt{end};\mathtt{end};L'_1}{\Gamma;\emptyset\vdash \mathbb{C}_{rp}[\![\mathtt{return}\;\tilde v.P]\!] :!(\tilde S);\mathtt{end};L'_1}\,(\textsc{T}\star) \quad \Gamma;\emptyset\vdash P' : \mathtt{end};\mathtt{end};L''_1}{\Gamma;\emptyset\vdash\mathbb{C}_{rp}[\![\mathtt{return}\;\tilde v.P]\!]|P' :!(\tilde S);\mathtt{end};L_1}\,(\textsc{TParL}) \quad \Gamma,\tilde x:\tilde s;\emptyset\vdash Q : T'_1;U'_2;\emptyset}{\Gamma;\emptyset\vdash(\mathbb{C}_{rp}[\![\mathtt{return}\;\tilde v.P]\!]|P')>\tilde x>Q : T'_1;U'_2;L_1}\,(\textsc{TPipe})$$

  where (T$\star$) stays for the application of (TRet),(TParL) and (TSes). To type the result obtained after the reduction we use Substitution Lemma and we have:

$$\dfrac{\dfrac{\dfrac{\Gamma;\emptyset \vdash \mathbb{C}_{rp}[\![P]\!] : \mathtt{end};\mathtt{end};L'_1\;\Gamma;\emptyset\vdash P' : \mathtt{end};\mathtt{end};L''_1}{\Gamma;\emptyset\vdash \mathbb{C}_{rp}[\![P]\!]|P' : \mathtt{end};\mathtt{end};L_1} \quad \Gamma,\tilde x:\tilde s;\emptyset\vdash Q : T'_1;U'_2;\emptyset}{\Gamma;\emptyset\vdash(\mathbb{C}_{rp}[\![P]\!]|P')>\tilde x>Q : \mathtt{end};\mathtt{end};L_1}\,(\textsc{TPipe})}{\star}$$

$$\dfrac{\star \qquad \Gamma;\emptyset\vdash Q[{}^{\tilde v}/_{\tilde x}] : T'_1;U'_2;\emptyset}{\Gamma;\emptyset\vdash(\mathbb{C}_{rp}[\![P]\!]|P')>\tilde x>Q \mid Q[{}^{\tilde v}/_{\tilde x}] : T'_1;U'_2;L_1}\,(\textsc{TParR})$$

- Rules (IfT) and (IfF) follows directly by the application of the induction hypothesis and the fact that (TIf) judges P and Q in the same manner.

- Rule (REC). The proof of this case might be concluded by a straightforward application of the inductive hypothesis since we have P' in both the premise and in the conclusion of the rule. However, to apply the induction we must prove that $P[^{\texttt{rec}\ X.P}/_X]$ is well-typed and in order to apply Lemma 4.12 the process must by judged in the same manner as $\texttt{rec}\ X.P$. By hypothesis and Lemma 4.12 we know that $\texttt{rec}\ X.P$ is well-typed and its typing should be derived with $\dfrac{\Gamma; X : T; U \vdash P : T; U; \emptyset}{\Gamma; \emptyset \vdash \texttt{rec}\ X.P : T; U; \emptyset}$ (TREC). Since both X and $\texttt{rec}\ X.P$ satisfy the hypotheses of the Substitution Lemma for process variables applied to P we have $\Gamma; \emptyset \vdash P[^{\texttt{rec}\ X.P}/_X] : T; U; \emptyset$ which is the desiderated judgment.

- Rule (SCOP). By induction we know that $P \rightarrow P'$ and $\Gamma; \emptyset \vdash P : T; U; L$ and L balanced implies $\Gamma'; \emptyset \vdash P' : T; U; L$. As usual we have two cases depending of m. If $m = a$ we can conclude by inductive hypothesis since $\Gamma(a) = \Gamma'(a)$ and rule (TNEW), if $m = r$ we can conclude with rule (TNEWR) since $\{r^+, r^-\} \subseteq L$ or $L \cap \{r^+, r^-\} = \emptyset$.

- Rule (STR) Follows by induction and by Subject Congruence Lemma.

□

Typing processes with session types gives an immediate consequence about the goodness of a process. Following (37) an error is a process which has incompatible kinds of communications in a dual sessions e.g. $\mathbb{D}_r[\![(\tilde{x}).P', (\tilde{x}).P'']\!]$, $\mathbb{D}[\![(\tilde{x}).P', \Sigma_{i=1}^{n}(l_i).P_i]\!]$, $\mathbb{D}[\![(\tilde{x}).P', \langle l \rangle.P'']\!]$ and so on for the other mismatching prefixes. A process P has an error if $P \equiv (\nu\tilde{m})\mathbb{D}[\![P', P'']\!]$ and $\mathbb{D}[\![P', P'']\!]$ is an error.

Theorem 4.18 (Safety). *Let $\Gamma, \emptyset \vdash P : T; U; L$ then P never reduces to a process with an error.*

Proof. By Subject Reduction it suffices to show that typable processes are not errors. The proof is by reductio ad absurdum, assuming error processes typable. Suppose for example $\Gamma, r : [T_r]; \emptyset \vdash \mathbb{D}_r[\![(\tilde{x}).P', (\tilde{x}).P'']\!] : T; U; L$. By Lemma 4.16 we have $\Gamma; \emptyset \vdash (\tilde{x}).P' : T_r; U_1; L_1$ for some U_1, L_1 and $\Gamma; \emptyset \vdash (\tilde{x}).P' : \overline{T_r}; U_2; L_2$ for U_2, L_2. But it cannot be $\overline{T_r} = T_r$ unless $T_r \leqslant \texttt{end}$ which is not the case since T_r is typed with the rule (TIN). □

The type system here proposed cannot be exploited directly for type inference, because three main sources of non-determinism that would

require guesses and backtracking: the rule for choices (TBRANCH) and (TCHOICE), the rule for recursion (TREC) and rule (TWEAK). In the next section we propose a first naive attempt to get a syntax directed type system.

4.4 A set based type system

In this section we introduce a syntax-directed type system to overcome the above mentioned sources of non determinism. The typing judgements in Figure 20 have the form $\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ where types in calligraphic: $\mathcal{T}, \mathcal{U}$ are sets of (possibly not closed) session types.

The basic intuitions behind this type system are two. First, the set $\mathcal{T}$ is obtained multiplexing each branch of the if-then-else (see rule (TIFSD)) or each branch of an external choice labeled with equal labels (see rule (TBRANCHSD)). Second recursion is typed using a type variable with the name dependent of the current process variable, to mean that its type is unknown (see rules (TRECSD)).

Rules (TBRANCHSD) and (TCHOICESD) are now fully syntax directed but rules (TDEFSD) and (TINVSD) require a little care. In particular, we use the subtyping relation to relax the service protocol w.r.t. the actual protocol specified in the process. The subtyping relation between sets and type is defined as expected in Figure 21. Note that the subtyping check is inverted in rule (TINVSD) due to the property in Proposition 3.6, but it is not related to the contravariance (63) of output (not present here because we require the syntactic equality of I/O types). Let us focus on the rule (TRECSD) for recursive processes. The type of the process P is computed by means of a type variable inserted in Θ. The resulting type is computed with the help of the function $\mu\alpha(T)$ (see Figure 21) which checks the possibility to close the variable α in the type T. We have three possibilities depending on whether α is free in T and the resulting type $\mu\alpha.T$ is contractive.

Some other auxiliary functions used in the type system are also in Figure 21. They are all simple (homomorphic extension of the operators in the type signature defined by pointwise application) but same, which deserves some explanations. Consider for instance the Example 4.1, where we use the same label l to offer an external choice. While the previous type system guesses automatically the common type of each branch with the same label (it is in some manner forced by the well-formedness of session types), here in a syntax direct type system we must exhibit such type. The idea is to treat branches labeled with the same label, like an

(TNEWRSD)

$$\frac{\Gamma, r^+ : [T], r^- : [\overline{T}]; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad L \cap \{r^+, r^-\} \neq 1}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\nu r)P : \mathcal{T}; \mathcal{U}; L \setminus \{r^+, r^-\}}$$

(TIFSD)

$$\frac{\Gamma \vdash v_i : S \quad i = 1, 2 \quad \Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}_1; \mathcal{U}_1; L \quad \Gamma; \Theta \vdash_{\mathsf{sd}} Q : \mathcal{T}_2; \mathcal{U}_2; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} \texttt{if } v_1 = v_2 \texttt{ then } P \texttt{ else } Q : \mathcal{T}_1 \cup \mathcal{T}_2; \mathcal{U}_1 \cup \mathcal{U}_2; L}$$

(TNEWSD)

$$\frac{\Gamma, a : S; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\nu a)P : \mathcal{T}; \mathcal{U}; L}$$

(TRECSD)

$$\frac{\Gamma; \Theta, X : \alpha_X; \alpha_X \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; \emptyset}{\Gamma; \Theta \vdash_{\mathsf{sd}} \texttt{rec } X.P : \mu\alpha_X(\mathcal{T}); \mu\alpha_X(U); \emptyset}$$

(TPVARSD)

$$\Gamma; \Theta, X : T; U \vdash_{\mathsf{sd}} X : \{T\}; \{U\}; \emptyset$$

(TSESSD)

$$\frac{\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad \Gamma \vdash r^p : [T] \quad T \leq \mathcal{T}}{\Gamma; \Theta \vdash_{\mathsf{sd}} r^p \triangleright P : \mathcal{U}; \{\mathsf{end}\}; L \uplus \{r^p\}}$$

(TDEFSD)

$$\frac{\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad \Gamma \vdash a : [T] \quad T \leq \mathcal{T}}{\Gamma; \Theta \vdash_{\mathsf{sd}} a.P : \mathcal{U}; \{\mathsf{end}\}; L}$$

(TINVSD)

$$\frac{\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad \Gamma \vdash v : [T] \quad \overline{\mathcal{T}} \leq T}{\Gamma; \Theta \vdash_{\mathsf{sd}} \overline{v}.P : \mathcal{U}; \{\mathsf{end}\}; L}$$

(TINSD)

$$\frac{\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\tilde{x}).P : ?(\tilde{S}).\mathcal{T}; \mathcal{U}; L}$$

(TOUTSD)

$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma; \Theta \vdash_{\mathsf{sd}} \langle \tilde{v} \rangle.P : !(\tilde{S}).\mathcal{T}; \mathcal{U}; L}$$

(TBRANCHSD)

$$\frac{I = \{1, \ldots, n\} \quad \forall i \in I,\ \Gamma; \Theta \vdash_{\mathsf{sd}} P_i : \mathcal{T}_i; \mathcal{U}_i; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} \Sigma_{i=1}^n (l_i).P_i : \&\{\mathsf{same}((l_i : \mathcal{T}_i)_{i \in I})\}; \bigcup_{i \in I} \mathcal{U}_i; L}$$

(TRETSD)

$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma; \Theta \vdash_{\mathsf{sd}} \texttt{return } \tilde{v}.P : \mathcal{T}; !(\tilde{S}).\mathcal{U}; L}$$

(TCHOICESD)

$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} \langle l \rangle.P : \oplus\{l : \mathcal{T}\}; \mathcal{U}; L}$$

(TPARLSD)

$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L_1 \quad \Gamma; \Theta \vdash_{\mathsf{sd}} Q : \{\mathsf{end}\}; \{\mathsf{end}\}; L_2}{\Gamma; \Theta \vdash_{\mathsf{sd}} P | Q : \mathcal{T}; \mathcal{U}; L_1 \uplus L_2}$$

(TPARRSD)

$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : \{\mathsf{end}\}; \{\mathsf{end}\}; L_1 \quad \Gamma; \Theta \vdash_{\mathsf{sd}} Q : \mathcal{T}; \mathcal{U}; L_2}{\Gamma; \Theta \vdash_{\mathsf{sd}} P | Q : \mathcal{T}; \mathcal{U}; L_1 \uplus L_2}$$

(TPARLRSD)

$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}_1; \mathcal{U}_1; L_1 \quad \Gamma; \Theta \vdash_{\mathsf{sd}} Q : \mathcal{T}_2; \mathcal{U}_2; L_2}{\Gamma; \Theta \vdash_{\mathsf{sd}} P | Q : \mathcal{T}_1 \cup \mathcal{T}_2 \cup \{\mathsf{end}\}; \mathcal{U}_1 \cup \mathcal{U}_2 \cup \{\mathsf{end}\}; L_1 \uplus L_2}$$

(TNILSD)

$$\Gamma; \Theta \vdash \mathbf{0} : \{\mathsf{end}\}; \{\mathsf{end}\}; \emptyset$$

(TPIPESD)

$$\frac{\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \{T_1\}; \{\mathsf{end}\}; L \quad \Gamma, \tilde{x} : \tilde{S}; \Theta \vdash_{\mathsf{sd}} Q : \mathcal{T}_2; \mathcal{U}_2; \emptyset \quad (\mathcal{T}, \mathcal{U}) = \mathsf{pipe}(T_1, \mathcal{T}_2, \mathcal{U}_2, \tilde{S})}{\Gamma; \Theta \vdash_{\mathsf{sd}} P > \tilde{x} > Q : \mathcal{T}; \mathcal{U}; L}$$

Figure 20: Syntax directed typing rules

$$\diamondsuit\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\} = \{\diamondsuit\{l_1 : T_1, \ldots l_n : T_n\} \mid i \in \{1, \ldots, n\} \text{ and } T_i \in \mathcal{T}_i\}$$
$$!(\tilde{S}).\mathcal{T} = \{!(\tilde{S}).T \mid T \in \mathcal{T}\} \qquad ?(\tilde{S}).\mathcal{T} = \{?(\tilde{S}).T \mid T \in \mathcal{T}\}$$
$$\mu\alpha(\mathcal{T}) = \{\mu\alpha(T) \mid T \in \mathcal{T}\}$$
$$\mathcal{T} \leq T \;\Leftrightarrow\; \forall T' \in \mathcal{T}.T' \leq T \qquad T \leq \overline{\mathcal{T}} \;\Leftrightarrow\; \forall T' \in \mathcal{T}.T \leq \overline{T'}$$

$$\mu\alpha(T) = \begin{cases} \mu\alpha.T & \text{if} \quad \alpha \in \mathsf{fv}(T) \wedge \mu\alpha.T \text{ is contractive} \\ \texttt{end} & \text{if} \quad \alpha \in \mathsf{fv}(T) \wedge \mu\alpha.T \text{ is not contractive} \\ T & \qquad \text{otherwise} \end{cases}$$

$$\&\{l : T\} \,;;\, \&\{l_1 : T_1, \ldots, l_n : T_n\} = \begin{cases} \&\{l_1 : T_1, \ldots, l : T, \ldots, l_n : T_n\} & \text{if } l \notin \{l_1, \ldots, l_n\} \\ \textit{undefined} \text{ otherwise} \end{cases}$$

$$\mathsf{same}(l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n) = \begin{cases} l_1 : \mathcal{T}_1, \mathsf{same}(l_2 : \mathcal{T}_2 \ldots, l_n : \mathcal{T}_n)) & \text{if } l_1 \notin \{l_2, \ldots, l_n\} \\ \mathsf{same}(l_2 : \mathcal{T}_2, \ldots, l_i : \mathcal{T}_1 \cup \mathcal{T}_i, \ldots, l_n : \mathcal{T}_n) & \text{if } l_1 = l_i \end{cases}$$
$$\mathsf{same}(l : \mathcal{T}) = l : \mathcal{T}$$

$$\mathsf{fv}(\mathcal{T}) = \bigcup_{T' \in \mathcal{T}} \mathsf{fv}(T')$$

Figure 21: Auxiliary functions for sets of types (where $\diamondsuit \in \{\&, \oplus\}$)

if-the-else and then to create a unique label but with the union of the types resulting from these branches. Function same is used together with $\diamondsuit\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\}$ which produces a set of choices taking the cartesian product of each $\mathcal{T}_i$. For example $\&\{l_1 : \{T_1, U_1\}, l_2 : \{T_2\}\}$ is the set $\mathcal{T} = \{\&\{l_1 : T_1, l_2 : T_2\}, \&\{l_1 : U_1, l_2 : T_2\}\}$ and $\&\{\mathsf{same}(l_1 : T_1, l_1 : U_1, l_2 : T_2)\}$ is equal to $\mathcal{T}$.

In this type system we also have three rules for parallel composition, rules (TPARLSD) and (TPARRSD) are similar to the respective counterparts in $\vdash$. Remaining rule (TPARLRSD) allows to compose a process even if neither $\mathcal{T}_1$ and $\mathcal{U}_1$ nor $\mathcal{T}_2$ and $\mathcal{U}_2$ have type $\{\mathtt{end}\}$. Consider for example the process $P = \mathtt{rec}\ X.\mathtt{rec}\ Y.(X|Y)$ the process is typable in $\vdash$ with $\emptyset;\emptyset \vdash P : \mathtt{end};\mathtt{end};\emptyset$ but here after applying rule (TPVARSD) for both X and Y we are stalled since neither rule among (TPARLSD) and (TPARRSD) is applicable. Rule (TPARLRSD) allows typing $X|Y$ but in the conclusion it adds the type $\mathtt{end}$ to mean that the resulting type of both branches must have something in common with $\mathtt{end}$ (see Theorem 4.30). Moreover, rule (TPARLRSD) has an implicit premise, $\mathcal{T}_1$ and $\mathcal{U}_1$ or $\mathcal{T}_2$ and $\mathcal{U}_2$ must be different from $\{\mathtt{end}\}$, i.e. (TPARLSD) and (TPARRSD) should be applied if possible.

Arbitrary typing judgments $\Gamma;\Theta \vdash_{\mathsf{sd}} P : \mathcal{T};\mathcal{U};L$ are not correct since sets of types are intended to group type with a common behavior. Later we shall prove that a judgment is correct if there exists the greatest lower bound of $\mathcal{T}$ and all elements in $\mathcal{U}$ are in $\leqslant$ relation.

Moreover, due some limitations of the type system we are only able to prove the soundness, (later we introduce another type system sound and complete with respect to judgments in $\vdash$). The limit of the type system is that it does not support recursion under service invocation, service definition and sessions, in fact rules (TDEF), (TINV) and (TSES) judge the body P with $\emptyset$ as process typing environment. This limitation is due to the fact that we need to compute $\leq$ in these rules but if $\mathcal{T}$ contains some type variables, inserted by rule (TPVARSD), then the subtyping is not defined. The other limitation is due to the fact that (TDEFSD) allows service declaration of the form $a.P$ (*not* a variable). This limitation is not strictly related to the type system but it allows a simple and terminating constraint solving algorithm.

Example 4.19. The first instance of service a in Example 4.1 is typed as below, where $\mathcal{T} = \&\{\mathsf{same}(l : \{\oplus\{l_1 : \alpha_{Y_2}\}, l : \oplus\{l_2 : \alpha_{Y_2}\}\})\} = \{\&\{l : \oplus\{l_1 : \alpha_{Y_2}\}\}, \&\{l : \oplus\{l_2 : \alpha_{Y_2}\}\}\}$, $\Theta = \alpha_{Y_2};\alpha_{Y_2}$, $\Gamma = a : [\mu\alpha.\&\{l : \oplus\{l_1 : \alpha, l_2 : \alpha, l_3 : \alpha\}\}]$ and the curly brackets are omitted when the set is a singleton:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma;\Theta \vdash_{\mathsf{sd}} Y_2 : \alpha_{Y_2};\mathtt{end}}{\Gamma;\Theta \vdash_{\mathsf{sd}} \langle l_1\rangle.Y_2 : \oplus\{l_1 : \alpha_{Y_2}\};\mathtt{end}}}{\Gamma;\Theta \vdash_{\mathsf{sd}} (l).\langle l_1\rangle.Y_2 : \&\{l : \oplus\{l_1 : \alpha_{Y_2}\}\};\mathtt{end}} \quad \dfrac{\dfrac{\Gamma;\Theta \vdash_{\mathsf{sd}} Y_2 : \alpha_{Y_2};\mathtt{end}}{\Gamma;\Theta \vdash_{\mathsf{sd}} \langle l_2\rangle.Y_2 : \oplus\{l_2 : \alpha_{Y_2}\};\mathtt{end}}}{\Gamma;\Theta \vdash_{\mathsf{sd}} (l).\langle l_2\rangle.Y_2 : \&\{l : \oplus\{l_2 : \alpha_{Y_2}\}\};\mathtt{end}}}{\Gamma;\Theta \vdash_{\mathsf{sd}} \Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2 : \mathcal{T};\mathtt{end}}}{\Gamma;\emptyset \vdash_{\mathsf{sd}} \mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2 : \mu\alpha_{Y_2}.\mathcal{T};\mathtt{end}}}{\Gamma;\emptyset \vdash_{\mathsf{sd}} a.\mathtt{rec}\ Y_2.\Sigma_{i=1}^{2}(l).\langle l_i\rangle.Y_2 : \mathtt{end};\mathtt{end}}$$

Example 4.20. P from Example 4.2 is typed in the same way as in $\vdash$ since neither recursion nor branching are present.

4.4.1 Soundness

To account for the free variables present in a type during the typing derivation (rule (TPVARSD)) we introduce some auxiliary syntax used in the proofs of theorems, called the $\bullet$-machinery. The idea is that the special symbol $\bullet$ acting like end replaces a free variable in the type. First of all we add a production for $\bullet$, both as a process and as a session type with the two obvious axioms $\Gamma; \Theta \vdash \bullet : \bullet; \bullet; \emptyset$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} \bullet : \{\bullet\}; \{\bullet\}; \emptyset$. Moreover we add a rule in the subtyping relation which asserts that "if $\mathsf{unfold}(T) = \bullet$ then $\mathsf{unfold}(V) = \bullet$", e.g. $\bullet \leq \bullet$. The next lemma is a very interesting property of the $\bullet$-machinery: if two types are in subtype relation then infinite copies of these types are in subtype relation.

Lemma 4.21. *If* $T_1[^{\bullet}/_{\alpha_1}] \leq T_2[^{\bullet}/_{\alpha_2}]$ *then* $\mu\alpha_1.T_1 \leq \mu\alpha_2.T_2$.

Proof. It suffices to prove that $R = \{(T_1[^{\mu\alpha_1.V_1}/_{\alpha_1}], T_2[^{\mu\alpha_2.V_2}/_{\alpha_2}]) | T_1[^{\bullet}/_{\alpha_1}] \leq T_2[^{\bullet}/_{\alpha_2}]$ and $V_1[^{\bullet}/_{\alpha_1}] \leq V_2[^{\bullet}/_{\alpha_2}]\}$ is a type simulation relation, which contains both $\mu\alpha_1.T_1$ and $\mu\alpha_2.T_2$ since $(T_1[^{\mu\alpha_1.T_1}/_{\alpha_1}], T_2[^{\mu\alpha_2.T_2}/_{\alpha_2}]) \in R$. We prove the case when $\mathsf{unfold}(T_1[^{\mu\alpha_1.V_1}/_{\alpha_1}])$ is an input. We have two cases $\mathsf{unfold}(T_1[^{\mu\alpha_1.V_1}/_{\alpha_1}]) = ?(\tilde{S}).T_1'[^{\mu\alpha_1.V_1}/_{\alpha_1}]$ or the case when $\mathsf{unfold}(T_1[^{\mu\alpha_1.V_1}/_{\alpha_1}]) = ?(\tilde{S}).V_1'[^{\mu\alpha_1.V_1}/_{\alpha_1}]$ and $\mathsf{unfold}(T_1) = \alpha_1$ and $\mathsf{unfold}(\mu\alpha_1.V_1) = ?(\tilde{S}).V_1'[^{\mu\alpha_1.V_1}/_{\alpha_1}]$. In the first case we have by definition of $T_1[^{\bullet}/_{\alpha_1}] \leq T_2[^{\bullet}/_{\alpha_2}]$ that $\mathsf{unfold}(T_1[^{\bullet}/_{\alpha_1}]) = ?(\tilde{S}).T_1'[^{\bullet}/_{\alpha_1}]$ and $\mathsf{unfold}(T_2[^{\bullet}/_{\alpha_2}]) = ?(\tilde{S}).T_2'[^{\bullet}/_{\alpha_2}]$ with $T_1'[^{\bullet}/_{\alpha_1}] \leq T_2'[^{\bullet}/_{\alpha_2}]$ which concludes since $(T_1'[^{\mu\alpha_1.V_1}/_{\alpha_1}], T_2'[^{\mu\alpha_2.V_2}/_{\alpha_2}]) \in R$. In the other case when $\mathsf{unfold}(T_1) = \alpha_1$ we have $\mathsf{unfold}(T_1[^{\bullet}/_{\alpha_1}]) = \bullet$ and by definition of $T_1[^{\bullet}/_{\alpha_1}] \leq T_2[^{\bullet}/_{\alpha_2}]$, $T_2[^{\bullet}/_{\alpha_2}] = \bullet$. By definition of $V_1[^{\bullet}/_{\alpha_1}] \leq V_2[^{\bullet}/_{\alpha_2}]$ since $\mathsf{unfold}(V_1[^{\bullet}/_{\alpha_1}]) = ?(\tilde{S}).V_1'[^{\bullet}/_{\alpha_1}]$ it must be the case that $\mathsf{unfold}(V_2[^{\bullet}/_{\alpha_2}]) = ?(\tilde{S}).V_2'[^{\bullet}/_{\alpha_2}]$ with $V_1'[^{\bullet}/_{\alpha_1}] \leq V_2'[^{\bullet}/_{\alpha_2}]$ which concludes since $(V_1'[^{\mu\alpha_1.V_1}/_{\alpha_1}], V_2'[^{\mu\alpha_2.V_2}/_{\alpha_2}]) \in R$. The case for output and choices are similar.

□

The converse of the previous lemma is a bit tricky, if two infinite types are in subtyping relation then it is possible to find a finite representation for both types s.t. these finite types are in subtyping relation.

Lemma 4.22. *If* $\mu\alpha.T_1 \leq \mu\alpha.T_2$ *then there exist* $\mu\alpha.T_1', \mu\alpha.T_2'$ *s.t.* $\mu\alpha.T_1 \lesseqgtr \mu\alpha.T_1'$ *and* $\mu\alpha.T_2 \lesseqgtr \mu\alpha.T_2'$ *and* $T_1'[^\bullet/_\alpha] \leq T_2'[^\bullet/_\alpha]$.

Proof. The proof is rather long but simple. The idea is to compute $\emptyset \vdash \mu\alpha.T_1 \overset{c}{\wedge} \mu\alpha.T_2 \rightarrow \mu\alpha.T_1' \triangleright \emptyset$ and $\emptyset \vdash \overline{\mu\alpha.T_1} \overset{c}{\wedge} \overline{\mu\alpha.T_2} \rightarrow \overline{\mu\alpha.T_2'} \triangleright \emptyset$ then $\mu\alpha.T_1' \lesseqgtr \mu\alpha.T_1$ and $\mu\alpha.T_2' \lesseqgtr \mu\alpha.T_2$ and finally conclude that $T_1'[^\bullet/_\alpha] \leq T_2'[^\bullet/_\alpha]$. In few words we exploit the intersection algorithm introduced in Chapter 3 to obtain an expanded representation of $\mu\alpha.T_1$ and $\mu\alpha.T_2$ since if $\mu\alpha.T_1 \leq \mu\alpha.T_2$ it holds that $\mu\alpha.T_1 \wedge \mu\alpha.T_2 \lesseqgtr \mu\alpha.T_1$ and $\mu\alpha.T_1 \vee \mu\alpha.T_2 \lesseqgtr \mu\alpha.T_2$. For example $\emptyset \vdash \mu\alpha.!(int).\alpha \overset{c}{\wedge} \mu\alpha.!(int).!(int).\alpha \rightarrow \mu\alpha.!(int).!(int).\alpha \triangleright \emptyset$ and $\emptyset \vdash \overline{\mu\alpha.!(int).!(int).\alpha} \overset{c}{\wedge} \overline{\mu\alpha.!(int).\alpha} \rightarrow \overline{\mu\alpha.!(int).!(int).\alpha} \triangleright \emptyset$ and $!(int).!(int).\bullet \lesseqgtr !(int).!(int).\bullet$.

In order to do so, we first define a predicate $sameplace(\alpha, T_1, T_2)$ which holds if each occurrence of α, free in both T_1 and T_2, appears at the same height in the syntax tree of both T_1 and T_2 and type prefixes in T_1 and T_2 are compatible in the sense of subtyping relation. For instance both $sameplace(\alpha, !(\tilde{S}).\oplus\{l : \alpha, l_1 : \texttt{end}\}, !(\tilde{S}).\oplus\{l : \alpha\})$ and $sameplace(\alpha, ?(\tilde{S}).\&\{l : \alpha\}, ?(\tilde{S}).\&\{l : \alpha, l_1 : \alpha\})$ hold and neither $sameplace(\alpha, !(\tilde{S}).\oplus\{l : \alpha, l_1 : \texttt{end}\}, !(\tilde{S}).\oplus\{l : \alpha, l_1 : \alpha\})$ nor $sameplace(\alpha, \mu\alpha_1.\oplus\{l : \alpha_1, l_1 : \alpha\}, \oplus\{l_1 : \alpha\})$ hold. In particular the last predicate does not hold since the two types have different syntactic structure and α appears at different height. Next we must prove that $sameplace$ is preserved by unfolding and then we prove by induction that for every T_3 and T_4 s.t. $\Sigma \vdash T_1 \overset{c}{\wedge} T_2 \rightarrow T_3 \triangleright \Sigma$ and $\overline{\Sigma} \vdash \overline{T_1} \overset{c}{\wedge} \overline{T_2} \rightarrow \overline{T_4} \triangleright \overline{\Sigma}$ then $sameplace(\alpha, T_3, T_4)$ holds for every $\alpha \in \mathsf{fv}(T_3)$ and $\mathsf{fv}(T_3) = \mathsf{fv}(T_4)$. Finally one can prove that $R = \{(T_1[^\bullet/_\alpha], T_2[^\bullet/_\alpha]) | \forall V_1, V_2\ ,\ T_1[^{\mu\alpha.V_1}/_\alpha] \leq T_2[^{\mu\alpha.V_2}/_\alpha]$ and $sameplace(\alpha, T_1, T_2)\}$ is a type simulation relation. ☐

An equivalent of Lemma 4.21 also holds for the meet relation.

Lemma 4.23. *If* $T_1[^\bullet/_\alpha] \overset{c}{\wedge} T_2[^\bullet/_\alpha] = T_3[^\bullet/_\alpha]$ *then* $\mu\alpha.T_1 \overset{c}{\wedge} \mu\alpha.T_2 = \mu\alpha.T_3$

Proof. The proof proceeds in a similar manner to what we have done in Lemma 4.21, we prove that $R = \{(T_1[^{\mu\alpha_1.V_1}/_{\alpha_1}], T_2[^{\mu\alpha_2.V_2}/_{\alpha_2}], T_3[^{\mu\alpha_2.V_3}/_{\alpha_3}]) | T_1[^\bullet/_{\alpha_1}] \overset{c}{\wedge} T_2[^\bullet/_{\alpha_2}] = T_3[^\bullet/_{\alpha_3}]$ and $V_1[^\bullet/_{\alpha_1}] \overset{c}{\wedge} V_2[^\bullet/_{\alpha_2}] = V_3[^\bullet/_{\alpha_3}]\}$ is a meet relation. ☐

Following is an *important* lemma used in the proof of the Lemma 4.26, but it turns out that this lemma can be used in general to obtain a solution

to cyclic constraint of the form $\alpha \leq T$ with α free variable in T (but not free communicated variable) solved by $\alpha \leq \mu\alpha.T$ e.g. if $\alpha \leq !(int).\alpha$ then as expected $\alpha \leq \mu\alpha.!(int).\alpha$.

Lemma 4.24. *If $T \leq V[^T/_\alpha]$ and $\alpha \notin \mathsf{fcv}(V[^T/_\alpha])$ then $T \leq \mu\alpha.V$*

Proof. We prove that $R = \{(T_1, T_2[^{\mu\alpha.V}/_\alpha]) | T_1 \leq T_2[^T/_\alpha]$ and $T \leq V[^T/_\alpha]\}$ is a type simulation relation since $(T, V[^{\mu\alpha.V}/_\alpha]) \in R$. □

The next lemma says, if only sequences of outputs are considered (like the type relative to the parent session), then $\leq$ always implies $\lesseqgtr$.

Lemma 4.25. *Let U a type generated from the following syntax:*
$U ::= \mathtt{end} \mid \bullet \mid !(\tilde{S}).U \mid \mu\alpha.U \mid \alpha$ then we have $U \leq U'$ iff $U \lesseqgtr U'$.

Proof. The only if part of the proof derives directly by the definition of $\lesseqgtr$. For the if part we prove that $R = \{(U_1, U_2) | U_2 \leq U_1\}$ is a type simulation relation. If $\mathsf{unfold}(U_2) = !(\tilde{S}).U_2'$ then by definition of $\leq$ we have $\mathsf{unfold}(U_1) = !(\tilde{S}).U_1'$ and $U_2' \leq U_1'$ which concludes. The other cases for $\mathsf{unfold}(U_1) = \bullet$ and $\mathsf{unfold}(U_2) = \mathtt{end}$ follows directly. □

Now that we have finished with properties concerning only types we prove two additional properties that relate $\bullet$ also with processes for both $\vdash$ and $\vdash_{\mathsf{sd}}$. We start proving a property that show how types change if $\bullet$ is replaced by a recursion in $\vdash$. In the statement of the lemma we disallow a recursion under a nested session since Theorem 4.30 proves the correctness from a well-typed process in $\vdash_{\mathsf{sd}}$ to $\vdash$ then each P automatic fulfill this hypothesis.

Lemma 4.26. *If $\Gamma; \emptyset \vdash P[^\bullet/_X] : T'; U'; \emptyset$ and X does not appear under service invocations or service definitions or sessions then $\Gamma; \emptyset \vdash \mathtt{rec}\ X.P : T; U; \emptyset$ where if $T' = \bullet$ then $T = \mathtt{end}$ or if $T' = T''[^\bullet/_\alpha]$ then $T = \mu\alpha.T''$ or $T' = T$ otherwise, and if $U' = \bullet$ then $U = \mathtt{end}$ or if $U' = U[^\bullet/_\alpha]$ then $U = \mu\alpha.U''$ or $U' = U$ otherwise.*

Proof. The proof is by induction on the typing of $\Gamma; \emptyset \vdash P[^\bullet/_X] : T; U; \emptyset$ with case analysis on the last applied rule. Without loss of generality we assume $\bullet$ does not appear in the syntax tree of P. The base cases are when $\dot{P} = X$ and $\dot{P} = \mathbf{0}$. If $X = \dot{X}$ we have $\dfrac{(\text{TBULLET})}{\Gamma; \emptyset \vdash \bullet : \bullet; \bullet; \emptyset}$ then $\dfrac{\Gamma; X : \mathtt{end}; \mathtt{end} \vdash X : \mathtt{end}; \mathtt{end}; \emptyset}{\Gamma; \emptyset \vdash \mathtt{rec}\ X.X : \mathtt{end}; \mathtt{end}; \emptyset}\ (\text{TREC})$. The case where $\dot{P} = X \neq \dot{X}$ holds trivially since the premise of typability of P does not hold and in the case where

$\dot{P} = \mathbf{0}$ then $\dfrac{(\text{TNIL})}{\Gamma; \emptyset \vdash \mathbf{0} : \texttt{end}; \texttt{end}; \emptyset}$ and $\dfrac{\Gamma; X : \texttt{end}; \texttt{end} \vdash \mathbf{0} : \texttt{end}; \texttt{end}; \emptyset}{\Gamma; \emptyset \vdash \texttt{rec}\ X.\mathbf{0} : \texttt{end}; \texttt{end}; \emptyset}$ (TREC) which concludes. (Notice how we applied the consistency condition to type recursion.)

In the inductive cases, depending on the last applied rule:

- rule (TIN) with $\dot{P} = (\tilde{x}).P$ consequently the inductive hypothesis holds for $\Gamma, \tilde{x} : \tilde{S}; \emptyset \vdash P[^{\bullet}/_X] : T; U; \emptyset$ for some $\tilde{S}$. The U-component of the type is unchanged then the result for this part follows directly by induction. Regarding the T-component we have various cases, if $T = T'[^{\bullet}/_{\alpha}]$ then $\dot{T}' = ?(\tilde{S}).T'[^{\bullet}/_{\alpha}]$, for some $\tilde{S}$, and by induction $\texttt{rec}\ X.P$ is typed with $\mu\alpha.T'$ and the result follows since $\texttt{rec}\ X.(\tilde{x}).P$ is typed with $\mu\alpha.?(\tilde{S}).T'$ (applying rule (TIN)). Otherwise if $T = \bullet$ then $\dot{T}' = ?(\tilde{S}).\bullet$ and by induction $\texttt{rec}\ X.P$ has type $\texttt{end}$. The result follows since $\texttt{rec}\ X.(\tilde{x}).P$ has type, $\mu\alpha.?(\tilde{S}).\alpha$. In the last case $\texttt{rec}\ X.P$ has type $T'' = T$, the type of $P[^{\bullet}/_X]$, and then $\texttt{rec}\ X.(\tilde{x}).P$ has type $?(\tilde{S}).T'' = \dot{T}'$ for some $\tilde{S}$. In particular the step from $\mu\alpha.T'$, the type of $\texttt{rec}\ X.P$, to $\dot{T} = \mu\alpha.?(\tilde{S}).T'$ is not straightforward. We have $\dfrac{\Gamma, \tilde{x} : \tilde{S}; X : V; U \vdash P : T'[^{V}/_{\alpha}]; U; \emptyset}{\Gamma; X : V; U \vdash (\tilde{x}).P : ?(\tilde{S}).T'[^{V}/_{\alpha}]; U; \emptyset}$ (TIN) for some unknown V then in order to apply (TREC) we must have $V = ?(\tilde{S}).T'[^{V}/_{\alpha}]$ and by Lemma 4.24, $V \leqslant \mu\alpha.?(\tilde{S}).T'$ and we conclude by rule (TWEAK).

- rules (TOUT), (TRET), (TCHOICE) and (TBRANCH) are similar to the previous case but (TBRANCH) requires an inner induction on the number of labels.

- rule (TDEF) with $\dot{P} = v.P$ and $\Gamma; X : T; U \vdash P : T; U; \emptyset$. By hypothesis $X \notin \mathsf{fpv}(P)$ then $P[^{\bullet}/_X] = P$. Applying the Strengthening Lemma we have $\Gamma; \emptyset \vdash P[^{\bullet}/_X] : T; U; \emptyset$ which concludes by an application of the rule (TDEF) since $\dot{T}' = \dot{T}$ and $\dot{U}' = \dot{U}$.

- rules (TINV) and (TSES) similar to the previous case.

- rule (TWEAK) follows by induction and by the transitivity of $\leqslant$.

- rule (TPIPE) with $\dot{P} = P > \tilde{x} > Q$. The thesis follows by induction on Q if $\Gamma; \emptyset \vdash P\ :!(\tilde{S}); \texttt{end}; L_1$ for some $\tilde{S}$,L_1 otherwise we have $\Gamma; \emptyset \vdash \dot{P}[^{\bullet}/_X] : \texttt{end}; \texttt{end}; L$ for some L with $\dot{T}' = \dot{T}$ and $\dot{U}' = \dot{U}$.

- (TWEAK). We have $\dot{P} = P$ and $\Gamma; \emptyset \vdash P[{}^{\bullet}/_{X}] : T'; U'; \emptyset$ implies by induction $\Gamma; \emptyset \vdash \texttt{rec}\ X.P : T; U; \emptyset$. Let $T'^{\leq}$ s.t. $T'^{\leq} \leq T'$. If $T' = \bullet$ then $T'^{\leq} = \bullet$ and $T = \texttt{end}$ which concludes this case. If $T' = T''[{}^{\bullet}/_{\alpha}]$ for some T'' then $T'^{\leq} = T''^{\leq}[{}^{\bullet}/_{\alpha}]$ for some $T''^{\leq}$. By Lemma 4.21, $T''[{}^{\bullet}/_{\alpha}] \leq T''^{\leq}[{}^{\bullet}/_{\alpha}]$ implies $\mu\alpha.T'' \leq \mu\alpha.T''^{\leq}$ which concludes.

- for the remaining rules the thesis follows directly by induction.

□

We need the corresponding lemma for $\vdash_{\mathsf{sd}}$ in the opposite direction from processes with free variables to processes with a $\bullet$. This time we have a direct correspondence between process variables and type variables.

Lemma 4.27. *If* $\Gamma; \Theta, X : \alpha_X; \alpha_X \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ *then* $\Gamma, \Theta \vdash_{\mathsf{sd}} P[{}^{\bullet}/_{X}] : \mathcal{T}[{}^{\bullet}/_{\alpha_X}]; \mathcal{U}[{}^{\bullet}/_{\alpha_X}]; L$

Proof. The proof is by straightforward induction on the typing derivation of P with case analysis on the last applied rule. Without loss of generality we assume $\bullet$ does not appear in the syntax tree of P. As usual base cases are (TNILSD) and (TPVARSD). In the last case we have two sub-cases depending on whether $\dot{X} = X$ or not. In the inductive cases when the conclusion of a rule can produce $\texttt{end}$ as a type (e.g. (TPIPESD)) the thesis follows since $\texttt{end}[{}^{\bullet}/_{\alpha_X}] = \texttt{end}$ for any α_X. We report only cases when the last applied rules are (TRECSD) and (TPARLRSD). We have $\dot{P} = \texttt{rec}\ Y.P$ and $Y \neq \dot{X}$ by hypothesis. Let $X = \dot{X}$, we have by induction on P that $\Gamma; \Theta, X : \alpha_X; \alpha_X \vdash P : \mathcal{T}; \mathcal{U}; L$ implies $\Gamma, \Theta \vdash_{\mathsf{sd}} P[{}^{\bullet}/_{X}] : \mathcal{T}[{}^{\bullet}/_{\alpha_X}]; \mathcal{U}[{}^{\bullet}/_{\alpha_X}]; L$. We have three cases for $\mathcal{T}[{}^{\bullet}/_{\alpha_X}]$ and three for $\mathcal{U}[{}^{\bullet}/_{\alpha_X}]$ depending of the results of the function $\mu\alpha_Y$ used in the conclusion of rule (TREC). Let $T \in \mathcal{T} \cup \mathcal{U}$ if $\mu\alpha_Y(T) = \texttt{end}$ then $\mu\alpha_Y(T[{}^{\bullet}/_{\alpha_X}]) = \texttt{end}$ which concludes, if $\mu\alpha_Y(T) = T$ then $\mu\alpha_Y(T[{}^{\bullet}/_{\alpha_X}]) = T[{}^{\bullet}/_{\alpha_X}]$ which concludes and if $\mu\alpha_Y(T) = \mu\alpha_Y.T$ then $\mu\alpha_Y(T) = \mu\alpha_Y.T[{}^{\bullet}/_{\alpha_X}]$ since $\alpha_X \neq \alpha_Y$ by the fact that $X \neq Y$ which concludes. When the last applied rule is (TPARLRSD) we have $\Gamma; \Theta, X : \alpha_X; \alpha_X \vdash_{\mathsf{sd}} P_i : \mathcal{T}_i; \mathcal{U}_i; L_i$ for $i \in \{1, 2\}$ and $\Gamma; \Theta, X : \alpha_X; \alpha_X \vdash_{\mathsf{sd}} P_1 | P_2 : \mathcal{T}_1 \cup \mathcal{T}_2 \cup \{\texttt{end}\}; \mathcal{U}_1 \cup \mathcal{U}_1 \cup \{\texttt{end}\}; L_1 \cup L_2$. By induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i[{}^{\bullet}/_{X}] : \mathcal{T}_i[{}^{\bullet}/_{\alpha_X}]; \mathcal{U}_i[{}^{\bullet}/_{\alpha_X}]; L_i$ and we can conclude with $\Gamma; \Theta \vdash_{\mathsf{sd}} (P_1|P_2)[{}^{\bullet}/_{X}] : (\mathcal{T}_1 \cup \mathcal{T}_2 \cup \{\texttt{end}\})[{}^{\bullet}/_{\alpha_X}]; (\mathcal{U}_1 \cup \mathcal{U}_1 \cup \{\texttt{end}\})[{}^{\bullet}/_{\alpha_X}]; L_1 \cup L_2$. □

The next proposition relates the free variables of the set $\mathcal{T}$ w.r.t to the domain of Θ. This fact is important since $\leq$ is defined only for closed

types, then the lemma guarantees the subtyping is defined if $\Theta = \emptyset$, i.e. in rules (TDEFSD), (TINVSD) and (TSESSD).

Lemma 4.28. *If* $\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ *then* $\mathsf{fv}(\mathcal{T}) \subseteq \mathsf{dom}(\Theta)$.

Proof. By straightforward induction on the typing of P. □

The following is another simpler property of $\vdash_{\mathsf{sd}}$ which we are going to use to state its soundness.

Lemma 4.29. *If* $\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ *then* $|\mathcal{T}| \geq 1$ *and* $|\mathcal{U}| \geq 1$

Proof. The proof is by straightforward rule induction on the rules for $\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$. □

The soundness theorem proves that a judgment $\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ is correct if both there exists the greatest lower bound of $\mathcal{T}$ and there exists a type U s.t. $U \leq \mathcal{U}$ and $\mathcal{U} \leq U$ which we shall indicate with $U \lesseqgtr \mathcal{U}$. The greatest lower bound of $\mathcal{T}$ is indicated with $\bigwedge \mathcal{T}$ defined point-wise exploiting Lemma 4.29 and properties of the meet in Lemma 3.14.

The theorem is formulated for closed processes, i.e. we require Θ to be $\emptyset$. With this formulation however we cannot directly exploit the induction in the rule (TRECSD) since the rule adds new ssumptions in Θ. The solution is to replace the occurrence of a free process variable with a $\bullet$ and then to use the induction hypothesis on the resulting process. We can conclude thanks to the two previous lemmas that give a direct correspondence from the substituted process with the original one.

Theorem 4.30 (Soundness)**.** *Let* $\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$. *If* $T = \bigwedge \mathcal{T}$ *is defined and there exists* U *s.t.* $U \lesseqgtr \mathcal{U}$ *then* $\Gamma; \emptyset \vdash P : T; U; L$.

Proof. The proof is by induction on the derivation of $\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ with case analysis on the last applied rule. Without loss of generality we assume $\bullet$ does not appear as a subprocess of P and consequently in the syntax tree of any types in $\mathcal{T}$. The only base case, is when the rule (TNILSD) is applied then $\Gamma; \emptyset \vdash_{\mathsf{sd}} \mathbf{0} : \{\mathtt{end}\}; \{\mathtt{end}\}; \emptyset$ and $\Gamma; \emptyset \vdash_{\mathsf{sd}} \mathbf{0} : \{\mathtt{end}\}; \{\mathtt{end}\}; \emptyset$. In the inductive cases if the last applied rule is:

- (TRECSD) we have $\dot{P} = \mathtt{rec}\ X.P$. Given that $\Gamma; X : \alpha_X; \alpha_X \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; \emptyset$ and $\mathsf{fv}(\mathcal{T}) \subseteq \mathsf{fv}(\mathcal{U}) \subseteq \{\alpha_X\}$ by Lemma 4.28, and $\Gamma; \emptyset \vdash_{\mathsf{sd}} \mathtt{rec}\ X.P : \mu\alpha_X(\mathcal{T}); \mu\alpha_X(\mathcal{U}); \emptyset$ we must prove $\Gamma; \emptyset \vdash \mathtt{rec}\ X.P : T; U; \emptyset$ and $T = \bigwedge \mu\alpha_X(\mathcal{T})$ and $U \lesseqgtr \mathcal{U}$. We proceed as follows, since we cannot directly exploit the induction on P we

reason on $P[^{\bullet}/_X]$ instead. By Lemma 4.27 applied to P we have $\Gamma;\emptyset \vdash_{\mathsf{sd}} P[^{\bullet}/_X] : \mathcal{T}[^{\bullet}/_{\alpha_X}];\mathcal{U}[^{\bullet}/_{\alpha_X}];\emptyset$. Now the induction hypothesis gives $\Gamma;\emptyset \vdash P[^{\bullet}/_X] : T_1;U_1;\emptyset$ for some T_1 and U_1. We consider case analysis for types in $\mathcal{T}$, case analysis for $\mathcal{U}$ is similar and follows by Lemma 4.25. If $\bullet \in \mathcal{T}$ then it must be the case $\bigwedge \mathcal{T} = \bullet$ (since no other type is smaller than $\bullet$) and by induction $T_1 = \bullet$. The result follows since by Lemma 4.26, $\Gamma;\emptyset \vdash P[^{\bullet}/_X] : T_1;U_1;\emptyset$ implies $\Gamma;\emptyset \vdash \texttt{rec}\ X.P : \texttt{end};U_2;\emptyset$ for some U_2 and $\mu\alpha_X(\alpha_X) = \texttt{end}$. If $T'[^{\bullet}/_{\alpha_X}] \in \mathcal{T}$ then it must be by induction that $T_1 = \bigwedge(\mathcal{T}[^{\bullet}/_{\alpha}])$. By Lemma 4.26 we have $\Gamma;\emptyset \vdash P[^{\bullet}/_X] : T_1;U_1;\emptyset$ implies $\Gamma;\emptyset \vdash \texttt{rec}\ X.P : \mu\alpha.T_2;U_2;\emptyset$ for some U_2 where $T_2[^{\bullet}/_{\alpha}] = T_1$ and Lemma 4.23 implies $\mu\alpha.T_2 = \bigwedge(\mu\alpha.\mathcal{T})$ which allows to conclude. The last case when $T' \in \mathcal{T}$ and $T' \neq T''[^{\bullet}/_{\alpha}]$ for any T'' follows directly by induction, by Lemma 4.26 and by the fact that $\mu\alpha_X(T') = T'$.

- (TDefSD) we have $\dot{P} = a.P$ and $\Gamma;\emptyset \vdash_{\mathsf{sd}} P : \mathcal{T};\mathcal{U};L$ and $\Gamma \vdash a : [T]$ and $T \leq \mathcal{T}$ which is valid since by Lemma 4.28, $\mathsf{fv}(\mathcal{T}) = \emptyset$. By induction we know that $\Gamma;\emptyset \vdash P : T';U;L$ holds if $T' = \bigwedge \mathcal{T}$ and $U \lesseqgtr \mathcal{U}$. The result follows since $T \leq T'$ applying the rule (TWeak) first and rule (TDef) then.

- (TSesSD) similar to the previous case.

- (TInvSD) we have $\dot{P} = \overline{v}.P$ and $\Gamma;\emptyset \vdash_{\mathsf{sd}} P : \mathcal{T};\mathcal{U};L$ and $\Gamma \vdash v : [T]$ and $\overline{\mathcal{T}} \leq T$ which is valid since by Lemma 4.28, $\mathsf{fv}(\overline{\mathcal{T}}) = \emptyset$. By induction we know that $\Gamma;\emptyset \vdash P : T';U;L$ holds if $T' = \bigwedge \mathcal{T}$ and $U \lesseqgtr \mathcal{U}$. The result follows since $\overline{T'} \leq T$ and $T' \leq \overline{T}$ and we conclude applying the rule (TWeak) first and rule (TInv) then.

- (TInSD) we have $\dot{P} = ?(\tilde{x}).P$ and $\Gamma;\emptyset \vdash_{\mathsf{sd}} P : \mathcal{T};\mathcal{U};L$ with $\Gamma \vdash \tilde{x} : \tilde{S}$. By induction we know that $\Gamma;\emptyset \vdash P : T;U;L$ if $T = \bigwedge \mathcal{T}$ and $U \lesseqgtr \mathcal{U}$. The fact that $?(\tilde{S}).\bigwedge \mathcal{T}$ implies $\bigwedge ?(\tilde{S}).\mathcal{T}$ follows by definition of meet relation.

- (TOutSD) similar to the previous case.

- (TRetSD) we have $\dot{P} = \texttt{return}\ \tilde{v}.P$ and $\Gamma;\emptyset \vdash_{\mathsf{sd}} P : \mathcal{T};\mathcal{U};L$ with $\Gamma \vdash \tilde{v} : \tilde{S}$. By induction we know that $\Gamma;\emptyset \vdash P : T;U;L$ if $T = \bigwedge \mathcal{T}$ and $U \lesseqgtr \mathcal{U}$. With T we are fine, regarding U we conclude with $?(\tilde{S}).U$ since $?(\tilde{S}).U \lesseqgtr ?(\tilde{S}).\mathcal{U}$.

- (TCHOICESD) we have $\dot{P} = \langle l \rangle.P$ and $\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$. By induction we know that $\Gamma; \emptyset \vdash P : T; U; L$ if $T = \bigwedge \mathcal{T}$ and $U \lesssim \mathcal{U}$. The result follows with $\Gamma; \emptyset \vdash \langle l \rangle.P : \oplus\{l : T\}; U; L$ since $\oplus\{l : \bigwedge \mathcal{T}\}$ implies $\bigwedge \oplus\{l : \mathcal{T}\}$ by definition of meet relation.

- (TBRANCHSD) we have $\dot{P} = \Sigma_{i=1}^{n}(l_i).P_i$ and $\Gamma; \emptyset \vdash_{\mathsf{sd}} P_i; \mathcal{T}_1; \mathcal{U}_i; L$ and by induction that $\Gamma; \emptyset \vdash P_i : T_i; U_i$ if $T_i = \bigwedge \mathcal{T}_i$ and $U_i \lesssim \mathcal{U}_i$. First of all (since $U \lesssim \mathcal{U}_1$ and $U \lesssim \mathcal{U}_2$ implies $U \lesssim \mathcal{U}_1 \cup \mathcal{U}_2$) we use rule (TWEAK) n times in order to have $\Gamma; \emptyset \vdash P_i : T_i; U; L$ where $U \lesssim U_i$ for all i. Regarding the T-component we notice that $\bigwedge(\mathcal{T}_1 \cup \mathcal{T}_2) = \bigwedge \mathcal{T}_1 \wedge \bigwedge \mathcal{T}_2$ by definition. Moreover, $\&\{l_1 : \bigwedge \mathcal{T}_1, \ldots, l_n : \bigwedge \mathcal{T}_n\} = \bigwedge(\&\{l_1 : \mathcal{T}_1, \ldots, l_n : \mathcal{T}_n\})$ by definition of meet relation. We conclude since letting $I = \{1, \ldots, n\}$, $\bigwedge \&\{\mathsf{same}(l_i : \mathcal{T}_i)_{i \in I}\} = \&\{\mathsf{same}(l_i : \bigwedge \mathcal{T}_i)_{i \in I}\}$ (this fact can be easily proved by induction exploiting the two properties mentioned above) and applying rule (TWEAK) as many times as necessary to type n processes having the same label with the greatest lower bound of the types returned by the inductive hypothesis relative to these processes, we can conclude with rule (TBRANCH) $\Gamma; \emptyset \vdash \dot{P} : \bigwedge \&\{\mathsf{same}(l_i : \mathcal{T}_i)_{i \in I}\}; U; L$.

- (TPARLRSD) we have $\dot{P} = P|Q$ and the inductive hypothesis holds for both $\Gamma; \emptyset \vdash P : T_1; U_1; L_1$ and $\Gamma; \emptyset \vdash P : T_2; U_2; L_2$ and T_1, U_1, T_2, U_2 satisfy the theorem hypothesis. However, the conclusion of rule (TPARLRSD) adds `end` in the two resulting types which means that T_1, U_1, T_2, U_2 must be equal to `end` because the only type less than `end` is `end` itself then we can conclude with either (TPARL) or (TPARR).

- (TPIPESD) we have $\dot{P} = P > \tilde{x} > Q$. If $\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \{!(\tilde{S})\}; \{\mathsf{end}\}; L_1$ by induction it must be $\Gamma; \emptyset \vdash P : !(\tilde{S}); \mathsf{end}; L_1$ and then the result follows directly by inductive hypothesis on the typing derivation of Q. If $\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \{\mathsf{end}\}; \{\mathsf{end}\}; L_1$ then $\Gamma; \emptyset \vdash P : \mathsf{end}; \mathsf{end}; L_1$ and the result follows for $\dot{T} = \mathsf{end}$ and $\dot{U} = \mathsf{end}$.

- remaining rules follow directly by induction.

□

We note that the side conditions $T = \bigwedge \mathcal{T}$ and $U \lesssim \mathcal{U}$, in the previous theorem, can be self-checked by the system itself creating two fresh services as follows: let a, b s.t. $\{a, b\} \cap \mathsf{fn}(P) = \emptyset$ if $\Gamma; \emptyset \vdash P : \mathcal{T}; \mathcal{U}; L$ then $\Gamma, a : U, b : \bigwedge \mathcal{T}; \emptyset \vdash a.b.P : \mathsf{end}; \mathsf{end}; L$ and $\Gamma; \emptyset \vdash (\nu a)(\nu b)a.b.P : \mathcal{T}; \mathcal{U}; L$.

Starting from the syntax directed type system we define an algorithm to extract a set $\mathcal{C}$ of constraints. We use a type variable when a particular type is unknown. The algorithm `INF` in Figure 22 takes a process P without free process variables (i.e. Θ is empty) and with every bound and free names different (we write $\Gamma \cup \{m : S\}$ and $\Gamma \cup \{x : S\}$ to mean $\Gamma, m : S$ and $\Gamma, x : S$). Together with P the algorithm also expects a typing environment Γ containing type assumptions on free names (each assumption can be either a type variable or a type inserted for type checking purposes) and returns a set $\mathcal{C}$ of constraints over type variables, the set of types $\mathcal{T}$ of the current session, the type $\mathcal{U}$ of the parent session and the set L of opened session names. (In the algorithm $\{\mathcal{T} \leq T\}$ is a shorthand for $\{T' \leq T \mid T' \in \mathcal{T}\}$.) `VALUEINF` is the counterpart of the algorithm for values, it takes a value and an environment Γ and returns the type of the value.

In the following we use: $\sigma, \rho, \ldots$ for substitutions, the juxtaposition $\sigma'\sigma$ for substitution composition, σT for the result of the simultaneous substitution σ applied to the free variables of T and $\sigma\mathcal{T}$,$\sigma\mathcal{C}$ for the result of the simultaneous substitution applied to each element in the set. We write $\sigma \vDash \mathcal{C}$ if σ is a substitution s.t. the entire set of constraints $\sigma\mathcal{C}$ holds. The `INF` algorithm is sound and complete w.r.t. the typing rules in the following sense:

Proposition 4.31 (Soundness and Completeness of `INF`)**.** *Let* $\texttt{INF}(P, \Gamma, \Theta) = (\mathcal{C}, \mathcal{T}, \mathcal{U}, L)$ *and* σ *a substitution s.t.* $(\mathsf{fv}(\mathcal{T}) \cup \mathsf{fv}(\mathcal{U})) \cap \mathsf{dom}(\sigma) = \emptyset$. *Then* $\sigma \vDash \mathcal{C}$ *iff* $\sigma\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$.

Proof. $\Rightarrow$) The proof is by induction on the recursive structure of a run of $\texttt{INF}(P, \Gamma, \Theta)$ with case analysis on the last applied rule. We sketch some inductive cases. When the last applied rule is:

- relative to service definition we have $\dot{P} = a.P$. By induction $\sigma \vDash \mathcal{C}$ (the set of constraints generated by `INF` for P, Γ, Θ) implies $\sigma\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$. By hypothesis also holds that $\sigma \vDash \mathcal{C} \cup \{\alpha \leq \mathcal{T}\} \cup \{\Gamma(a) = [\alpha]\}$ then we have $\dfrac{\sigma\Gamma; \emptyset \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad \sigma\Gamma \vdash a : [\sigma\alpha] \quad \sigma\alpha \leq \mathcal{T}}{\sigma\Gamma; \emptyset \vdash_{\mathsf{sd}} a.P : \mathcal{U}; \{\texttt{end}\}; \texttt{L}}\ \text{(TDEFSD)}$ which concludes.

- relative to the return construct, (TRETSD) we have $\dot{P} = \texttt{return}\ \tilde{v}.P$. By induction $\sigma \vDash \mathcal{C}$ (the set of constraints generated by `INF` for P, Γ, Θ) implies $\sigma\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ and then $\dfrac{\sigma\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L \quad \sigma\Gamma \vdash \tilde{v} : \sigma\tilde{S}}{\sigma\Gamma; \Theta \vdash_{\mathsf{sd}} \texttt{return}\ \tilde{v}.P : \mathcal{T}; !(\tilde{S}).\mathcal{U}; L}\ \text{(TRETSD)}$ which concludes.

```
VALUEINF(x,Γ)= Γ(x)    VALUEINF(s,Γ)= Γ(s)    VALUEINF(n,Γ)= (int)
INF(0,Γ,Θ)= (∅,{end},{end},∅)
INF(a.P,Γ,Θ)= let (C,T,U,L)=INF(P,Γ,∅)
              in (C ∪ {α ≤ T} ∪ {Γ(a) = [α]},U,{end},L)     (where α fresh)
INF(ā.P,Γ,Θ)= let (C,T,U,L)=INF(P,Γ,∅)
              in (C ∪ {T̄ ≤ α} ∪ {Γ(v) = [α]},U,{end},L)     (where α fresh)
INF((x1,...,xn).P,Γ,Θ)=
              let (C,T,U,L)=INF(P,Γ ∪ {x1 : β1,...,xn : βn},Θ)   β1,.,βn fresh
              in (C,?(β1,...,βn).T,U,L)
INF(⟨v1,...,vn⟩.P,Γ,Θ)= let (C,T,U,L)=INF(P,Γ,Θ) in
              let S1=VALUEINF(v1,Γ)......Sn=VALUEINF(vn,Γ)
              in (C,!(S1,...,Sn).T,U,L)
INF(return v1,...,vn.P,Γ,Θ)= let (C,T,U,L)=INF(P,Γ,Θ) in
              let S1=VALUEINF(v1,Γ)......Sn=VALUEINF(vn,Γ)
              in (C,T,!(S1,...,Sn).U,L)
INF(if v1 = v2 then P else Q,Γ,Θ)= let (C,T,U,L)=INF(P,Γ,Θ) in
              let (C1,T1,U1,L)=INF(Q,Γ,Θ) in
              let S1=VALUEINF(v1,Γ) S2=VALUEINF(v2,Γ)
              in (C ∪ C1 ∪ {S1 = S2},T ∪ T1,U ∪ U1,L)
INF((νa)P,Γ,Θ)=INF(P,Γ ∪ {a : β},Θ) (where β fresh)
INF((νr)P,Γ,Θ)=
        let (C,T,U,L)=INF(P,Γ ∪ {r+ : [α1],r− : [α2]},Θ) (where α1,α2 fresh)
              in (C ∪ {α1 = ᾱ2},T,U,L \ {r+,r−})
               (where L ∩ {r+,r−} = {r+,r−} or ∅)
INF(P|Q,Γ,Θ)=   let (C,T,U,L1)= INF(P,Γ,Θ) in
                let (C1,T1,U1,L2)=INF(Q,Γ,Θ) in
                if T=={end} and U=={end} then (T2,U2)=(T1,U1)
                else if T1=={end} and U1=={end} then (T2,U2)=(T,U)
                else (T2,U2)=(T ∪ T1 ∪ {end},U ∪ U1 ∪ {end})
              in (C ∪ C1,T2,U2,L1 ⊎ L2)
INF(Σ^n_{i=1}(li).Pi,Γ,Θ)=
              let (C1,T1,U1,L)=INF(P1,Γ,Θ)......(Cn,Tn,Un,L)=INF(Pn,Γ,Θ)
              U=⋃_i Ui    i ∈1...n
              in  (C1 ∪ ... ∪ Cn,&{same(l1 : T1,...,ln : Tn)},U,L)
INF(⟨l⟩.P,Γ,Θ)=let (C,T,U,L)= INF(P,Γ,Θ) in (C,⊕{l : T},U,L)
INF(P > x1,...,xn > Q,Γ,Θ)= let (C,T,U,L)=INF(P,Γ,∅)
              in let(C1,T1,U1,∅)=INF(Q,Γ ∪ {x1 : S1,...,xn : Sn},Θ)
               in if T=={end} and U=={end} then
                 (C ∪ C1,{end},{end},L)
              else if T=={!(S̃)} and U=={end} then
                 (C ∪ C1,T1,U1,L)
              else error!!!
INF(r^p ▷ P,Γ,Θ)= let (C,T,U,L)=INF(P,Γ,∅)
           in (C ∪ {α ≤ T} ∪ {Γ(r^p) = [α]},U,{end},(L ⊎ {r^p}))
           (where α fresh)
INF(rec X.P,Γ,Θ)= let (C,T,U,∅)=INF(P,Γ,Θ ∪ {X : αX;αX})
                in (C,μα(T),μα(U),∅)
INF(X,Γ,Θ)= (∅,{(fst Θ(X))},{(snd Θ(X))},∅)
```

Figure 22: The algorithm to extract constraints in Ocaml-like syntax

$\Leftarrow$) The proof is by induction on the recursive structure of a run of $\mathtt{INF}(P, \Gamma, \Theta)$ with case analysis on the last applied rule. The proof is almost similar to the if part with inverse implications

- service restriction; We have $\dot{P} = (\nu a)P$ and $\sigma\Gamma; \Theta \vdash (\nu a)P : \mathcal{T}; \mathcal{U}; L$ and by induction $\sigma\Gamma, a : \sigma\beta; \Theta \vdash_{\mathsf{sd}} P : \mathcal{T}; \mathcal{U}; L$ implies $\sigma \vDash \mathcal{C}$ (the set of constraints generated by INF for P, $\Gamma, a : \beta$ and Θ) which concludes.

- parallel composition: We have $\dot{P} = P_1 | P_2$ and by induction $\sigma\Gamma; \Theta \vdash_{\mathsf{sd}} P_i : \mathcal{T}_i; \mathcal{U}_i; L$ $\sigma \vDash \mathcal{C}_i$ and $\mathtt{INF}(P_i, \Gamma, \Theta) = (\mathcal{C}_i, \mathcal{T}_i, \mathcal{U}_i)$ where $i \in \{1, 2\}$. By hypothesis $(\mathsf{fv}(\mathcal{T}_i) \cup \mathsf{fv}(\mathcal{U}_i)) \cap \mathsf{dom}(\sigma) = \emptyset$ then it must be the case that either $\mathcal{T}_1 = \mathtt{end}$ and $\mathcal{U}_1 = \mathtt{end}$ or $\mathcal{T}_2 = \mathtt{end}$ and $\mathcal{U}_2 = \mathtt{end}$ or neither. INF mimes all the rules for parallel composition and the result follows since $\sigma \vDash \mathcal{C}_1 \cup \mathcal{C}_2$.

□

Some observations about the previous theorem follow. The first interesting fact is that, regarding the process environment Θ, we do not need any solving substitution σ since all the work is done by the syntax directed type system (and the relative proofs). The restriction of the domain of the solving substitution is also due to the fact that such substitution should not interfere with the work done by the type system.

We now, give the solve algorithm which takes a set of constraints $\mathcal{C}$ and returns *success* if there exists a substitution σ that solves the set of constraints, i.e. $\sigma \vDash \mathcal{C}$, or it fails otherwise. Let $(\mathcal{C}, \mathcal{T}, \mathcal{U}, _) = \mathtt{INF}(P, \Gamma, \emptyset)$, we apply the solve algorithm defined in Figure 23 on $\mathcal{C} \cup \{(\alpha_t \leq T) | T \in \mathcal{T}\} \cup \{(\alpha_u \leq U) | U \in \mathcal{U}\}$ where both α_t and α_u are fresh. The semantic of the match is Ocaml-like but with a slight abuse of notation we assume pattern-matching over sets is possible (while we had to implement it in the running version of the algorithm).

It easy to see that each step preserves the solvability of the original constraints. Some comments about the algorithm follow. In line 1 we solve the unification constraints and in a similar manner in line 2 we solve inequality using the most general syntactic unifier $\sigma_{T_1 \leq T_2}$ (notice the precondition of the rule). In lines 3 and 4 we compute the meet and the join resulting after the computation of the most general syntactic unifier. In few words we first compute a substitution s.t. $T_1 \wedge T_2$ or $T_1 \vee T_2$ is defined using the syntactic unifier relative to the meet exists $\sigma_{T_1 \wedge T_2}$ then we compute a representant of the intersection returned by the algorithm

```
let rec solve 𝒞 = match 𝒞 with
1   |S₁ = S₂, 𝒞' -> solve σ_{S₁=S₂} 𝒞'
2   |T₁ ≤ T₂, 𝒞' when fv(T₁) ∪ fv(T₂) = fcv(T₁) ∪ fcv(T₂) -> solve σ_{T₁≤T₂} 𝒞'
3   |α ≤ T₁, α ≤ T₂, 𝒞' -> solve σ_{T₁∧T₂}(𝒞', α ≤ T₁ ∧ T₂)
4   |T₁ ≤ α, T₂ ≤ α, 𝒞' -> solve σ_{¬T₁∧¬T₂}(𝒞', ¬(¬T₁ ∧ ¬T₂) ≤ α)
5   |α ≤ T₁, T₂ ≤ α, 𝒞' -> solve 𝒞', T₂ ≤ T₁, α ≤ T₁
6   |α₁ = ¬α₂, α₁ ≤ T₁, α₂ ≤ T₂, 𝒞' ->  solve 𝒞', ¬T₁ ≤ T₂
7   |T ≤ T₁, 𝒞'-> solve 𝒞'
8   |_ -> success
```

Figure 23: The solving algorithm

in Figure 6 with the substituted types as input. Line 5 is relative to the invocation by a client to a service: we use the transitivity to merge the two definitions. We also remember the equation relative to the service since yet another client can be later discovered by some substitution due to the service name extrusion. Line 6 joins the two sides of a session. During the running of `INF` we used the equation $\alpha_1 = \overline{\alpha_2}$ as a bookmark to remember that both α_1 and α_2 are specifications relative to the same session with dual polarity. Here we join the two equations, notice also that there does not exist in $\mathcal{C}'$ any other inequality either of the form $\alpha_1 \leq U_1$ or of the form $\alpha_2 \leq U_2$ otherwise line 4 would be applied. We can also remove the bookmark equation since session delegation is not allowed in CST then the relative equations are not necessary anymore.

It is not directly reported here but of course the algorithm fails if one of the unifier does not exist. Finally notice, the low complexity of `solve`, i.e. $|\mathcal{C}|^2$ is a good complexity lover bound since the worst case is when we should apply $|\mathcal{C}|/2$ times line 5. Further the size of $|\mathcal{C}|$, apart for the unification constraints, depends only on the number of service invocations, service declarations and sessions in the input process.

Lemma 4.32. `solve` *terminates.*

Proof. Observe that, at each iteration, the algorithm decrements either the overall number of constraints or the number of constraints of the form $\alpha \leq T_1$ and $T_2 \leq \alpha$. $\square$

As already discussed the most general syntactic unifiers, $\sigma_{T_1 \leq T_1}$ and $\sigma_{T_1 \wedge T_1}$, are the most general unifier if the set of free variables and the set of free communicated variables coincide. The next lemma proves that

for set of constraints generated by `INF` we are guaranteed that the most general syntactic unifier $\sigma_{T_1 \wedge T_2}$ is in fact the most general unifier when Θ is empty. The notion of free variables is extended point-wise to Θ using $\mathsf{fv}(\Theta)$.

Lemma 4.33. *Let $\mathcal{C}$ s.t. $(\mathcal{C}, _, _, _) = \mathtt{INF}(P, \Gamma, \Theta)$. If $\alpha \leq T \in \mathcal{C}$ then $(\mathsf{fv}(T) \setminus \mathsf{fcv}(T)) \subseteq \mathsf{fv}(\Theta)$ and if $T \leq \alpha \in \mathcal{C}$ then $(\mathsf{fv}(T) \setminus \mathsf{fcv}(T)) \subseteq \mathsf{fv}(\Theta)$.*

Proof. The proof is by straightforward induction on the recursive structure of a run of $\mathtt{INF}(P, \Gamma, \Theta)$ with case analysis on the last applied rule. □

Finally the main theorem which asserts the validity of our `solve` algorithm for closed processes without free names. The condition on the domain of σ says that we want a minimal substitution w.r.t. to the free variables in the set of constraints, moreover this allows the application of Lemma 3.33.

Theorem 4.34. *Let $\mathcal{C}$ s.t. $(\mathcal{C}, _, _, _) = \mathtt{INF}(P, \emptyset, \emptyset)$ then $\mathtt{solve}\ \mathcal{C} = \mathit{success}$ if and only there exists σ such that $\sigma \vDash \mathcal{C}$ and $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(\mathcal{C})$.*

Proof. $\Rightarrow$) The proof is by induction on the recursive structure of a run of `solve` $\mathcal{C}$ with case analysis on the last applied rule. However the induction does not work directly since the number of elements in the constraint set do not decrease at each interaction. The well founded order we are going to define is: $\mathcal{C} < \mathcal{C}'$ if $|\mathcal{C}| < |\mathcal{C}'|$ or if $|\mathcal{C}| = |\mathcal{C}'|$ and $\mathcal{C}$ contains less equations of the form $\alpha \leq T_1$ and $T_2 \leq \alpha$ than $\mathcal{C}'$. In the base case, when the line 8 of `solve` is applied then `solve` is invoked with the empty set of constraints and $\emptyset \vDash \emptyset$ that is an empty set of constraints is solved by the empty substitution. In the inductive case when the last line is:

- the line 1 of `solve`. We have by induction that if `solve` $\sigma_{S_1=S_2}\mathcal{C}' = \mathit{success}$ there exists σ s.t. $\sigma \vDash \sigma_{S_1=S_2}\mathcal{C}'$. The result follows since by Lemma 3.33, $\sigma_{S_1=S_2}\sigma \vDash \mathcal{C}'$ and $\sigma_{S_1=S_2}\sigma \vDash \mathcal{C}', S_1 = S_2$. The composition of $\sigma_{S_1=S_2}\sigma$ is defined since by induction $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(\sigma_{S_1=S_2}\mathcal{C}')$ which implies $\mathsf{dom}(\sigma_{S_1=S_2}) \cap \mathsf{dom}(\sigma) = \emptyset$. We implicitly assume this step in the next cases.

- the line 2 of `solve` is similar to the previous case but we use Proposition 3.30.

- the line 3 of `solve`. We have by induction that if `solve` $\sigma_{T_1 \wedge T_2}(\mathcal{C}', \alpha \leq T_1 \wedge T_2) = \mathit{success}$ there exists σ s.t.

$\sigma \vDash \sigma_{T_1 \wedge T_2}(\mathcal{C}', \alpha \leq T_1 \wedge T_2)$. Then $\sigma_{T_1 \wedge T_2}\sigma \vDash \mathcal{C}', \alpha \leq T_1 \wedge T_2$ and by definition of greatest lower bound $\sigma_{T_1 \wedge T_2}\sigma \vDash \mathcal{C}', \alpha \leq T_1, \alpha \leq T_2$ which concludes.

- the line 4 of `solve`. We have by induction that if `solve` $\sigma_{\overline{T_1} \wedge \overline{T_2}}(\mathcal{C}', \overline{\overline{T_1} \wedge \overline{T_2}} \leq \alpha) = \textit{success}$ there exists σ s.t. $\sigma \vDash \sigma_{\overline{T_1} \wedge \overline{T_2}}(\mathcal{C}', \overline{\overline{T_1} \wedge \overline{T_2}} \leq \alpha)$. Then $\sigma_{\overline{T_1} \wedge \overline{T_2}}\sigma \vDash \mathcal{C}', \overline{\overline{T_1} \wedge \overline{T_2}} \leq \alpha$ and $\sigma_{\overline{T_1} \wedge \overline{T_2}}\sigma \vDash \mathcal{C}', T_1 \vee T_2 \leq \alpha$ by Lemma 3.13 and by definition of least upper bound $\sigma_{\overline{T_1} \wedge \overline{T_2}}\sigma \vDash \mathcal{C}', T_1 \leq \alpha, T_2 \leq \alpha$ which concludes.

- the line 5 of `solve`. We have by induction that if `solve` $(\mathcal{C}', T_2 \leq T_1, \alpha \leq T_1) = \textit{success}$ then there exists σ s.t. $\sigma \vDash \mathcal{C}', T_2 \leq T_1, \alpha \leq T_1$ and then $[{}^{T_1}/_{\alpha}]\sigma' \vDash \mathcal{C}', \alpha \leq T_1, T_2 \leq \alpha$ where σ' is obtained from σ removing the substitution for α (remember that another constraint of the form $\alpha \leq T_3$ cannot appear in $\mathcal{C}'$ since service definitions cannot be variables, that is α cannot be less than T_1).

- line 6 of `solve`. We have by induction that if `solve` $(\mathcal{C}', \overline{T_1} \leq T_2)$ then there exists σ s.t. $\sigma \vDash \mathcal{C}', \overline{T_1} \leq T_2$ and we must prove $\sigma' \vDash \mathcal{C}', \alpha_1 = \overline{\alpha_2}, \alpha_1 \leq T_1, \alpha_2 \leq T_2$ for some σ'. Since neither $\overline{\alpha_2}$ nor α_2 nor α_1 appear elsewhere in $\mathcal{C}$ by session linearity, we have $[{}^{\overline{T_1}}/_{\alpha_1}][{}^{T_1}/_{\alpha_2}]\sigma \vDash \mathcal{C}', \alpha_1 = \overline{\alpha_2}, \overline{T_1} \leq \alpha_1, \alpha_2 \leq T_2$ which concludes this case.

- the line 7 of `solve`. We have $\dot{\mathcal{C}} = \mathcal{C}', T \leq T_1$ and by induction if `solve` $\mathcal{C}' = \textit{success}$ there exists σ s.t. $\sigma \vDash \mathcal{C}'$. Moreover one can easily prove (exploiting the hypothesis that the set of constraints is returned by $\texttt{INF}(P, \emptyset, \emptyset)$) that if line 6 is applied either $T = \alpha$ or $T_1 = \alpha_1$ for some α, α_1 which means that whatever substitution which solves either $\alpha \leq T_1$ or $T \leq \alpha$ holds (otherwise another line would be applied), let us take $[{}^{T_1}/_{\alpha}]\sigma$ or $[{}^{T}/_{\alpha_1}]\sigma$ respectively.

$\Leftarrow$) In the only if direction the proof shows that, each time, we always choose the most general way to proceed without compromising the result. The easiest way to prove the proposition is to do it by induction on the resolution steps of `solve` $\mathcal{C}$ and the result trivially holds by induction. But we must pay attention and also prove that the substitution used in the inductive hypothesis is implied by the premise, for example in line 1 we know that $\sigma \vDash \sigma_{S_1 = S_2}\mathcal{C}'$ and we must show that $\sigma' \vDash \mathcal{C}', S_1 = S_2$ is implied for some σ'. Moreover, the reasoning is sound since `solve` is defined for every constraints returned by `INF`.

Likewise, when line 8 is applied, the empty set of constraints is solved by the empty substitution. In the inductive case when

- line 1 is applied we must prove that $\sigma \vDash \sigma_{S_1=S_2}\mathcal{C}$ is implied by the premise. If $\rho \vDash \mathcal{C}, S_1 = S_2$ for some ρ then by definition of mgu it must be $\rho = \sigma_{S_1=S_2}\rho'$ and applying Lemma 3.33 we can conclude since $\rho' \vDash \sigma_{S_1=S_2}(\mathcal{C}, S_1 = S_2)$. Notice that a less general substitution than $\sigma_{S_1=S_2}$ would not allow us to conclude.

- lines 2 is applied, similar to the previous case using the fact that $\sigma_{T_1 \leq T_2}$ with the fact that (when the set of free variables and the set of free communicated variables coincide) is the most general substitution that satisfies $T_1 \leq T_2$ (Proposition 3.30).

- line 3 is applied we have $\rho \vDash \alpha \leq T_1, \alpha \leq T_2, \mathcal{C}$ then it must be the case that $\rho = \sigma_{T_1 \wedge T_2}\sigma'$ by Proposition 3.31 and Lemma 4.33 for some σ' which allows to conclude.

- line 4 is applied similar to the previous case.

- line 5 and line 6 and line 7. Regarding line 5 we must prove that $\sigma \vDash \mathcal{C}, \alpha \leq T_1, T_2 \leq \alpha$ is implied by $\sigma \vDash \mathcal{C}, \alpha \leq T_1, T_2 \leq T_1$ which follows directly by transitivity. About line 6 observe that $\sigma \vDash \alpha_1 = \overline{\alpha_2}, \alpha_1 \leq T_1, \alpha_2 \leq T_2$ implies $\sigma \vDash \overline{T_1} \leq \alpha_2, \alpha_2 \leq T_2$ which concludes by transitivity. Finally about line 7 observe that if $\sigma \vDash \mathcal{C}', \alpha_1 \leq T_1$ then $\sigma' \vDash \mathcal{C}'$ where σ' is obtained from σ removing the substitution relative to α_1. The case when $\sigma \vDash \mathcal{C}', T_1 \leq \alpha$ is similar and we are done since line 7 is applied only for these constraints as we have already discussed.

□

Solving the set of constraints generated by `INF` with an arbitrary Γ can be an issue depending of what kind of assumptions are allowed. If the user is allowed to insert only closed types as assumptions, then `solve` still works, otherwise one need a very different algorithm since for example assumptions inserted by the user can have mutual dependencies.

Example 4.35. The constraints generated by `INF` for the process in Example 4.1 are:

$$\alpha_a \leq \mu\alpha_{Y_2}.\&\{l : \oplus\{l_1 : \alpha_{Y_2}\}\} \quad \alpha_a \leq \mu\alpha_{Y_2}.\&\{l : \oplus\{l_2 : \alpha_{Y_2}\}\}$$

$$\alpha_a \leq \mu\alpha_{Y_1}.\&\{l : \oplus\{l_1 : \alpha_{Y_1}\}\} \quad \alpha_a \leq \mu\alpha_{Y_1}.\&\{l : \oplus\{l_3 : \alpha_{Y_1}\}\}$$

$$\overline{\mu\alpha_X. \oplus \{l : \&\{l_1 : \alpha_X\}\}} \leq \alpha_a \quad \overline{\mu\alpha_X. \oplus \{l : \&\{l_2 : \alpha_X\}\}} \leq \alpha_a$$
$$\overline{\mu\alpha_X. \oplus \{l : \&\{l_3 : \alpha_X\}\}} \leq \alpha_a \quad \overline{\mu\alpha_X. \oplus \{l : \&\{l_4 : \alpha_X\}\}} \leq \alpha_a$$

The first group of two constraints is relative to the first definition of service a, notice that the replication does not add any additional constraint. The second line is relative to the second definition of a and the third group of constraints is relative to service invocation. After computing all intersections and unions for each group we have:

$$\alpha_a \leq \&\{l : \oplus\{l_1 : \mu\alpha_{Y_2}.\&\{l : \oplus\{l_1 : \alpha_{Y_2}\}\}, l_2 : \mu\alpha_{Y_2}.\&\{l : \oplus\{l_2 : \alpha_{Y_2}\}\}\}\}$$

$$\alpha_a \leq \&\{l : \oplus\{l_1 : \mu\alpha_{Y_1}.\&\{l : \oplus\{l_1 : \alpha_{Y_1}\}\}, l_3 : \mu\alpha_{Y_1}.\&\{l : \oplus\{l_3 : \alpha_{Y_1}\}\}\}\}$$

$$\overline{\mu\alpha_X. \oplus \{l : \&\{l_1 : \alpha_X, l_2 : \alpha_X, l_3 : \alpha_X, l_4 : \alpha_X\}\}} \leq \alpha_a$$

in which remains an equation for each declaration/invocation of a and after intersecting the two equations relative to the two replies of a, remains to compute:

$$\overline{\mu\alpha_X. \oplus \{l : \&\{l_1 : \alpha_X, l_2 : \alpha_X, l_3 : \alpha_X, l_4 : \alpha_X\}\}} \leq \mu\alpha_Y.\&\{l : \oplus\{l_1 : \alpha_Y, l_2 : \alpha_Y, l_3 : \alpha_Y\}\}$$

which holds. For brevity we wrote $\mu\alpha_Y.\&\{l : \oplus\{l_1 : \alpha_Y, l_2 : \alpha_Y, l_3 : \alpha_Y\}\}$ which is a minorant of the real intersection.

Example 4.36. We report the four main steps of `solve` applied to the Example 4.2.

1	$\beta_a = [\alpha_a] \quad \beta_x = [\alpha_x] \quad \beta_b = [\alpha_b] \quad \alpha_a \leq ?(\beta_x) \quad \overline{!(\beta_a)} \leq \alpha_x \quad \overline{!(\beta_b)} \leq \alpha_a \quad \alpha_b \leq ?(\beta_{x_1})$
2	$\overline{!([\alpha_b])} \leq ?([\alpha_x]) \quad \overline{!([\alpha_a])} \leq \alpha_x \quad \alpha_b \leq ?(\beta_{x_1})$
3	$\overline{!([\alpha_a])} \leq ?(\beta_{x_1})$
4	$success$

Notice how we can solve directly also a list of constraints with circular

dependencies.

4.5 Type inference or Type checking

Until now we have been vague in the usage of the terms type inference and type checking. Actually what we have done is a hybrid system between type inference and type checking. A type checker takes in input a process with the binders annotated and output true or false whether or not the type annotations match the process behavior. This mean that the user has to specify manually the type of each bound name, which can be a real problem as the complexity of types increases very fast. On the other hand type inference takes a process without type annotation and tries to discover whether there exists a type assignment for the bound names that makes the process well typed. Now it is clear that our `solve` algorithm (Figure 23) neither requires the type annotations nor returns a full type assignment. We tackled the problem from this perspective since it is simpler but of course if `solve` algorithms are correct they must manage the real types at some point. Moreover since we are in presence of subtyping we are able to return a list of inequalities that a type must satisfies, not a closed equality; that is we are able to say "according to this process the type of a certain service must respect these constraints" and not always "the type of this service is this".

We now informally describe the steps one has to follow to obtain information about types. First of all instead of leaving to `INF` the management of the standard type environment we directly input to `INF` a type environment which already has a fresh type variable for each bound name. Remember that `INF` inputs a process with each bound name different from the free names and from the other bound names. With this modification we have the control of types contained in the typing environment. At this point the `solve` algorithm takes in input the standard environment too and whenever it applies a substitution to the set of constrains it applies the same substitution to each type contained in the environment. Moreover instead of removing inequalities we keep them into a list. When `solve` succeeds, in the environment we will find a type variable for each bound name and in the list all constraints the type variable must obey. Here the modified version of `solve`:

let rec solve $(\mathcal{C}, \Gamma, \mathcal{L})$ = match $\mathcal{C}$ with

1 $\mid S_1 = S_2, \mathcal{C}'$ -> solve $\sigma_{S_1=S_2}(\mathcal{C}', \Gamma, \mathcal{L})$

2 $\mid T_1 \leq T_2, \mathcal{C}'$ when $\mathsf{fv}(T_1) \cup \mathsf{fv}(T_2) = \mathsf{fcv}(T_1) \cup \mathsf{fcv}(T_2)$ -> solve $\sigma_{T_1 \leq T_2}(\mathcal{C}', \Gamma, \mathcal{L})$

3 $\mid \alpha \leq T_1, \alpha \leq T_2, \mathcal{C}'$ -> solve $\sigma_{T_1 \wedge T_2}(\mathcal{C}' \cup \{\alpha \leq T_1 \wedge T_2\}, \Gamma, \mathcal{L})$

4 $\mid T_1 \leq \alpha, T_2 \leq \alpha, \mathcal{C}'$ -> solve $\sigma_{\overline{T_1} \wedge \overline{T_2}}(\mathcal{C}' \cup \{\overline{\overline{T_1} \wedge \overline{T_2}} \leq \alpha\}, \Gamma, \mathcal{L})$

5 $\mid \alpha \leq T_1, T_2 \leq \alpha, \mathcal{C}'$ -> solve $(\mathcal{C}' \cup \{T_2 \leq T_1\} \cup \{\alpha \leq T_1\}, \Gamma,$
$\mathcal{L} \cup \{\alpha \leq T_1, T_2 \leq \alpha\})$

6 $\mid \alpha_1 = \overline{\alpha_2}, \alpha_1 \leq T_1, \alpha_2 \leq T_2, \mathcal{C}'$ -> solve $(\mathcal{C}' \cup \{\overline{T_1} \leq T_2\}, \Gamma,$
$\mathcal{L} \cup \{\alpha_1 \leq T_1, \alpha_2 \leq T_2\})$

7 $\mid T \leq T_1, \mathcal{C}'$-> solve $(\mathcal{C}', \Gamma, \mathcal{L})$

8 $\mid$_ -> $(\Gamma, \mathcal{L})$

The invocation of `solve` is intended with both $\mathcal{C}$ and Γ returned by `INF` and $\mathcal{L}$ initially empty. In particular in lines 1 and 2 we apply the respective substitution to Γ and $\mathcal{L}$ too and in line 5 and 6 we save the couple of constraints before removing them.

Example 4.37. Let us take the set of constraints generated in the Example 4.36

$$\begin{array}{ll}
\mathcal{C} & \begin{array}{l} \beta_a = [\alpha_a] \quad \beta_x = [\alpha_x] \quad \beta_b = [\alpha_b] \quad \alpha_a \leq ?(\beta_x) \\ \overline{!(\beta_a)} \leq \alpha_x \quad \overline{!(\beta_b)} \leq \alpha_a \quad \alpha_b \leq ?(\beta_{x_1}) \end{array} \\
\Gamma & a : \beta_a \quad x : \beta_x \quad x_1 : \beta_{x_1} \quad b : \beta_b \\
\mathcal{L} & \emptyset \\
 & \\
\mathcal{C} & \overline{!([\alpha_b])} \leq ?([\alpha_x]) \quad \overline{!([\alpha_a])} \leq \alpha_x \quad \alpha_b \leq ?(\beta_{x_1}) \\
\Gamma & a : [\alpha_a] \quad x : [\alpha_x] \quad x_1 : \beta_{x_1} \quad b : [\alpha_b] \\
\mathcal{L} & \alpha_a \leq ?([\alpha_x]) \quad \overline{!([\alpha_b])} \leq \alpha_a \\
 & \\
\mathcal{C} & \overline{!([\alpha_a])} \leq ?(\beta_{x_1}) \\
\Gamma & a : [\alpha_a] \quad x : [\alpha_b] \quad x_1 : \beta_{x_1} \quad b : [\alpha_b] \\
\mathcal{L} & \alpha_a \leq ?([\alpha_b]) \quad \overline{!([\alpha_b])} \leq \alpha_a \quad \overline{!([\alpha_a])} \leq \alpha_b \quad \alpha_b \leq ?(\beta_{x_1}) \\
 & \\
\Gamma & a : [\alpha_a] \quad x : [\alpha_b] \quad x_1 : [\alpha_a] \quad b : [\alpha_b] \\
\mathcal{L} & \alpha_a \leq ?([\alpha_b]) \quad \overline{!([\alpha_b])} \leq \alpha_a \quad \overline{!([\alpha_a])} \leq \alpha_b \quad \alpha_b \leq ?([\alpha_a])
\end{array}$$

The last line is the result of `solve`: the environment and the constraints it satisfies. For constraints contained in $\mathcal{L}$ we can use Lemma 4.25 ($\leq$ implies $\lessgtr$) and by the symmetry of $\lessgtr$ we have only: $\alpha_b \lessgtr ?([\alpha_a])$ and $\alpha_a \lessgtr ?([\alpha_b])$. At this point, using unification of infinite trees (since $\lessgtr$ implies =) taking $\alpha_a = \mu\alpha.?([?([\alpha])])$ and $\alpha_b = ?[\mu\alpha.?([?([\alpha])])]$ we are ok.

Example 4.38. Let us take the set of constraints generated in the example 4.35, the modified `solve` returns:

$$
\begin{array}{ll}
\Gamma & a : [\alpha_a] \\
\mathcal{L} & \alpha_a \leq (\mu\alpha_{Y_2}.\&\{l : \oplus\{l_1 : \alpha_{Y_2}\}\} \wedge \mu\alpha_{Y_2}.\&\{l : \oplus\{l_2 : \alpha_{Y_2}\}\} \\
 & \wedge\mu\alpha_{Y_1}.\&\{l : \oplus\{l_1 : \alpha_{Y_1}\}\} \wedge \mu\alpha_{Y_1}.\&\{l : \oplus\{l_3 : \alpha_{Y_1}\}\}) \\
 & (\overline{\mu\alpha_X . \oplus \{l : \&\{l_1 : \alpha_X\}\}} \vee \overline{\mu\alpha_X . \oplus \{l : \&\{l_2 : \alpha_X\}\}} \\
 & \vee\overline{\mu\alpha_X . \oplus \{l : \&\{l_3 : \alpha_X\}\}} \vee \overline{\mu\alpha_X . \oplus \{l : \&\{l_4 : \alpha_X\}\}}) \leq \alpha_a
\end{array}
$$

Here α_a is not in a closed form but it is between the intersection and the union of some inequalities.

In the previous example we might return the type of the service a, but however this is not a desirable feature since in this manner we limit the range of possibilities if we want to put into the system another definition of the service a. For instance suppose we want to know the type of a. To this end we remove inequalities relative to the service invocation (since they are fulfilled by the fact the `solve` succeeds) and we retain only those relative to the definition of a. At this point we can judge the type of a as $[\mu\alpha_Y.\&\{l : \oplus\{l_1 : \alpha_Y, l_2 : \alpha_Y, l_3 : \alpha_Y\}\}]$ (which is a minorant of the real intersection but it is not a problem). Suppose we have another specification of a which behaves like $[\mu\alpha_Y.\&\{l : \oplus\{l_1 : \alpha_Y, l_4 : \alpha_Y\}\}]$ we can only conclude that this specification is not compatible with the type of a since $\mu\alpha_Y.\&\{l : \oplus\{l_1 : \alpha_Y, l_2 : \alpha_Y, l_3 : \alpha_Y\} \leq \mu\alpha_Y.\&\{l : \oplus\{l_1 : \alpha_Y, l_4 : \alpha_Y\}\}$ does *not* hold. However this is not the case since running `solve` once again with the set of constraints in $\mathcal{L}$ together with the constraint $\alpha_a \leq \mu\alpha_Y.\&\{l : \oplus\{l_1 : \alpha_Y, l_4 : \alpha_Y\}\}$ we obtain the algorithm succeeds. Another problem is due to the fact that certain specifications give raise to a type with type variables such as the service $poly.(x)$.`return` x. The only inequality we can extract from this specification is $\alpha_{poly} \leq ?(\beta)$. At this point one have two ways to proceed either say that α_{poly} is a polymorphic type such as $\forall\beta.?(\beta)$ or instantiate β with some type. Both the previous ways are wrong. The first one with polymorphism, since two clients that use α_{poly} to send a value of different types are not typable in our system. The second one, since we can instantiate β with a type different from the actual client need.

4.6 A sound and complete syntax directed type system

In previous section we discuss a very naive approach to the reconstruction of recursive session types. Unfortunately the resulting system

is not complete e.g. $a : [!(int)]; \emptyset \vdash \texttt{rec } X.a.\langle 5 \rangle.X : \texttt{end}; \texttt{end}; \emptyset$ holds but it does not have a counterpart in $\vdash_{\mathsf{sd}}$. In this section we introduce a new syntax directed type system sound and complete. It comes out that not only the number of constraints grows faster than in the previous type system but also that the constraints resolution algorithm has an exponential complexity. We give details of the solving algorithm in the next chapter, since it is also capable to solve constraints generated from HVK-X calculus with session delegation.

The new syntax directed type system is reported in Figure 24.

We overload the symbol $\vdash_{\mathsf{sd}}$ without confusion since type judgments are of the form $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T; U; L$, i.e. sets disappear from judgments because we generate a new constraint whenever the type system in Figure 20 required union among sets (namely, rules (TIFSD) and (TBRANCHSD), notice the new definition of the function same). The recursion is handled constraining also assumptions contained in the process typing environment (rule (TRECSD)) (notice rule (TRECSD) requires the consistency check that in the previous type system was automatically solved by the function $\mu\alpha(T)$) and the rule (TPARLRSD) is not necessary anymore. Other rules are similar to the previous type system.

This time the syntax directed type system produces exactly the *same judgments* as the non-syntax directed version.

Theorem 4.39 (Soundness). *If* $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T; U; L$ *then* $\Gamma; \Theta \vdash P : T; U; L$.

Proof. The proof is by induction on the typing derivation of $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T; U; L$ with case analysis on the last applied rule. In the base case when the last applied rule is (TNILSD) we have $\Gamma; \Theta \vdash_{\mathsf{sd}} \mathbf{0} : \texttt{end}; \texttt{end}; \emptyset$ and $\Gamma; \Theta \vdash \mathbf{0} : \texttt{end}; \texttt{end}; \emptyset$. In the base case when the last applied rule is (TPVARSD) we have $\Gamma; \Theta, X : T; U \vdash_{\mathsf{sd}} X : T; U; \emptyset$ and then $\Gamma; \Theta, X : T; U \vdash_{\mathsf{sd}} X; T; U; \emptyset$. In the inductive cases when the last applied rule is:

- (TIFSD) we have $\dot{P} = \texttt{if } v_1 = v_2 \texttt{ then } P_1 \texttt{ else } P_2$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i : T_i; U_i; L_i$ and $i \in \{1, 2\}$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} \dot{P} : T; U; L$ and $T \leq T_i$ and $U_i \lesseqqgtr U$. By induction $\Gamma; \Theta \vdash P_i : T_i; U_i; L$. Since $T \leq T_i$ applying (TWEAK) $\Gamma; \Theta \vdash P_i : T; U; L$ and we can conclude $\Gamma; \Theta \vdash \dot{P} : T; U; L$.

- (TRECSD) we have $\dot{P} = \texttt{rec } X.P$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} \dot{P} : T; U; \emptyset$ and $\Gamma, X : T; U; \Theta \vdash_{\mathsf{sd}} P : T'; U'; \emptyset$ with $T \leq T'$ and $U \lesseqqgtr U'$. By induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U'; \emptyset$ and the result follows applying (TWEAK) to replace T' with T and rule (REC) to conclude.

(TNEWRSD)
$$\frac{\Gamma, r^+ : [T], r^- : [\overline{T}]; \Theta \vdash_{\mathsf{sd}} P : T'; U; L \quad |L \cap \{r^+, r^-\}| \neq 1}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\nu r)P : T'; U; L \setminus \{r^+, r^-\}}$$

(TNILSD)
$$\Gamma; \Theta \vdash \mathbf{0} : \mathsf{end}; \mathsf{end}; \emptyset$$

(TNEWSD)
$$\frac{\Gamma, a : S; \Theta \vdash_{\mathsf{sd}} P : T; U; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\nu a)P : T; U; L}$$

(TIFSD)
$$\frac{\Gamma \vdash v_i : S \quad \Gamma; \Theta \vdash_{\mathsf{sd}} P_i : T_i; U_i; L \quad T \leq T_i \quad U \lessgtr U_i \quad i = 1, 2}{\Gamma; \Theta \vdash_{\mathsf{sd}} \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q : T; U; L}$$

(TRECSD)
$$\frac{\Gamma; \Theta, X : T; U \vdash_{\mathsf{sd}} P : (T'; U'; \emptyset)^* \quad T \leq T' \quad U \lessgtr U'}{\Gamma; \Theta \vdash_{\mathsf{sd}} \mathtt{rec}\ X.P : T; U; \emptyset}$$

(TPVARSD)
$$\Gamma; \Theta, X : T; U \vdash_{\mathsf{sd}} X : T; U; \emptyset$$

(TSESSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U'; L \quad \Gamma \vdash r^p : [T] \quad T \leq T'}{\Gamma; \Theta \vdash_{\mathsf{sd}} r^p \triangleright P : U'; \mathsf{end}; L \uplus \{r^p\}}$$

(TDEFSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U; L \quad \Gamma \vdash v : [T] \quad T \leq T'}{\Gamma; \Theta \vdash_{\mathsf{sd}} v.P : U; \mathsf{end}; L}$$

(TINVSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U; L \quad \Gamma \vdash v : [T] \quad \overline{T'} \leq T}{\Gamma; \Theta \vdash_{\mathsf{sd}} \overline{v}.P : U; \mathsf{end}; L}$$

(TINSD)
$$\frac{\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash_{\mathsf{sd}} P : T; U; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\tilde{x}).P : ?(\tilde{S}).T; U; L}$$

(TOUTSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : T; U; L \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma; \Theta \vdash_{\mathsf{sd}} \langle \tilde{v} \rangle.P : !(\tilde{S}).T; U; L}$$

(TBRANCHSD)
$$\frac{I = \{1, \ldots, n\} \quad \forall i \in I,\ \Gamma; \Theta \vdash_{\mathsf{sd}} P_i : T_i; U_i; L \quad U \lessgtr U_i}{\Gamma; \Theta \vdash_{\mathsf{sd}} \Sigma_{i=1}^{n} (l_i).P_i : \&\{\mathsf{same}((l_i : T_i)_{i \in I})\}; U; L}$$

(TRETSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : T; U; L \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma; \Theta \vdash_{\mathsf{sd}} \mathtt{return}\ \tilde{v}.P : T; !(\tilde{S}).U; L}$$

(TCHOICESD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : T; U; L}{\Gamma; \Theta \vdash_{\mathsf{sd}} \langle l \rangle.P : \oplus\{l : T\}; U; L}$$

(TPARLSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : T; U; L_1 \quad \Gamma; \Theta \vdash_{\mathsf{sd}} Q : \mathsf{end}; \mathsf{end}; L_2}{\Gamma; \Theta \vdash_{\mathsf{sd}} P|Q : T; U; L_1 \uplus L_2}$$

(TPARRSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P : \mathsf{end}; \mathsf{end}; L_1 \quad \Gamma; \Theta \vdash_{\mathsf{sd}} Q : T; U; L_2}{\Gamma; \Theta \vdash_{\mathsf{sd}} P|Q : T; U; L_1 \uplus L_2}$$

(TPIPESD)
$$\frac{\Gamma; \emptyset \vdash_{\mathsf{sd}} P : T_1; \mathsf{end}; L \quad \Gamma, \tilde{x} : \tilde{S}; \Theta \vdash_{\mathsf{sd}} Q : T_2; U_2; \emptyset \quad (T, U) = \mathsf{pipe}(T_1, T_2, U_2, \tilde{S})}{\Gamma; \Theta \vdash_{\mathsf{sd}} P > \tilde{x} > Q : T; U; L}$$

$$\begin{aligned}
\mathsf{same}(l_1 : T_1, \ldots, l_n : T_n) \quad &= \quad \begin{cases} l_1 : T_1, \mathsf{same}(l_2 : T_2 \ldots, l_n : T_n) & \text{if } l_1 \notin \{l_2, \ldots, l_n\} \\ \mathsf{same}(l_2 : T_2, \ldots, l_i : U, \ldots, l_n : T_n) & \text{if } \begin{array}{c} l_1 = l_i \text{ and} \\ U \leq T_1, T_i \end{array} \end{cases} \\
\mathsf{same}(l : T) \quad &= \quad l : T
\end{aligned}$$

Figure 24: Syntax directed typing rules

- (TSESSD) we have $\dot{P} = r^p \triangleright P$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U; L$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} \dot{P} : \mathtt{end}; U$ and $\Gamma \vdash r^p : [T]$ and $T \leq T'$. By induction $\Gamma; \Theta \vdash P : T'; U; L$. The result follows applying (TWEAK) to replace T' with T and (TSES) to conclude.

- (TDEFSD) similar to the previous case.

- (TINVSD) we have $\dot{P} = \overline{v}.P$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U; L$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} \dot{P} : \mathtt{end}; U$ and $\Gamma \vdash v : [T]$ and $\overline{T'} \leq T$ or $\overline{T} \leq T'$. The result follows applying (TWEAK) to replace $\overline{T'} \leq T$ with $\overline{T}$ and (TINV) to conclude.

- (TBRANCHSD) the result follows by induction on the typing of each P_i and applying the rule (TWEAK) for each inequality generated by same.

- remaining rules follows directly by induction.

□

But also the completeness of the type system holds, since the syntax directed type system is able to simulate every judgments produced by $\vdash$.

Theorem 4.40 (Completeness). *If $\Gamma; \Theta \vdash P : T; U; L$ then there exists T', U' s.t. $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U'; L$ and $T \leq T'$ and $U \lesseqgtr U'$.*

Proof. The proof is by induction on the typing derivation of $\Gamma; \Theta \vdash P : T; U; L$ with case analysis on the last applied rule. In the base cases when the last applied rule is (TNIL) the result follows directly. Otherwise, when the last applied rule is (TPVAR) we have $\Gamma; \Theta, X : T; U \vdash X : T; U; \emptyset$ and $\Gamma; \Theta, X : T; U \vdash_{\mathsf{sd}} X : T; U; \emptyset$ and the result follows by reflexivity of $\leq$ and $\lesseqgtr$. In the inductive cases when the last applied rule is:

- (TIF) we have $\dot{P} = \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ and $\Gamma; \Theta \vdash \dot{P} : T; U; L$. Moreover by induction hypothesis we have for $i \in \{1, 2\}$ that $\Gamma; \Theta \vdash P_i : T; U; L$ implies that there exist T_1, T_2, U_1, U_2 and that $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i : T_i; U_i; L$ with $T \leq T_i$ and $U \lesseqgtr U_i$. The result follows by an application of the rule (TIFSD).

- (TREC) we have $\dot{P} = \mathtt{rec}\ X.P$ and $\Gamma; \Theta \vdash \dot{P} : T; U; \emptyset$ and $\Gamma; \Theta, X : T; U \vdash P : T; U; \emptyset$. By induction we have $\Gamma; \Theta, X : T; U \vdash_{\mathsf{sd}} P : T'; U'; \emptyset$ for some T',U' s.t. $T \leq T'$ and $U \lesseqgtr U'$ and an application of the rule (TREC) concludes this case.

- (TSES) we have $\dot{P} = r^p \triangleright P$ and $\Gamma; \Theta \vdash P : T; U; L$ and $\Gamma; \Theta \vdash \dot{P} : \mathtt{end}; U$ and $\Gamma \vdash r^p : [T]$. By induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U'; L$ for some T', U' s.t. $T \leq T'$ and $U \lesseqgtr U'$. The result follows applying (TSESSD).

- (TDEF) similar to the previous case.

- (TINV) we have $\dot{P} = \overline{v}.P$ and $\Gamma; \Theta \vdash P : \overline{T}; U; L$ and $\Gamma; \Theta \vdash \dot{P} : \mathtt{end}; U$ and $\Gamma \vdash v : [T]$. By induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P : T'; U'; L$ for some T', U' and $T' \leq \overline{T}$ or $\overline{T'} \leq T$ which allows an application of the rule (TINV) to conclude.

- (TIN) we have $\dot{P} = (\tilde{x}).P$ and $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P : T; U; L$ and $\Gamma; \Theta \vdash (\tilde{x}).P : ?(\tilde{S}).T; U; L$. By induction $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P : T'; U'; L$ where $T \leq T'$ and $U' \lesseqgtr U$. The result follows since by definition of $\leq$, $?(\tilde{S}).T \leq ?(\tilde{S}).T'$.

- (TOUT) and (TRET) similar to the previous case.

- (TCHOICE) similar to case for (TIN) together with the fact that $\oplus\{l : T\} \;;;\; \oplus\{l_i : T_i\}_{i \in I} \leq \oplus\{l : T\}$ holds for any $\oplus\{l_i : T_i\}_{i \in I}$ by definition of $\leq$.

- (TBRANCH) we have $\dot{P} = \Sigma_{i=1}^{n}(l_i).P_i$ and $\Gamma; \Theta \vdash P_i : T_i; U; L$ for all i and $\Gamma; \Theta \vdash \dot{P} : \&\{l_i : T_i\}_{i \in J}; U; L$ where $J \subseteq I = \{1, \ldots, n\}$. In addition $\forall i \forall j \in I \; i \neq j$ and $l_i = l_j$ implies $T_i = T_j$ since by well-formedness condition two equal labels cannot appear in the type, that is at the type level the type system shrinks the type relative to two branches with equal label until a common type is found (if it exists). By induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i : T_i'; U_i'; L$ for some T_i' and U_i' s.t. $T_i \leq T_i'$ and $U \lesseqgtr U_i'$. The last equality satisfies the theorem conclusion, but to conclude we must still prove $\&\{l_i : T_i\}_{i \in J} \leq \&\{\mathsf{same}(l_i : T_i')_{i \in I}\}$ for each $J \subseteq I$ and $T_i \leq T_i'$. We note that J contains less or equal branches than $\&\{\mathsf{same}(l_i : T_i')_{i \in I}\}$ by the well-formedness condition and in addition each time the same function is computed with the second case (i.e. the case with two equal labels) we have $V \leq T_1'$ and $V \leq T_i'$ for some i but as discussed above $T_1 = T_i$ then we can select V to be equal to T_i. At this point the result follows by the definition of $\leq$.

- (TWEAK) follows directly by induction and by the transitivity of $\leq$.

- remaining rules follows directly by induction.

$\square$

We report in Figure 25 the corresponding algorithm to extract constraints from the type system in Figure 24. The most interesting rules are: the rule relative to recursion which behaves exactly as the rule for recursion in the previous type system adding a type variable in the assumptions and both the rules for parallel composition and pipe which non-deterministically have different returning values offered by means of the **or** keyword. This is due to the fact that we cannot statically know the types of P and Q in both the current protocol and the parent protocol. Thus we can think of INF as returning *a list of sets* of constraints and we must check if at least one constraint set in the list is satisfiable. Moreover, here we do not try to solve the recursion on-the-fly (i.e. via the function $\mu\alpha()$) but instead we generate a constraint: it is a task of the constraint solver algorithm to solve the recursion. In the case relative to the external choice we use an overloaded function same similar to the original one but in addition we collect a list of constraints too.

$$\begin{array}{lll}
\mathsf{same}(l_1 : T_1, \ldots, l_n : T_n) & = & \begin{cases} l_1 : T_1, \mathsf{same}(l_2 : T_2 \ldots, l_n : T_n) & \text{if } l_1 \notin \{l_2, \ldots, l_n\} \\ \mathsf{same}(l_2 : T_2, \ldots, l_i : \alpha, \ldots, l_n : T_n) & \text{if } \begin{array}{l} l_1 = l_i \text{ and} \\ \alpha \leq T_1, T_i \\ \text{and } \alpha \text{ fresh} \end{array} \end{cases} \\
\mathsf{same}(l : T) & = & l : T
\end{array}$$

We abbreviate a list $l_1 : T_1, \ldots, l_n : T_n$ as $\widetilde{l : T}$ and we write $(\mathcal{C}, \widetilde{l' : T'}) = \mathsf{same}(l_1 : T_1, \ldots, l_n : T_n)$ to obtain the resulting pair (the set of constraints and the list of labels/types) from the above defined function and only $\widetilde{l' : T'} = \mathsf{same}(l_1 : T_1, \ldots, l_n : T_n)$ to obtain the result from the function defined in Figure 24. It is simple to see that the following lemma holds:

Lemma 4.41. *Let* $(\mathcal{C}, \widetilde{l' : T'}) = \mathsf{same}(l_1 : T_1, \ldots, l_n : T_n)$. *If* $\sigma \models \mathcal{C}$ *then* $\widetilde{l' : \sigma T'} = \mathsf{same}(l_1 : \sigma T_1, \ldots, l_n : \sigma T_n)$.

Proof. A direct consequence of the two definitions. $\square$

Also the correspondence theorem between INF and the type system has a slight different formulation since the solving substitution is applied to the process environment too.

Proposition 4.42 (Soundness and Completeness of INF). *Let* $\mathtt{INF}(P, \Gamma, \Theta) = (\mathcal{C}, T, U, L)$. $\sigma \models \mathcal{C}$ *iff* $\sigma\Gamma; \sigma\Theta \vdash_{\mathsf{sd}} P : \sigma T; \sigma U; L$.

VALUEINF(x,Γ) = $\Gamma(x)$ VALUEINF(s,Γ) = $\Gamma(s)$ VALUEINF(n,Γ) = (int)

INF$(\mathbf{0},\Gamma,\Theta)$ = $(\emptyset,\text{end},\text{end},\emptyset)$

INF$(v.P,\Gamma,\Theta)$ = **let** $(\mathcal{C},T,U,L)$=INF$(P,\Gamma,\emptyset)$

in $(\mathcal{C}\cup\{\alpha\leq\ T\}\cup\{\Gamma(v)=[\alpha]\},U,\text{end},L)$ (where α fresh)

INF$(\overline{v}.P,\Gamma,\Theta)$ = **let** $(\mathcal{C},T,U,L)$=INF$(P,\Gamma,\emptyset)$

in $(\mathcal{C}\cup\{\overline{T}\leq\alpha\}\cup\{\Gamma(v)=[\alpha]\},U,\text{end},L)$ (where α fresh)

INF$((x_1,\ldots,x_n).P,\Gamma,\Theta)$ =

let $(\mathcal{C},T,U,L)$=INF$(P,\Gamma\cup\{x_1:\beta_1,\ldots,x_n:\beta_n\},\Theta)$ $\beta_1,.,\beta_n$ fresh

in $(\mathcal{C},?(\beta_1,\ldots,\beta_n).T,U,L)$

INF$(\langle v_1,\ldots,v_n\rangle.P,\Gamma,\Theta)$ = **let** $(\mathcal{C},T,U,L)$=INF(P,Γ,Θ) **in**

let S_1=VALUEINF(v_1,Γ) ...... S_n=VALUEINF(v_n,Γ)

in $(\mathcal{C},!(S_1,\ldots,S_n).T,U,L)$

INF(return $v_1,\ldots,v_n.P,\Gamma,\Theta$) = **let** $(\mathcal{C},T,U,L)$=INF(P,Γ,Θ) **in**

let S_1=VALUEINF(v_1,Γ) ...... S_n=VALUEINF(v_n,Γ)

in $(\mathcal{C},T,!(S_1,\ldots,S_n).U,L)$

INF(if $v_1=v_2$ **then** P **else** Q,Γ,Θ) = **let** $(\mathcal{C},T,U,L)$=INF(P,Γ,Θ) **in**

let $(\mathcal{C}_1,T_1,U_1,L)$=INF$(Q,\Gamma,\Theta)$ **in**

let S_1=VALUEINF(v_1,Γ) S_2=VALUEINF(v_2,Γ)

in $(\mathcal{C}\cup\mathcal{C}_1\cup\{S_1=S_2,\alpha\leq T_1,\alpha\leq T_2,\alpha_1\lesseqgtr U_1,\alpha_2\lesseqgtr U_2\},\alpha,\alpha_1,L)$

(where α,α_1 fresh)

INF$((\nu a)P,\Gamma,\Theta)$=INF$(P,\Gamma\cup\{a:\beta\},\Theta)$ (where β fresh)

INF$((\nu r)P,\Gamma,\Theta)$ =

let $(\mathcal{C},T,U,L)$=INF$(P,\Gamma\cup\{r^+:[\alpha_1],r^-:[\alpha_2]\},\Theta)$ (where α_1,α_2 fresh)

in $(\mathcal{C}\cup\{\alpha_1=\overline{\alpha_2}\},T,U,L\setminus\{r^+,r^-\})$

(where $L\cap\{r^+,r^-\}=\{r^+,r^-\}$ or $\emptyset$)

INF$(P|Q,\Gamma,\Theta)$ = **let** $(\mathcal{C},T,U,L_1)$= INF(P,Γ,Θ) **in**

let $(\mathcal{C}_1,T_1,U_1,L_2)$=INF$(Q,\Gamma,\Theta)$

in $(\mathcal{C}\cup\mathcal{C}_1\cup T_1=\text{end},U_1=\text{end},T,U,L_1\uplus L_2)$

or $(\mathcal{C}\cup\mathcal{C}_1\cup T=\text{end},U=\text{end},T_1,U_1,L_1\uplus L_2)$

INF$(\Sigma_{i=1}^n(l_i).P_i,\Gamma,\Theta)$ =

let $(\mathcal{C}_1,T_1,U_1,L)$=INF$(P_1,\Gamma,\Theta)$ ...... $(\mathcal{C}_n,T_n,U_n,L)$=INF$(P_n,\Gamma,\Theta)$

in let $(\mathcal{C}',\widetilde{l':T'})=\text{same}(l_1:T_1,\ldots,l_n:T_n)$

in $(\mathcal{C}'\cup\mathcal{C}_1\cup\ldots\cup\mathcal{C}_n\cup\{U\lesseqgtr U_1,\ldots,U\lesseqgtr U_n\},\&\{\widetilde{l':T'}\},U,L)$

where each type variables generated by same is fresh

INF$(\langle l\rangle.P,\Gamma,\Theta)$=**let** $(\mathcal{C},T,U,L)$= INF(P,Γ,Θ) **in** $(\mathcal{C},\oplus\{l:T\},U,L)$

INF$(P>x_1,\ldots,x_n>Q,\Gamma,\Theta)$ = **let** $(\mathcal{C},T,U,L)$=INF$(P,\Gamma,\emptyset)$

in let$(\mathcal{C}_1,T_1,U_1,\emptyset)$=INF$(Q,\Gamma\cup\{x_1:S_1,\ldots,x_n:S_n\},\Theta)$

$(\mathcal{C}\cup\mathcal{C}_1\cup\{T=\text{end},U=\text{end}\},\{\text{end}\},\{\text{end}\},\text{L})$

or $(\mathcal{C}\cup\mathcal{C}_1\cup\{T=!(\beta),U=\text{end}\},T_1,U_1,L)$

where β fresh

INF$(r^p\triangleright P,\Gamma,\Theta)$ = **let** $(\mathcal{C},T,U,L)$=INF$(P,\Gamma,\emptyset)$

in $(\mathcal{C}\cup\{\alpha\leq\ T\}\cup\{\Gamma(r^p)=[\alpha]\},U,\text{end},(L\uplus\{r^p\}))$

(where α fresh)

INF(rec $X.P,\Gamma,\Theta$) = **let** $(\mathcal{C},T,U,\emptyset)$=INF$(P,\Gamma,\Theta\cup\{X:\alpha_1;\alpha_2\})$

in if $nocuract(P)$ then $\mathcal{C}=\mathcal{C}\cup\{T=\text{end}\}$

in if $noretact(P)$ then $\mathcal{C}=\mathcal{C}\cup\{U=\text{end}\}$

in $(\mathcal{C}\cup\{\alpha_1\leq T,\alpha_2\lesseqgtr U\},\alpha_1,\alpha_2,\emptyset)$

(where α_1,α_2 fresh)

INF(X,Γ,Θ) = $(\emptyset$,(fst $\Theta(X)$),(snd $\Theta(X)$),$\emptyset)$

Figure 25: The algorithm to extract constraints in Ocaml-like syntax

Proof. The proof is by induction on the recursive structure of a run of $\mathtt{INF}(P, \Gamma, \Theta)$ and it is almost similar to the proof of Proposition 4.31, we report the inductive case relative to the recursion rule for the soundness part of the proof and the inductive case relative to parallel composition for the completeness part of the proof.

- We have $\dot{P} = \mathtt{rec}\ X.P$. Let $\mathcal{C}$ be the set of constraints generated by $\mathtt{INF}(P, \Gamma, \Theta)$. By induction $\sigma \vDash \mathcal{C}$ implies $\sigma\Gamma; \sigma\Theta, X : \sigma T'; \sigma U' \vdash_{\mathsf{sd}} P : \sigma T, \sigma U; L$. By hypothesis also holds $\sigma \vDash \mathcal{C} \cup \{T \leq T', U \lesseqgtr U'\}$ plus two optional constraints if either $nocuract(P)$ or $noretact(P)$ holds then we have (TRECSD) $\dfrac{\sigma\Gamma; \sigma\Theta, X : \sigma T'; \sigma U' \vdash_{\mathsf{sd}} P : (\sigma T, \sigma U; \emptyset)^\star \quad \sigma T \leq \sigma T' \quad \sigma U \lesseqgtr \sigma U'}{\sigma\Gamma; \sigma\Theta \vdash_{\mathsf{sd}} \mathtt{rec}\ X.P : \sigma T'; \sigma U'; \emptyset}$ which concludes.

- We have $\dot{P} = P_1 | P_2$ and by induction $\sigma\Gamma; \sigma\Theta \vdash P_i : \sigma T_i; \sigma U_i; L_i$ implies $\sigma \vDash \mathcal{C}_i$ where $\mathcal{C}_i$ are the sets of constraints generated by $\mathtt{INF}$ with input P_i for $i \in \{1, 2\}$. We have two cases if $\sigma T_1 = \mathtt{end}$ and $\sigma U_1 = \mathtt{end}$ we conclude with $\sigma \vDash \mathcal{C}_1 \cup C_2$ since $\sigma\Gamma; \sigma\Theta \vdash P|Q : \sigma T_2; \sigma U_2; L_1 \uplus L_2$ and $\mathtt{INF}(P|Q, \Gamma, \Theta) = (\mathcal{C}_1 \cup C_2, T_2, U_2, L_1 \uplus L_2)$ the other case $\sigma T_2 = \mathtt{end}$ and $\sigma U_2 = \mathtt{end}$, follows since also $\mathtt{INF}(P|Q, \Gamma, \Theta) = (\mathcal{C}_1 \cup C_2, T_1, U_1, L_1 \uplus L_2)$ (remember we use the `or` keyword to allows for both cases).

□

Example 4.43. We show the set of constraints generated by `INF` for the Example 4.1. Actually the result is a list of sets of constraints due to the various parallel compositions, but all except one contain the same constraints since no return action are performed. The different one has an additional constraint $\alpha_{X_1} = \mathtt{end}$.

$$\begin{gathered}\mathtt{end} = \mathtt{end} \quad \alpha_X \leq \oplus\{l : \&\{l_1 : \alpha_1, l_2 : \alpha_2, l_3 : \alpha_3, l_4 : \alpha_4\}\} \quad \alpha_1 \leq \alpha_X \quad \alpha_2 \leq \alpha_X \\ \alpha_3 \leq \alpha_X \quad \alpha_4 \leq \alpha_X \quad \overline{\alpha_X} \leq \alpha_a \quad \alpha_5 \leq \&\{l : \oplus\{l_1 : \alpha_{Y_2}\}\} \quad \alpha_6 \leq \&\{l : \oplus\{l_2 : \alpha_{Y_2}\}\} \\ \alpha_7 \leq \alpha_5 \quad \alpha_7 \leq \alpha_6 \quad \alpha_{Y_2} \leq \alpha_7 \quad \alpha_a \leq \alpha_{Y_2} \quad \alpha_8 \leq \oplus\{l_3 : \alpha_{Y_1}\} \quad \alpha_9 \leq \oplus\{l_1 : \alpha_{Y_1}\} \\ \alpha_{10} \leq \alpha_8 \quad \alpha_{10} \leq \alpha_9 \quad \alpha_{Y_1} \leq \alpha_{10} \quad \alpha_a \leq \alpha_{10}\end{gathered}$$

Compare the list of constraints generated in this example with the corresponding list of constraints generated for the same example by the previous `INF` algorithm. Moreover the `solve` algorithm defined if Figure 23 does not work anymore. As a simple counterexample consider the constraint $\overline{\alpha_X} \leq \alpha_a$ simply `solve` does not handle it or consider the constraints $\alpha_1 \leq \alpha_X$ and $\alpha_2 \leq \alpha_X$ the syntactic unifier relative to the meet

relation is not the most general one because the set of free variables is different from the set of communicated variables (Proposition 3.31).

We shall see in the next chapter that the set of constraints can be solved but with an exponential cost.

4.7 Concluding remarks on CST

In this chapter we have introduced CST, a session-typed variant of CaSPiS. First we have given a standard non syntax directed type system which not only assigns a type to each value and session in a process but checks session linearity. We have seen that session linearity is an important fact to keep typability along reduction steps. As a direct consequence of the subject reduction we have proved the so-called session safety, i.e. that a well-typed program cannot goes wrong. The type system is non deterministic in many ways: it allows the non-deterministic application of the rule (TWEAK), it allows to choose arbitrary branches in each choices, rules (TBRANCH) and (TCHOICE), and most importantly it guesses the type of process variables in rule (TREC). The latter fact is quite problematic since solving recursion involves a cyclic reasoning. Despite these problems the non-determinism helps in the proof of theorems, which result simpler.

Then, we have proposed a syntax directed type system which uses sets to multiplex types of different process-branches and on-the-fly resolution of the recursion. A type judgment in this set based type system is correct if it exists the greatest lower bound of each element in the set. Recursion resolution instead is based on the naive idea that if a process has a certain type T then the same process with recursion has a type of the form $\mu\alpha.T'$ where T' is obtained from T replacing the trailing occurrences of α with end. However this fact could not allow to achieve the completeness of the type system since a process variable cannot be nested inside a service like e.g. in rec $X.a.X$. Nevertheless we give a constraint extraction algorithm which extracts a set of constraints to be satisfied if and only if the process is well typed and an algorithm to solve these constraints. Since half of the work is done by the syntax directed type system it turns out that the solving algorithm has a quadratic complexity with respect to the number of constraints in input. We also prove that the solving algorithm is sound and complete w.r.t. the syntax directed type system in the sense that it solves all the constraints generated by the constraint extractor algorithm for a certain process and if it solves a set of constraint then it is guaranteed that the process is typable.

Finally we have introduced a new syntax directed type system which is, this time, sound and complete. We also have given for this type system a constraint extractor algorithm (sound and complete) and we have ended the chapter observing the number of constraints generated by the two different syntax directed versions of the type system for the same process. Not only the number of constraints has an impressive growth but the solving algorithm does not work anymore. The latter issue can be overcome by using the solving algorithm defined in the next Chapter.

Chapter 5

HVK-X a full session calculus

5.1 Introduction

In this chapter we introduce a variant of the language described in (76). The main difference with the original language is in the treatment of recursion. Here we choose to keep the full recursion which is provided in the original proposal by means of process definitions. These two paradigms in presence of replications are similar: one can be encoded in the other and vice versa (provided that the number of process definitions is finite (59)). We choose the general recursion to stay closer with CST and the π-calculus. Moreover, the presentation of the language is simplified; for instance we do not need all the structural congruence rules relative to the process definitions. We call this calculus HVK-X from the initials of the original authors together with X that stays for a general recursion variable.

From the type system point of view, the main difference regards the introduction of subtyping; by means of a specific weakening rule the type system can replace a type of an active session with one of its subtypes. The weakening rule is also used to insert at any time an assumption about a session with type `end`. This characteristic resembles the type system proposed in (26) while in the original proposal the ended sessions can be inserted within specific rules. Successively we try to eliminate the non determinism proposing a set of syntax directed rules from which

we extract a set of constraints. The satisfiability of these constraints implies the typability of the original process. We also study an algorithm to automatically solve these constraints which is more complicated than the previous `solve` (see Figure 23). This time the presence of session delegation together with recursion and name extrusion results in an algorithm with an exponential cost. In the previous `solve` we actively use the intersection, here, to account for the presence of free variables, we merge each constraint multiplexing them in each possible context. We prove the soundness and the completeness of the proposed algorithm; while the soundness holds the completeness direction holds for closed processes without free names. This limitation is also due to the high complexity solving locally a set of constraints.

We conclude providing two encoding functions from HVK-X to π-calculus and from HVK-X to CST. Instead of proving some equivalence between the operational rules of two calculi (62) we prove the goodness of our encodings showing the correspondence between type systems. In particular, for the first encoding we prove the soundness and the completeness of the proposed type system with respect to the simply typed π-calculus. For the second encoding instead we prove the soundness and the completeness of the proposed type system with respect to the type system proposed in Chapter 4. This result is not obvious due to the substantial differences between these calculi and can be viewed as the proof of the existence of the solving algorithm which is missed at the end of the previous chapter. In fact we can use both encodings to check the well-typedness for π-calculus and CST by implementing just the one for HVK-X. On the other hand encodings can be used to find problems in different type systems; in fact thanks to this encoding we discovered a lot of problems in the earlier version of CST type system.

Background. The calculus used here was first proposed in (76) and inspired on (37) where session were considered for the fist time. We modify the original proposal removing process definition replaced by the general recursion. We however modified typing rules since our main concern is about type inference. All remaining contents are introduced here.

5.2 Syntax and operational semantics

As in the case of CST we assume an infinite collection $r, s \ldots$ of session names, an infinite collection $a, \ldots$ of service names, an infinite collection

P, Q	$::=$	$\mathbf{0}$	(nil)
	$\mid$	$v(\mathbf{x}).P$	(session acceptance)
	$\mid$	$\overline{v}(\mathbf{x}).P$	(session request)
	$\mid$	$\kappa!(\tilde{v}).P$	(output)
	$\mid$	$\kappa?(\tilde{x}).P$	(input)
	$\mid$	$\kappa!(l).P$	(label selection)
	$\mid$	$\Sigma_{i=1}^{n}\kappa?(l_i).P_i$	(label branching)
	$\mid$	$\kappa!\langle\langle\kappa'\rangle\rangle.P$	(session sending)
	$\mid$	$\kappa?((\mathbf{x})).P$	(session receiving)
	$\mid$	$\texttt{if } v = w \texttt{ then } P \texttt{ else } Q$	(if-then-else)
	$\mid$	$P \mid Q$	(parallel)
	$\mid$	$(\nu m)P$	(restriction)
	$\mid$	$\texttt{rec } X.P$	(recursion)
	$\mid$	X	(process variable)
v, w	$::=$	$x, y, \ldots$	(variable)
	$\mid$	a	(service name)
	$\mid$	$\underline{n}$	(integers)
κ	$::=$	$\mathbf{x}, \mathbf{y}, \ldots$	(session var.)
	$\mid$	r^p	(session)

Figure 26: Syntax of HVK-X

of variables $x, y, \ldots$, an infinite collection of process variables $X, Y, \ldots$ and an infinite collection $l, \ldots$ of labels. Here in addition we use $\mathbf{x}, \mathbf{y}, \ldots$ to range over session variables and $\kappa, \ldots$ to range over both polarized sessions and session variables. The syntax of processes $P, Q, \ldots$ is defined in Figure 26. Two primitives $v(\mathbf{x}).P$ and $\overline{v}(\mathbf{x}).P$ model service declaration and service invocation respectively (which are called here service request and service accept). Each of the four primitives for session communications has a κ which is the session subject and it is used to specify the session in which communications happen e.g. $\Sigma_{i=1}^{n}\kappa?(l_i).P_i$ is the external choice with subject κ (the same subject for each offered branch). In addition, we can send and receive session sides by means of the two primitives: $\kappa!\langle\langle\kappa'\rangle\rangle.P$ which is read as, send the session κ' through the session κ, and $\kappa?((\mathbf{x})).P$ which is read as, receive a session through κ and bind it to the variable $\mathbf{x}$. In order to keep linearity of sessions when κ' is

delegated, by means of $\kappa!\langle\!\langle\kappa'\rangle\!\rangle.P$, it should not be used in P for communications.

Priority of the operators is the same as in the π-calculus so the parallel composition binds less than the restriction operator. Binders for the calculus are $\kappa?(\tilde{x}).P$ for the tuple of variables $\tilde{x}$ in P, $(\nu m)P$ for the name m in P, $a(\mathbf{x}).P$ and $\kappa?(\!(\mathbf{x})\!).P$ for the session variable $\mathtt{x}$ in P and $\mathtt{rec}\ X.P$ for the process variable X in P. The derived notion of free names (fn), defined in Figure 27, and bound names of a process, free process variables (fpv), defined in Figure 29, and bound process variables and closed process (when $\mathsf{fpv}(P) = \emptyset$) are standard. As before we define a more specific set of free polarized names that contains both session names and session variables. We denote such set as fpn and its definition is given in Figure 28.

Differently from CST in HVK-X each communication is annotated with the session subject, in order to open many sessions at the same time as well as interleaving communications in any order. This strategy allows to remove session nesting, the pipe and the return primitive and additionally it allows introducing session delegation.

The structural congruence relation $\equiv$ (Figure 30) is standard from the π-calculus. Differently from (76) we do not have to define the structural congruence for process definitions.

As usual thanks to the structural congruence we give the operational semantics of the calculus by grouping redexes.

(LINK)	$(a(\mathbf{x}).P)\vert(\overline{a}(\mathbf{y}).Q) \to (\nu r)(P[^{r^+}/_{\mathbf{x}}]\vert Q[^{r^-}/_{\mathbf{y}}])$	$r \notin \mathsf{fn}(P\vert Q)$
(COM)	$(r^p!(\tilde{v}).P)\vert(r^{\overline{p}}?(\tilde{x}).Q) \to (P\vert Q[^{\tilde{v}}/_{\tilde{x}}])$	$\vert\tilde{v}\vert = \vert\tilde{x}\vert$
(LABEL)	$(r^p!(l_k).P)\vert(\Sigma_{i=1}^n r^{\overline{p}}?(l_i).P_i) \to (P\vert P_i)$	$(1 \leq k \leq n)$
(PASS)	$(r^p!\langle\!\langle r'^q\rangle\!\rangle.P)\vert(r^{\overline{p}}?(\!(\mathbf{x})\!).Q) \to P\vert Q[^{r'^q}/_{\mathbf{x}}]$	$r \neq r'$
(IF1)	$\mathtt{if}\ v = v\ \mathtt{then}\ P\ \mathtt{else}\ Q \to P$	
(IF2)	$\mathtt{if}\ v = w\ \mathtt{then}\ P\ \mathtt{else}\ Q \to Q$	$v \neq w$
(REC)	$P[^{\mathtt{rec}\ X.P}/_X] \to P' \Rightarrow \mathtt{rec}\ X.P \to P'$	
(SCOP)	$P \to Q \quad \Rightarrow \quad (\nu m)P \to (\nu m)Q$	
(PAR)	$P \to P' \quad \Rightarrow \quad P\vert Q \to P'\vert Q$	
(STR)	$P \equiv P'$ and $P' \to Q'$ and $Q' \equiv Q \quad \Rightarrow \quad P \to Q$	

Rule (LINK) models the invocation of a service and then it creates a new fresh session r allowing P and Q to communicate. The communication model imposes that P and Q can communicate only if they have the same session with dual polarity, thus r^+ and r^- are substituted in place of the respective bound session variable. Rule (COM) is the standard message

$$
\begin{array}{lcl lcl}
\mathsf{fn}(\mathbf{0}) & = & \emptyset & \mathsf{fn}(\begin{array}{c}\texttt{if } v = w \texttt{ then}\\ P \texttt{ else } Q\end{array}) & = & \begin{array}{c}\mathsf{fn}(u) \cup \mathsf{fn}(w) \cup\\ \mathsf{fn}(P) \cup \mathsf{fn}(Q)\end{array}\\
\mathsf{fn}(v(\mathbf{x}).P) & = & \mathsf{fn}(v) \cup (\mathsf{fn}(P) \setminus \mathbf{x}) & & & \\
\mathsf{fn}(\overline{v}(\mathbf{x}).P) & = & \mathsf{fn}(v) \cup (\mathsf{fn}(P) \setminus \mathbf{x}) & \mathsf{fn}(P|Q) & = & \mathsf{fn}(P) \cup \mathsf{fn}(Q)\\
\mathsf{fn}(\kappa!(\tilde{v}).P) & = & \mathsf{fn}(\kappa) \cup \mathsf{fn}(\tilde{v}) \cup \mathsf{fn}(P) & \mathsf{fn}((\nu m)P) & = & \mathsf{fn}(P) \setminus \{m\}\\
\mathsf{fn}(\kappa?(\tilde{x}).P) & = & \mathsf{fn}(\kappa) \cup (\mathsf{fn}(P) \setminus \{\tilde{x}\}) & \mathsf{fn}(\texttt{rec } X.P) & = & \mathsf{fn}(P)\\
\mathsf{fn}(\kappa!(l).P) & = & \mathsf{fn}(\kappa) \cup \mathsf{fn}(P) & \mathsf{fn}(X) & = & \emptyset\\
\mathsf{fn}(\Sigma_{i=1}^{n}\kappa!(l_i).P_i) & = & \mathsf{fn}(\kappa) \cup \bigcup_{i=1}^{n} \mathsf{fn}(P_i) & \mathsf{fn}(\mathbf{x}) & = & \{\mathbf{x}\}\\
\mathsf{fn}(\kappa!\langle\!\langle \kappa' \rangle\!\rangle.P) & = & \mathsf{fn}(\kappa) \cup \mathsf{fn}(\kappa') \cup (P) & \mathsf{fn}(r^p) & = & \{r\}\\
\mathsf{fn}(\kappa?(\!(\mathbf{x})\!).P) & = & \mathsf{fn}(\kappa) \cup (P \setminus \{\mathbf{x}\}) & \mathsf{fn}(a) = \{a\} & & \mathsf{fn}(x) = \{x\}
\end{array}
$$

Figure 27: Definition of free names

$$
\begin{array}{l}
\mathsf{fpn}(\mathbf{0}) = \mathsf{fpn}(X) = \emptyset\\
\mathsf{fpn}(v(\mathbf{x}).P) = \mathsf{fpn}(\overline{v}(\mathbf{x}).P) = \mathsf{fpn}(P) \setminus \{\mathbf{x}\}\\
\mathsf{fpn}(\kappa!(l).P) = \mathsf{fpn}(\kappa!(\tilde{v}).P) = \mathsf{fpn}(\kappa?(\tilde{x}).P) = \mathsf{fpn}(P) \cup \{\kappa\}\\
\mathsf{fpn}(\Sigma_{i=1}^{n}\kappa?(l_i).P_i = \bigcup_{i=1}^{n} \mathsf{fpn}(P_i) \cup \{\kappa\}\\
\mathsf{fpn}(\kappa!\langle\!\langle \kappa' \rangle\!\rangle.P) = \mathsf{fpn}(P) \cup \{\kappa, \kappa'\}\\
\mathsf{fpn}(\kappa?(\!(\mathbf{x})\!).P) = (\mathsf{fpn}(P) \cup \{\kappa\}) \setminus \{\mathbf{x}\}\\
\mathsf{fpn}(\texttt{if } v = w \texttt{ then } P \texttt{ else } Q) = \mathsf{fpn}(P|Q) = \mathsf{fpn}(P) \cup \mathsf{fpn}(Q)\\
\mathsf{fpn}((\nu a)P) = \mathsf{fpn}(\texttt{rec } X.P) = \mathsf{fpn}(P)\\
\mathsf{fpn}((\nu r)P) = \mathsf{fpn}(P) \setminus \{r^+, r^-\}
\end{array}
$$

Figure 28: Definition of free polarized session names

passing communication within a session κ and rule (Label) allows choosing a label offered from the partner within a session κ. Other rules are standard: (If1-If2) allows testing if two values are equal, (Rec) is the standard rule for recursion, (Scop) allows reductions inside restrictions, (Par) allows for parallel composition and (Str) applies structural congruence to group redexes.

Example 5.1. In this example we define a proxy server that multiplexes values from a client by invoking a new instance of a service for each received value. Process $Proxy$ has two recursions one with process variable Y that allows for replication and one with process variable X that allows to repeatedly receive requests from a client.

$$Proxy = \texttt{rec } Y.(a(\mathbf{x}).\texttt{rec } X.\mathbf{x}?(x).\overline{b}(\mathbf{y}).\mathbf{y}!(x).X|Y)$$

The real service processes only one request per time

$$Service = \texttt{rec } X.b(\mathbf{x}).\mathbf{x}?(x)|X$$

$$
\begin{array}{lcl}
\mathsf{fpv}(\mathbf{0}) & = & \emptyset \\
\mathsf{fpv}(v(\mathbf{x}).P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\overline{v}(\mathbf{x}).P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\kappa!(\tilde{v}).P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\kappa?(\tilde{x}).P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\kappa!(l).P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\Sigma_{i=1}^{n}\kappa!(l_i).P_i) & = & \bigcup_{i=1}^{n}\mathsf{fpv}(P_i)
\end{array}
\quad
\begin{array}{lcl}
\mathsf{fpv}(\kappa!\langle\!\langle\kappa'\rangle\!\rangle.P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\kappa?(\!(\mathbf{x})\!).P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\begin{array}{c}\texttt{if then} \\ P\ \texttt{else}\ Q\end{array}) & = & \mathsf{fpv}(P)\cup\mathsf{fpv}(Q) \\
\mathsf{fpv}(P|Q) & = & \mathsf{fpv}(P)\cup\mathsf{fpv}(Q) \\
\mathsf{fpv}((\nu m)P) & = & \mathsf{fpv}(P) \\
\mathsf{fpv}(\texttt{rec}\ X.P) & = & \mathsf{fpv}(P)\setminus\{X\} \\
\mathsf{fpv}(X) & = & \{X\}
\end{array}
$$

Figure 29: Definition of free process names

$$
\begin{array}{l}
P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R) \\
(\nu m)P|Q \equiv (\nu m)(P|Q) \text{ if } m \notin \mathsf{fn}(Q) \\
(\nu m)\mathbf{0} \equiv \mathbf{0} \\
(\nu m)(\nu n)P \equiv (\nu n)(\nu m)Q
\end{array}
$$

Figure 30: Structural congruence

and finally the client which invokes the proxy and continuously sends requests by means of recursion.

$$Client = \overline{a}(\mathbf{x}).\texttt{rec}\ X.\mathbf{x}!(5).\mathbf{x}!(6).X$$

Example 5.2. In this example we show how session delegation works. We show a delegation of an ended session since it is notoriously (6) a bug of the type system of the original proposal (37). We remind that the original proposal describes a different calculus from HVK-X since it does not use polarities annotation (see Section 6.2.2 for a brief description of the calculus). Session calculi with polarity annotations does not present any problems in the treatment of ended session delegation, but as we shall see the treatment introduces some issues while trying to obtain a full syntax-directed type system.

$$Nulldel = a(\mathbf{x}).b(\mathbf{y}).\mathbf{y}!\langle\!\langle\mathbf{x}\rangle\!\rangle.\mathbf{0}|\overline{a}(\mathbf{x})|\overline{b}(\mathbf{y}).\mathbf{y}?(\!(\mathbf{x})\!).\mathbf{0}$$

The protocol of a is an ended session but the first subprocess delegates via b to the third subprocess the capability to take place in a conversation

with a. We show the process reductions:

$$
\begin{array}{l}
a(\mathbf{x}).b(\mathbf{y}).\mathbf{y}!\langle\!\langle \mathbf{x} \rangle\!\rangle.\mathbf{0}|\overline{a}(\mathbf{x})|\bar{b}(\mathbf{y}).\mathbf{y}?(\!(\mathbf{x})\!).\mathbf{0} \rightarrow \\
b(\mathbf{y}).\mathbf{y}!\langle\!\langle r^+ \rangle\!\rangle.\mathbf{0}|\mathbf{0}|\bar{b}(\mathbf{y}).\mathbf{y}?(\!(\mathbf{x})\!).\mathbf{0} \rightarrow \\
r_1^+!\langle\!\langle r^+ \rangle\!\rangle.\mathbf{0}|\mathbf{0}|r_1^-?(\!(\mathbf{x})\!).\mathbf{0} \rightarrow \\
\mathbf{0}|\mathbf{0}|\mathbf{0}
\end{array}
$$

5.3 Type system

5.3.1 A type system for HVK-X

The set of types we consider is the same introduced in Figure 1. We start introducing the type discipline inspired to (76). For processes, the type judgements we consider are of the form $\Gamma; \Theta \vdash P \triangleright \Delta$. We omit those for values which are the same as in Figure 16 but without rule (SES). Typing environments Γ, Θ and Δ are finite partial mappings. Γ is the standard typing environment, it maps variables and service names to sort type S, Θ is the process typing environment and maps process variables X to a list of pairs of the form κT where T is not end. An assumption of the form $X : \kappa T$ means that we assume X to have an open session κ with type T. Δ is the linear environment and maps polarized sessions and session variables to session types. We define the comma operator $\Gamma, x : S$ and $\Gamma, a : S$ and $\Theta, X : \kappa_1 T_1, \ldots, \kappa_n T_n$ as expected. Also we introduce some abbreviations for ease of notation: we write $\Theta, X : \tilde{\kappa}\tilde{T}$ for $\Theta, X : \kappa_1 T_1, \ldots, \kappa_n T_n$ if $\tilde{k} = \kappa_1, \ldots, \kappa_n$ and $\tilde{T} = T_1, \ldots, T_n$ and we write $\tilde{\kappa} : \tilde{T}$ for the linear environment $\Delta = \emptyset, \kappa_1 : T_1, \ldots, \kappa_n : T_n$ if $\tilde{\kappa} = k_1, \ldots, k_n$ and $\tilde{T} = T_1, \ldots, T_n$. We write $\{\tilde{k}\}$ for the set $\{\kappa_1, \ldots, \kappa_n\}$ if $\tilde{\kappa} = \kappa_1, \ldots, \kappa_n$ and similar for $\{\tilde{T}\}$. Consequently the domain of a linear environment Δ is the set $\mathsf{dom}(\Delta) = \{\tilde{\kappa}\}$ if $\Delta = \tilde{k} : \tilde{T}$ for some $\tilde{\kappa}, \tilde{T}$ With the help of these definitions we can define syntactically each environment.

Definition 5.3 (Well-formed typing environments)**.** Γ, Θ, Δ are well formed if they are generated from the following grammar:

$$
\begin{array}{l}
\Gamma ::= \emptyset \mid \Gamma, a : [T] \mid \Gamma, x : S \\
\Theta ::= \emptyset \mid \Theta, X : \tilde{\kappa}\tilde{T} \quad \text{and } \mathtt{end} \notin \{\tilde{T}\} \\
\Delta ::= \emptyset \mid \Delta, \kappa : T
\end{array}
$$

The reason we disallowed ended sessions as assumptions in Θ is technical and it is related to the proof of the Strengthening Lemma. More-

over, notice the differences from these environments and those defined in the previous chapter. Environments reflect the differences between languages, for example in CST we stored the type of a session in Γ since communications are somewhat grouped by the operator $r^p \triangleright P$ so we do not need a specialized environment to collect session types but we used the set L to keep linearity. Instead here since communications can be interleaved Δ is used to store assumptions about the opened sessions.

Some comments for the typing rules reported in Figure 31 are in order. Rule (SNIL) types the process $\mathbf{0}$ with the empty environment, rule (SACC) checks that P uses $\mathbf{x}$ in accordance with the type stored in Γ for the service, (SREQ) is similar but it checks the duality w.r.t. the type in Γ. Rule (SWEAK) uses the operation $\Delta_1 \leq \Delta_2$ defined as:

$$\Delta_1 \leq \Delta_2 = \begin{cases} \Delta_1(\kappa) \leq \Delta_2(\kappa) & \text{if } \kappa \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) \\ \Delta_1(\kappa) = \mathsf{end} & \text{if } \kappa \in \mathsf{dom}(\Delta_1) \setminus \mathsf{dom}(\Delta_2) \end{cases}$$

The operation allows both to replace the type of an existing session with one of its own subtype and to add new ended sessions. This allows axioms ((SNIL) and (SPVAR)) with only the strictly necessary sessions. When we want to limit the arbitrariness of the operation we write ${\Delta_1}_{\downarrow \{\tilde{\kappa}\}} \leq \Delta_2$ if $\Delta_1 \leq \Delta_2$ and $\mathsf{dom}(\Delta_1) = \{\tilde{\kappa}\}$.

Rules (SIN) and (SOUT) add an input action and an output action respectively to the type of κ, rule (SBRANCH) imposes the linear environment of each P_i to be equal but for κ which is used to offer an external choice. The final type in the conclusion has a subset of the total offered labels mainly because a process can offer equal labels. Equal labels are not allowed instead at the type system level, by the well-formedness condition. However we want to type such process as long as equal labels have a common type (see Example 4.1).

(SNIL)
$$\Gamma;\Theta \vdash \mathbf{0} \triangleright \emptyset$$

(SACC)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta, \mathbf{x} : T \quad \Gamma \vdash v : [T]}{\Gamma;\Theta \vdash v(\mathbf{x}).P \triangleright \Delta}$$

(SREQ)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta, \mathbf{x} : \overline{T} \quad \Gamma \vdash v : [T]}{\Gamma;\Theta \vdash \overline{v}(\mathbf{x}).P \triangleright \Delta}$$

(SWEAK)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta \quad \Delta' \leq \Delta}{\Gamma;\Theta \vdash P \triangleright \Delta'}$$

(SIN)
$$\frac{\Gamma, \tilde{x} : \tilde{S};\Theta \vdash P \triangleright \Delta, \kappa : T}{\Gamma;\Theta \vdash \kappa?(\tilde{x}).P \triangleright \Delta, \kappa :?(\tilde{S}).T}$$

(SOUT)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta, \kappa : T \quad \Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma;\Theta \vdash \kappa!(\tilde{v}).P \triangleright \Delta, \kappa :!(\tilde{S}).T}$$

(SBRANCH)
$$\frac{\emptyset \subset J \subseteq I = \{1,\ldots,n\} \quad \forall i \in I\ (\Gamma;\Theta \vdash P_i \triangleright \Delta, \kappa : T_i)}{\Gamma;\Theta \vdash \Sigma_{i=1}^{n} \kappa?(l_i).P_i \triangleright \Delta, \kappa : \&\{l_j : T_j\}_{j\in J}}$$

(SCHOICE)
$$\frac{l = l_i \in \{l_1,\ldots,l_n\} \quad \Gamma;\Theta \vdash P \triangleright \Delta, \kappa : T_i}{\Gamma;\Theta \vdash \kappa!(l).P \triangleright \Delta, \kappa : \oplus\{l_1 : T_1;\ldots;l_n : T_n\}}$$

(SCATCH)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta, \kappa : T, \mathbf{x} : U}{\Gamma;\Theta \vdash \kappa?(\!(\mathbf{x})\!).P \triangleright \Delta, \kappa :?(U).T}$$

(STHROW)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta, \kappa : T}{\Gamma;\Theta \vdash \kappa!\langle\!\langle \kappa' \rangle\!\rangle.P \triangleright \Delta, \kappa :!(U).T, \kappa' : U}$$

(SIF)
$$\frac{(\Gamma \vdash v_i : S \quad \Gamma;\Theta \vdash P_i \triangleright \Delta)\ i \in \{1,2\}}{\Gamma;\Theta \vdash \texttt{if } v_1 = v_2 \texttt{ then } P_1 \texttt{ else } P_2 \triangleright \Delta}$$

(SPAR)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta_1 \quad \Gamma;\Theta \vdash Q \triangleright \Delta_2}{\Gamma;\Theta \vdash P|Q \triangleright \Delta_1, \Delta_2}$$

(SNEWR)
$$\frac{\Gamma;\Theta \vdash P \triangleright \Delta, r^+ : T, r^- : \overline{T}}{\Gamma;\Theta \vdash (\nu r)P \triangleright \Delta}$$

(SNEW)
$$\frac{\Gamma, a : S;\Theta \vdash P \triangleright \Delta}{\Gamma;\Theta \vdash (\nu a)P \triangleright \Delta}$$

(SREC)
$$\frac{\Gamma;\Theta, X : \tilde{\kappa}\tilde{T} \vdash P \triangleright \tilde{\kappa} : \tilde{T} \quad \{\tilde{\kappa}\} = \mathsf{fpn}(P)}{\Gamma;\Theta \vdash \texttt{rec } X.P \triangleright \tilde{\kappa} : \tilde{T}}$$

(SPVAR)
$$\Gamma;\Theta, X : \tilde{\kappa}\tilde{T} \vdash X \triangleright \tilde{\kappa} : \tilde{T}$$

Figure 31: Typing rules

Rule (SCHOICE) adds arbitrary internal choices together with the correct one, notice that we would simulate this behavior with the help of rule (SWEAK) but we duplicate it to simplify proofs since we do not need to account for the arbitrary presence of (SWEAK) in each type derivation ending with rules (SCHOICE) or (SBRANCH). Two rules allows for session delegation (SCATCH) and (STHROW). Rule (SCATCH) allows using $\mathbf{x}$ only after it is catched, in the continuation P, vice versa rule (STHROW) allows communications over κ' only before it is sent. The rule for parallel composition uses the operation Δ_1, Δ_2 defined as the point-wise extension of $\Delta, \kappa : T$ to check the disjointness of the two linear environments. Rule (SNEWR) restricts r only if its two polarized versions in the linear environment are dual to each other. Finally the rule for recursion (SREC) requires that the body of recursion P is typed only with the set of free polarized names. It is also clear that each session in the set of free polarized names has a type different from $\mathtt{end}$ (remember the condition imposed by the well-formedness) since a throw instruction of the form $\kappa!\langle\langle\kappa'\rangle\rangle.P$ cannot be inside the body of a recursion $\mathtt{rec}\ X.Q$ if $\kappa' \in \mathsf{fpn}(Q)$.

This choices is also consistent with (76) which allows the body of a definition to be typed only with those sessions used as parameters of a definition. Here parameters are explicit, i.e. they are all the free polarized names in P. Besides this fact the rule is the standard rule for recursion. Notice that this time the process $\mathtt{rec}\ X.X$ is typed only with $\Gamma; \Theta \vdash \mathtt{rec}\ X.X \triangleright \emptyset$.

Example 5.4. Letting $T_\alpha = \mu\alpha.?(int).\alpha$ and $\Theta = X : \mathbf{x}\, T_\alpha, Y : \emptyset$ and $\Gamma = a : [T_\alpha], b : [?(int)]$, the process $Proxy$ from Example 5.1 can be typed as:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x; int; \Theta \vdash X \triangleright \mathbf{x} : T_\alpha}{\Gamma, x : int; \Theta \vdash X \triangleright \mathbf{x} : T_\alpha, \mathbf{y} : \mathtt{end}}\ \text{(SWEAK)}}{\Gamma, x : int; \Theta \vdash \mathbf{y}!(x).X \triangleright \mathbf{x} : T_\alpha, \mathbf{y} :!(int)}\ \text{(SOUT)}}{\Gamma, x : int; \Theta \vdash \overline{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \mathbf{x} : T_\alpha}\ \text{(SREQ)}}{\Gamma; \Theta \vdash \mathbf{x}?(x).\overline{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \mathbf{x} :?(int).T_\alpha}\ \text{(SIN)}}{\Gamma; \Theta \vdash \mathbf{x}?(x).\overline{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \mathbf{x} : T_\alpha}\ \text{(SWEAK)}}{\Gamma; \Theta \vdash \mathtt{rec}\ X.\mathbf{x}?(x).\overline{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \mathbf{x} : T_\alpha}\ \text{(SREC)}}{\Gamma; Y : \emptyset \vdash a(\mathbf{x}).\mathtt{rec}\ X.\mathbf{x}?(x).\overline{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \emptyset}\ \text{(SACC)} \quad \Gamma; Y : \emptyset \vdash Y \triangleright \emptyset}{\Gamma; Y : \emptyset \vdash a(\mathbf{x}).\mathtt{rec}\ X.\mathbf{x}?(x).\overline{b}(\mathbf{y}).\mathbf{y}!(x).X | Y \triangleright \emptyset}\ \text{(SPAR)}}{\Gamma; \emptyset \vdash Proxy \triangleright \emptyset}\ \text{(SREC)}$$

Example 5.5. Letting $\Gamma = a : [\mathsf{end}], b : [!(\mathsf{end})]$ each of the three sub processes in Example 5.2 can be typed as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; \emptyset \vdash \mathbf{y}!\langle\!\langle \mathbf{x} \rangle\!\rangle.\mathbf{0} \triangleright \mathbf{y} : !(\mathtt{end}), \mathbf{x} : \mathtt{end}}}{\Gamma; \emptyset \vdash b(\mathbf{y}).\mathbf{y}!\langle\!\langle \mathbf{x} \rangle\!\rangle.\mathbf{0} \triangleright \mathbf{x} : \mathtt{end}} \text{(SACC)}}{\Gamma; \emptyset \vdash a(\mathbf{x}).b(\mathbf{y}).\mathbf{y}!\langle\!\langle \mathbf{x} \rangle\!\rangle.\mathbf{0} \triangleright \emptyset} \text{(SACC)} \qquad \dfrac{\dfrac{\Gamma; \emptyset \vdash \mathbf{0} \triangleright \emptyset}{\Gamma; \emptyset \vdash \overline{a}(\mathbf{x}) \triangleright \mathbf{x} : \mathtt{end}} \text{(SNIL)}}{\Gamma; \emptyset \vdash \overline{a}(\mathbf{x}) \triangleright \emptyset} \text{(SREQ)}$$

$$\dfrac{\dfrac{\vdots}{\Gamma; \emptyset \vdash \mathbf{y}?(\!(\mathbf{x})\!).\mathbf{0} \triangleright \mathbf{y} : ?(\mathtt{end})}}{\Gamma; \emptyset \vdash \overline{b}(\mathbf{y}).\mathbf{y}?(\!(\mathbf{x})\!).\mathbf{0} \triangleright \emptyset} \text{(SREQ)}$$

5.3.2 Subject reduction and safety

The aim of this section is to prove that typing is preserved along reductions steps, due to the above discussed modifications the typing preservation is not obvious from the subject reduction theorem proved in (76). We prove the Weakening Lemma which allows adding assumptions into each environment but for the linear one, which can use the rule (SWEAK) to add assumptions for ended sessions. The Strengthening Lemma can be used to remove unused assumptions, this time the statement concerning the linear environment allows to remove assumptions relative to ended sessions if the name does not belong to the set of free polarized names.

Lemma 5.6 (Weakening). *If $\Gamma; \Theta \vdash P \triangleright \Delta$ and $v \notin \mathsf{fn}(P)$ with v either a service name or a variable then $\Gamma, v : S; \Theta \vdash P \triangleright \Delta$ for any S. If $\Gamma; \Theta \vdash P \triangleright \Delta$ and $X \notin \mathsf{fpv}(P)$ then $\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash P \triangleright \Delta$ for any $\tilde{\kappa}\tilde{T}$.*

Proof. The proof of both statements is by induction on the typing derivation of $\Gamma; \Theta \vdash P \triangleright \Delta$ with case analysis on the last applied rule. □

Lemma 5.7 (Strengthening). *If $\Gamma, v : S; \Theta \vdash P \triangleright \Delta$ and $v \notin \mathsf{fn}(P)$ with v either a service name or a variable then $\Gamma; \Theta \vdash P \triangleright \Delta$. If $\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash P \triangleright \Delta$ and $X \notin \mathsf{fpv}(P)$ then $\Gamma; \Theta \vdash P \triangleright \Delta$. If $\Theta; \Gamma \vdash P \triangleright \Delta, \kappa : \mathtt{end}$ and $\kappa \notin \mathsf{fpn}(P)$ then $\Theta; \Gamma \vdash P \triangleright \Delta$.*

Proof. The proof of each statement is by induction on the typing derivation of $\Gamma, v : S; \Theta \vdash P \triangleright \Delta$, $\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash P \triangleright \Delta$ and $\Theta; \Gamma \vdash P \triangleright \Delta, \kappa : \mathtt{end}$ respectively. We show the base case relative to a process variable, the others are straightforward. We have $\Gamma; \Theta', X : \tilde{\kappa}\tilde{T} \vdash X \triangleright \tilde{\kappa} : \tilde{T}$ and since each T is different from `end` by the well-formedness condition the premise of the theorem is false so the implication holds. □

This lemma relates the various definitions of free names to each environment.

Lemma 5.8 (Free names and environments)**.** *Let* $\Gamma; \Theta \vdash P \triangleright \Delta$ *then*

- *if* $a \notin \mathsf{dom}(\Gamma)$ *then* $a \notin \mathsf{fn}(P)$*.*
- *if* $X \notin \mathsf{dom}(\Theta)$ *then* $X \notin \mathsf{fpv}(P)$*.*
- *if* $\kappa \notin \mathsf{dom}(\Delta)$ *then* $\kappa \notin \mathsf{fpn}(P)$*.*

Proof. The proof is by straightforward induction on the typing derivation of $\Gamma; \Theta \vdash P \triangleright \Delta$ with case analysis on the last applied rule. □

Previous lemma is stated in the form $\neg p_1$ implies $\neg p_2$ but sometimes we will use it in the form p_2 implies p_1 which is logically equivalent.

Substitution Lemma allows to collapse two assumptions but for the linear environment which allows substituting a session variable with a polarized session name.

Lemma 5.9 (Substitution)**.** *If* $\Gamma, x : S; \Theta \vdash P \triangleright \Delta$ *and* $\Gamma \vdash v : S$ *then* $\Gamma; \Theta \vdash P[{}^{v}/_{x}] \triangleright \Delta$*. If* $\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash P \triangleright \Delta$ *and* $\Gamma; \Theta \vdash Q \triangleright \tilde{\kappa} : \tilde{T}$ *then* $\Gamma; \Theta \vdash P[{}^{Q}/_{X}] \triangleright \Delta$*. If* $\Gamma; \Theta \vdash P \triangleright \Delta, \mathbf{x} : T$ *and* $r^p \notin \mathsf{dom}(\Delta)$ *then* $\Gamma; \Theta \vdash P[{}^{r^p}/_{\mathbf{x}}] \triangleright \Delta, r^p : T$*.*

Proof. All statements are proved by induction on the typing derivation with case analysis on the last applied rule. Regarding the first statement we prove the cases when last applied rules are an output action and a service invocation. Service definition is similar and other cases are simpler and follow directly by inductive hypothesis. Without loss of generality we prove the theorem for the case when the tuple length in input, output actions is one. We have $\dfrac{\Gamma, x : S, \dot{v} : S; \Theta \vdash P' \triangleright \Delta, \kappa : T \quad \Gamma, \dot{v} : S, x : S \vdash w : S}{\Gamma, x : S, \dot{v} : S; \Theta \vdash \kappa!(w).P' \triangleright \Delta, \kappa :!(S).T}\,(\textsc{SOut})$ and then two different possibilities depending if $\dot{x} = w$ or not. The latter case follows directly by inductive hypothesis. In the former case instead we know by inductive hypothesis that $\Gamma; \Theta \vdash P'[{}^{\dot{v}}/_{w}] \triangleright \Delta$ and (SOut) gives $\Gamma; \Theta \vdash \kappa!(w).P'[{}^{\dot{v}}/_{w}] \triangleright \Delta'$ i.e. $\Gamma; \Theta \vdash (\kappa!(\dot{v}).P')[{}^{\dot{v}}/_{w}] \triangleright \Delta'$ the desiderated result. Service invocation is similar we have $\dfrac{\Gamma, x : [T], \dot{v} : S; \Theta \vdash P' \triangleright \Delta, \mathbf{y} : \overline{T} \quad \Gamma, x : [T], \dot{v} : S \vdash v : [T]}{\Gamma, x : [T], \dot{v} : S; \Theta \vdash \overline{v}(\mathbf{y}).P' \triangleright \Delta}\,(\textsc{SReq})$ and (provided that $\dot{x} = v$) thanks to the inductive hypothesis and by applying (SReq) we can conclude $\Gamma; \Theta \vdash (\overline{\dot{v}(\mathbf{y})}.P')[{}^{\dot{v}}/_{v}] \triangleright \Delta'$.

The second statement relative to the Θ environment follows directly by straightforward induction on the typing derivation, we only sketch the base case where $\dot{P} = \dot{X}$. In this case we have $\frac{\text{(SPVAR)}}{\Gamma; \Theta, \dot{X} : \tilde{\kappa}\tilde{T} \vdash \dot{X} \triangleright \tilde{\kappa} : \tilde{T}}$ and $\Gamma; \Theta \vdash \dot{X}[^{\dot{Q}}/_{\dot{X}}] \triangleright \tilde{\kappa} : \tilde{T}$ follows by hypothesis.

Finally about the third statement for the linear environment Δ we sketch the case when the last applied rules are

- (STHROW), and $\dot{P} = \kappa!\langle\langle\kappa'\rangle\rangle.P$. We have three cases (notice that $\kappa \neq \kappa'$ otherwise the rule is not applicable) depending of if $\dot{\mathrm{x}} = \kappa$, if $\dot{\mathrm{x}} = \kappa'$ or if $\kappa \neq \dot{\mathrm{x}} \neq \kappa'$. The latter case follows directly by induction. If $\dot{\mathrm{x}} = \kappa$ we have $\dfrac{\Gamma; \Theta \vdash P \triangleright \Delta, \kappa : T}{\Gamma; \Theta \vdash \kappa!\langle\langle\kappa'\rangle\rangle.P \triangleright \Delta, \kappa :!(U).T, \kappa' : U}$ (STHROW) and by induction hypothesis $\Gamma; \Theta \vdash P[^{r^p}/_{\kappa}] \triangleright \Delta, r^p : T$ and applying (STHROW) we conclude $\Gamma; \Theta \vdash (\kappa!\langle\langle\kappa'\rangle\rangle.P)[^{r^p}/_{\kappa}] \triangleright \Delta, r^p :!(U).T, \kappa' : U$. If $\mathbf{x} = \kappa'$ applying Lemma 5.8 we know that $\kappa' \notin \mathsf{fpn}(P)$ then $P[^{r^p}/_{\kappa'}] = P$ and the result follows by applying rule (STHROW) that gives $\Theta; \Gamma \vdash \kappa!\langle\langle r^p\rangle\rangle.P \triangleright \Delta, \kappa :!(U).T, r^p : U$.

- (SDEF) and $\dot{P} = v(\mathbf{x}).P$. By hypothesis $\Gamma; \Theta \vdash \dot{P} \triangleright \Delta$ for some Δ and $\kappa \notin \mathsf{dom}(\Delta)$ thus if $\mathbf{x} = \dot{\mathrm{x}}$ the premise of the theorem is false so the theorem is true. When $\mathbf{x} = \dot{\mathrm{x}}$ the hypothesis follows directly by induction.

□

Corollary 5.10 (Substitution lemma for tuples). *If* $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P \triangleright \Delta$ *and* $\Gamma \vdash \tilde{v} : \tilde{S}$ *then* $\Gamma; \Theta \vdash P[^{\tilde{v}}/_{\tilde{x}}] \triangleright \Delta$.

Proof. The proof is by induction on the tuple length, the base case is the Substitution Lemma the inductive case is proved thanks to the Substitution Lemma as in the previous chapter. □

The subject congruence proves the typability is preserved by the structural congruence.

Lemma 5.11 (Subject Congruence). *If* $\Gamma; \Theta \vdash P \triangleright \Delta$ *and* $P \equiv Q$ *then* $\Gamma; \Theta \vdash Q \triangleright \Delta$.

Proof. As before we show the typing of each congruence rule holds in both directions, which it suffices to conclude.
Case $\mathbf{P|0 \equiv P}$:

$$\dfrac{\Gamma;\Theta \vdash P \triangleright \Delta \qquad \Gamma;\Theta \vdash \mathbf{0} \triangleright \emptyset}{\Gamma;\Theta \vdash P|\mathbf{0} \triangleright \Delta}\ (\textsc{SPar}) \qquad \Gamma;\Theta \vdash P \triangleright \Delta$$

Case $\mathbf{P}|\mathbf{Q} \equiv \mathbf{Q}|\mathbf{P}$:

$$\dfrac{\Gamma;\Theta \vdash P \triangleright \Delta_1 \qquad \Gamma;\Theta \vdash Q \triangleright \Delta_2}{\Gamma;\Theta \vdash P|Q \triangleright \Delta_1,\Delta_2}\ (\textsc{SPar})$$

$$\dfrac{\Gamma;\Theta \vdash Q \triangleright \Delta_2 \qquad \Gamma;\Theta \vdash P \triangleright \Delta_1}{\Gamma;\Theta \vdash Q|P \triangleright \Delta_2,\Delta_1}\ (\textsc{SPar})$$

We can conclude since $\dot{\Delta} = \Delta_1, \Delta_2 = \Delta_2, \Delta_1$
Case $(\mathbf{P}|\mathbf{Q})|\mathbf{R} \equiv \mathbf{P}|(\mathbf{Q}|\mathbf{R})$: This case is similar to the previous one.
Case $(\nu\mathbf{m})\mathbf{P}|\mathbf{Q} \equiv (\nu\mathbf{m})(\mathbf{P}|\mathbf{Q})$ if $\mathbf{m} \notin \mathsf{fn}(\mathbf{Q})$: We have two cases depending of whether m is a session name or a service name. If $m = r$ we have:

$$\dfrac{\dfrac{\Gamma;\Theta \vdash P \triangleright \Delta_1, r^+ : T, r^- : \overline{T}}{\Gamma;\Theta \vdash (\nu r)P \triangleright \Delta_1}\ (\textsc{SNewS}) \qquad \Gamma;\Theta \vdash Q \triangleright \Delta_2}{\Gamma;\Theta \vdash (\nu r)P|Q \triangleright \Delta}\ (\textsc{SPar})$$

$$\dfrac{\dfrac{\Gamma;\Theta \vdash P \triangleright \Delta_1, r^+ : T, r^- : \overline{T} \qquad \Gamma;\Theta \vdash Q \triangleright \Delta_2}{\Gamma;\Theta \vdash P|Q \triangleright \Delta, r^+ : T, r^- : \overline{T}}\ (\textsc{SPar})}{\Gamma;\Theta \vdash (\nu r)(P|Q) \triangleright \Delta}\ (\textsc{SNewS})$$

From the top tree to the bottom tree we use the Strengthening Lemma to remove both r^+ and r^- from $\mathsf{dom}(\Delta_2)$ since $r \notin \mathsf{fn}(Q)$ implies $\{r^+, r^-\} \cap \mathsf{fpn}(Q) = \emptyset$. Instead, if $m = a$ we have:

$$\dfrac{\dfrac{\Gamma, a : [T];\Theta \vdash P \triangleright \Delta_1}{\Gamma;\Theta \vdash (\nu a)P \triangleright \Delta_1}\ (\textsc{SNew}) \qquad \Gamma;\Theta \vdash Q \triangleright \Delta_2}{\Gamma;\Theta \vdash (\nu a)P|Q \triangleright \Delta}\ (\textsc{SPar})$$

$$\dfrac{\dfrac{\Gamma, a : [T];\Theta \vdash P \triangleright \Delta_1 \qquad \Gamma, a : [T];\Theta \vdash Q \triangleright \Delta_2}{\Gamma, a : [T];\Theta \vdash P|Q \triangleright \Delta}\ (\textsc{SPar})}{\Gamma;\Theta \vdash (\nu a)(P|Q) \triangleright \Delta}\ (\textsc{SNewS})$$

In detail, when the proof is from the first proof tree to the second proof tree we use the Weakening Lemma to type Q otherwise if the proof is in the opposite direction we use the Strengthening Lemma to type Q.
Case $(\nu m)\mathbf{0} \equiv \mathbf{0}$: in case $m = r$ both processes can be typed with the empty session environment otherwise if $m = a$ the result follows applying either the Strengthening Lemma or the Weakening Lemma according to the direction of the proof.
Case $(\nu m)(\nu n)P \equiv (\nu n)(\nu m)Q$: follows directly thanks to the commutativity of the comma operator among environments. □

Eventually we are ready to prove the Subject Reduction Theorem, we say that Δ is balanced if $\Delta(r^p) = T$ and $r^{\overline{p}}$ in $\mathsf{dom}(\Delta)$ then $\Delta(r^{\overline{p}}) = \overline{T}$. Two balanced linear environments Δ_1 and Δ_2 are *balanced equal* if $\mathsf{dom}(\Delta_1) = \mathsf{dom}(\Delta_2)$ and for all r^p s.t. $r^p \in \mathsf{dom}(\Delta_1)$ and $r^{\overline{p}} \notin \mathsf{dom}(\Delta_1)$ then $\Delta_1(r^p) = \Delta_2(r^p)$.

Theorem 5.12 (Subject Reduction). *If* $\Gamma; \Theta \vdash P \triangleright \Delta$ *with* Δ *balanced and* $P \to Q$ *then there exists a balanced* Δ' *s.t.* $\Gamma; \Theta \vdash Q \triangleright \Delta'$ *with* Δ *and* Δ' *balanced equal.*

Proof. The proof is by induction on the derivation of $P \to Q$ with case analysis on the last applied rule.

Base Case:(LINK) $(a(\mathbf{x}).P)|(\overline{a}(\mathbf{y}).Q) \quad \to \quad (\nu r)(P[^{r^+}/_{\mathbf{x}}]|Q[^{r^-}/_{\mathbf{y}}]) \qquad r \notin \mathsf{fn}(P|Q)$. The typing of $\dot{P}$ is derived from:

$$\dfrac{(\textsc{SAcc})\ \dfrac{\Gamma, a:[T];\Theta \vdash P \triangleright \Delta_1, \mathbf{x}:T}{\Gamma, a:[T];\Theta \vdash a(\mathbf{x}).P \triangleright \Delta_1} \qquad \dfrac{\Gamma, a:[T];\Theta \vdash Q \triangleright \Delta_2, \mathbf{y}:\overline{T}}{\Gamma, a:[T];\Theta \vdash \overline{a}(\mathbf{y}).Q \triangleright \Delta_2}\ (\textsc{SInv})}{\Gamma, a:[T];\Theta \vdash \dot{P} \triangleright \dot{\Delta}}\ (\textsc{SPar})$$

$$\dfrac{\dfrac{\overbrace{\Gamma, a:[T];\Theta \vdash P[^{r^+}/_{\mathbf{x}}] \triangleright \Delta_1, r^+:T}^{\text{Sub. Lemma}} \qquad \overbrace{\Gamma, a:[T];\Theta \vdash Q[^{r^-}/_{\mathbf{y}}] \triangleright \Delta_2, r^-:\overline{T}}^{\text{Sub. Lemma}}}{\Gamma, a:[T];\Theta \vdash P[^{r^+}/_{\mathbf{x}}]|Q[^{r^-}/_{\mathbf{y}}] \triangleright \Delta_1, \Delta_2, r^+:T, r^-:\overline{T}}\ (\textsc{SPar})}{\Gamma, a:[T];\Theta \vdash \dot{Q} \triangleright \dot{\Delta}'}\ (\textsc{SNewR})$$

We notice that $r \notin \mathsf{fn}(P|Q)$ implies $\{r^+, r^-\} \cap \mathsf{fpn}(P) = \emptyset$ and $\{r^+, r^-\} \cap \mathsf{fpn}(Q) = \emptyset$ and we can use the Strengthen Lemma to remove r^+ and r^- from both $\mathsf{dom}(\Delta_1)$ and $\mathsf{dom}(\Delta_2)$ to fit the premise of the Substitution Lemma. Also $\dot{\Delta} = \dot{\Delta}'$.

Base Case:(COM)$(r^p!(\tilde{v}).P)|(r^{\overline{p}}?(\tilde{x}).Q) \to (P|Q[^{\tilde{v}}/_{\tilde{x}}]) \quad |\tilde{v}| = |\tilde{x}|$. The typing of $\dot{P}$ is derived from:

$$\dfrac{(\textsc{SOut})\ \dfrac{\Gamma;\Theta \vdash P \triangleright \Delta_1, r^p:T}{\Gamma;\Theta \vdash r^p!(\tilde{v}).P \triangleright \Delta_1, r^p:!(\tilde{S}).T} \qquad \dfrac{\Gamma, \tilde{x}:\tilde{S};\Theta \vdash Q \triangleright \Delta_2, r^{\overline{p}}:\overline{T}}{\Gamma;\Theta \vdash r^{\overline{p}}?(\tilde{x}).Q \triangleright \Delta_2, r^{\overline{p}}:?(\tilde{S}).\overline{T}}\ (\textsc{SIn})}{\Gamma;\Theta \vdash \dot{P} \triangleright \dot{\Delta}}\ (\textsc{SPar})$$

$$\dfrac{\Gamma;\Theta \vdash P \triangleright \Delta_1, r^p:T \qquad \overbrace{\Gamma;\Theta \vdash Q[^{\tilde{v}}/_{\tilde{x}}] \triangleright \Delta_2, r^{\overline{p}}:\overline{T}}^{\text{Sub. Lemma}}}{\Gamma;\Theta \vdash \dot{Q} \triangleright \dot{\Delta}'}\ (\textsc{SPar})$$

To conclude we have $\dot{\Delta}$ and $\dot{\Delta}'$ balanced equal.
Base Case:(Label) similar to the previous one.
Base Case:(Pass) $(r^p!\langle\langle r'^q\rangle\rangle.P)|(r^{\overline{p}}?((\mathbf{x})).Q) \rightarrow P|Q[^{r'^q}/_{\mathbf{x}}]$. The typing of $\dot{P}$ is derived from:

$$\dfrac{\dfrac{\Gamma;\Theta \vdash P \triangleright \Delta_1, r^p : T}{\Gamma;\Theta \vdash r^p!\langle\langle r'^q\rangle\rangle.P \triangleright \Delta_1, r^p :!(U).T, r'^q : U} \quad \dfrac{\Gamma;\Theta \vdash Q \triangleright \Delta_2, r^{\overline{p}} : \overline{T}, \mathbf{x} : U}{\Gamma;\Theta \vdash r^{\overline{p}}?((\mathbf{x})).Q \triangleright \Delta_2, r^{\overline{p}} :?(U).\overline{T}}\ (\text{SCatch})}{\Gamma;\Theta \vdash \dot{Q} \triangleright \dot{\Delta}}\ (\text{SPar})$$

$$\dfrac{\Gamma;\Theta \vdash P \triangleright \Delta_1, r^p : T \qquad \overbrace{\Gamma;\Theta \vdash Q[^{r'^q}/_{\mathbf{x}}] \triangleright \Delta_2, r'^q : U, r^{\overline{p}} : \overline{T}}^{\text{Sub. Lemma}}}{\Gamma;\Theta \vdash \dot{P} \triangleright \dot{\Delta}'}\ (\text{SPar})$$

Some comments are in order. By hypothesis $\Delta_1', r^p \ :!(U).T, r'^q \ : U, \Delta_2', r^{\overline{p}} \ :?(U).\overline{T}$ is defined then it must be the case that $r'^q \notin \mathsf{dom}(\Delta_2', r^{\overline{p}} \ :?(U).\overline{T})$ in order to fit the hypothesis of the Substitution Lemma. Moreover, $\dot{\Delta}$ and $\dot{\Delta}'$ are balanced equal.
Inductive Case:(If1) **and** (If2). By application of the inductive hypothesis either on P or Q together with the fact that typing rule (SIf) requires both branches to have the same type.
Inductive Case:(Rec). We have $\dot{P} = \texttt{rec}\ X.P$. By hypothesis we know that $\Gamma;\Theta \vdash \texttt{rec}\ X.P \triangleright \Delta$ and in order to apply the induction we must prove $P[^{\texttt{rec}\ X.P}/_X]$ is well typed. By hypothesis for some $\tilde{k}$ and $\tilde{T}$, $\Gamma;\Theta, X : \tilde{k}\tilde{T} \vdash P \triangleright \tilde{k} : \tilde{T}$ then $\Gamma;\Theta \vdash P[^{\texttt{rec}\ X.P}/_X] \triangleright \tilde{k}\tilde{T}$ applying the Substitution Lemma. We can conclude since $P[^{\texttt{rec}\ X.P}/_X] \rightarrow \dot{Q}$ and by inductive hypothesis $\dot{Q}$ is well typed.
Inductive Case:(Scop). We have $\dot{P} = (\nu m)P$ and $\dot{Q} = (\nu m)Q$ and by inductive hypothesis $\Gamma;\Theta \vdash P \triangleright \Delta$ with Δ balanced implies $\Gamma;\Theta \vdash Q \triangleright \Delta'$ with Δ' balanced. As usual we have two cases depending of if $m = a$ or if $m = r$. In the former case we can conclude directly by inductive hypothesis and rule (SNew). If $m = r$ by inductive hypothesis r^+ and r^- duals belong to $\mathsf{dom}(\Delta')$ and we can apply rule (SNewR) to conclude.
Inductive Case:(Par) We have $\dot{P} = P|Q$ and $\dot{Q} = P'|Q$. By inductive hypothesis we know that P and P' are typed respectively in some balanced equal linear environments Δ_1 and Δ'. Moreover by hypothesis and rule (SPar) there exists Δ_2 that types Q and Δ_1, Δ_2 is defined and balanced. The thesis follows since $\Delta', \Delta_2 = \dot{\Delta}'$ is defined and balanced with $\dot{\Delta}$ and $\dot{\Delta}'$ balanced equal, by the fact that Δ_1 and Δ' are balanced equal.

Inductive Case:(STR) Follows directly by inductive inductive hypothesis and Lemma 5.11. □

Following (76) to formalise type safety, we need the some auxiliary notions. A r^p-process is a process prefixed by subject r^p (such as $r^p!(\tilde{v}).P$ and $r^p?(\!(\mathbf{x})\!).P$). Next, a r-redex is the parallel composition of a r^p-process with a $r^{\overline{p}}$-process, i.e. either of form $r^p!(\tilde{v}).P|r^{\overline{p}}?(\tilde{x}).Q$ or of the form $r^p!\langle\!\langle\kappa\rangle\!\rangle.P|r^{\overline{p}}?(\!(\mathbf{x})\!).Q$ or of the form $r^p!(l).P|\Sigma_{i=1}^{n} r^{\overline{p}}?(l_i).P_i$. Then P is an error if $P \equiv (\nu\tilde{m})(Q_1|Q_2|Q_3)$ for some Q_3 and for some r, Q_1 is a r^p-process and Q_2 is a $r^{\overline{p}}$-process that do not form a r-redex, or Q_1 and Q_2 for some r are two r^p processes. We then have:

Theorem 5.13 (Safety). *A typable program never reduces to an error.*

Proof. By Subject Reduction it suffices to show that typable programs are not errors. The proof is by reductio ad absurdum, assuming error processes typable. When $P|Q$ is the parallel composition of a r^p-process and a $r^{\overline{p}}$-process that do not form a r-redex, there are several cases to consider. They are all alike; take for example the pair of two inputs. We have $\Gamma;\Theta \vdash P \triangleright \Delta_1, r^p :?(\tilde{S}).T$ and $\Gamma;\Theta \vdash Q \triangleright \Delta_2, r^{\overline{p}} :?(\tilde{S}).U$ for some Δ_1, Δ_2 and $\tilde{S}, T, U$ but the resulting environment is not balanced contradicting the hypothesis of the subject reduction. In the case when P and Q are two r^p processes then $\Gamma;\Theta \vdash P \triangleright \Delta_1, r^p : T$ and $\Gamma;\Theta \vdash Q \triangleright \Delta_1, r^p : U$ for some Δ_1, Δ_2, T, U but the rule (SPAR) would not applicable contradicting the hypothesis of typability.

□

5.4 A syntax directed type system

In this section we introduce a syntax directed version of the previous type system. Typing judgments are of the form $\Gamma;\Theta \vdash_{\mathsf{sd}} P \triangleright \Delta$ where environments are the same as above. First of all to obtain a full syntax directed type system we need to solve the problem of the non-deterministic introduction of ended sessions. The type system in Figure 31 uses (SWEAK) to add ended sessions. Here we introduce a *linear access* function with in mind the idea to insert in the linear environment an ended session only when it is strictly necessary. Linear access is of the form $\Delta \vdash \kappa : T \Rightarrow \Delta'$ in which we input Δ and κ and we obtain in output the type T of κ and the Δ' resulting from Δ without the assumption on κ. The idea is very simple, we ask to the linear access function accessing Δ to retrieve an

assumption about a session κ and if no assumption about κ exists then it returns an ended session. Accordingly we define $\Delta \vdash \kappa : T, \kappa' : U \Rightarrow \Delta'$ which allows to access two sessions per time. The linear access function is reported in Figure 32.

Syntax directed typing rules in Figure 33 exploit both the linear access function and the subtyping relation $\Delta_{\downarrow\{\tilde{\kappa}\}} \leq \Delta'$ to get rid of the rule (SWEAK). Rule (SACCSD) accesses Δ to retrieve the type of $\mathbf{x} : T'$ then it concludes if the type assumed for v is less than T', this last check is necessary due to the lack of (SWEAK); rule (SREQSD) is similar. Rules (SINSD) and (SOUTSD) add an input and an output action respectively to the type of the current session κ, similar for (SCHOICESD) which judges κ as an internal choice with only one option labeled with l. Before discussing (SBRANCHSD) let us take a look to the rule (SIFD) which is in part similar. In rule (SIFSD) we allow the process in each branch to be typed with a different Δ_i as long as there exists a common $\Delta_{\downarrow \bigcup_{i=1}^{2} \mathsf{dom}(\Delta_i)} \leq \Delta_i$. (Notice that we limit the domain of the subtyping relation.) The case is subtle and we need such check since it can be the case that one branch has some ended sessions that the other branch ignores, since a spurious ended session can be introduced by means of a throw instruction (see rule (STHROWSD)). As an example consider a little variant of the Example 5.2, $a(\mathbf{x}).b(\mathbf{y}).c(\mathbf{z}).\mathtt{if}\ test\ \mathtt{then}\ \mathbf{x}!\langle\langle\mathbf{y}\rangle\rangle.\mathbf{0}\ \mathtt{else}\ \mathbf{x}!\langle\langle\mathbf{z}\rangle\rangle.\mathbf{0}$ it is easy to see that the process is typed in $\vdash$ and then we want it to be typed in $\vdash_{\mathsf{sd}}$ as well. Consider the first branch, typed as $\Gamma; \Theta \vdash_{\mathsf{sd}} \mathbf{x}!\langle\langle\mathbf{y}\rangle\rangle.\mathbf{0} \triangleright \mathbf{x} :!(\mathtt{end}), \mathbf{y} : \mathtt{end}$ and the second branch is typed as $\Gamma; \Theta \vdash_{\mathsf{sd}} \mathbf{x}!\langle\langle\mathbf{z}\rangle\rangle.\mathbf{0} \triangleright \mathbf{x} :!(\mathtt{end}), \mathbf{z} : \mathtt{end}$ then the entire if-then-else instruction is typed with $\Delta = \mathbf{x} :!(\mathtt{end}), \mathbf{y} : \mathtt{end}, \mathbf{z} : \mathtt{end}$ since $\Delta_{\downarrow\{\mathbf{x},\mathbf{y},\mathbf{z}\}} \leq \mathbf{x} :!(\mathtt{end}), \mathbf{y} : \mathtt{end}$ and $\Delta_{\downarrow\{\mathbf{x},\mathbf{y},\mathbf{z}\}} \leq \mathbf{x} :!(\mathtt{end}), \mathbf{z} : \mathtt{end}$.

Another solution would be to disallow delegation of ended sessions adding a premise of the form $U \neq \mathtt{end}$ in both rules (SCATCH) and (SCATCHSD) (as the solution proposed in (6)). However we prefer to retain the full treatment rather than introducing this new kind of constraint.

Rule (SBRANCHSD) is similar but it allows the current session κ to have a different type T_i for each branch so we can judge the process as an external choice. We use the function same to handle the case in which equal labels are offered. Rule (SCATCHSD) checks, by means of subtyping, that the received session is used in accordance in the body P. Interestingly this check could be avoided in the presence of a complete subtyping relation, which automatically checks the sent sessions are also in subtyping relation. As already discussed we do not use the subtyping in depth since we employ the syntactic unifier to solve the set of constraints.

$$\Delta, \kappa : T \vdash \kappa : T \Rightarrow \Delta \qquad \frac{\kappa \notin \mathsf{dom}(\Delta)}{\Delta \vdash \kappa : \mathsf{end} \Rightarrow \Delta}$$

$$\Delta, \kappa : T, \kappa' : U \vdash \kappa : T, \kappa' : U \Rightarrow \Delta \qquad \frac{\{\kappa, \kappa'\} \cap \mathsf{dom}(\Delta) = \emptyset}{\Delta \vdash \kappa : \mathsf{end}, \kappa' : \mathsf{end} \Rightarrow \Delta}$$

$$\frac{\kappa \notin \mathsf{dom}(\Delta)}{\Delta, \kappa' : T \vdash \kappa : \mathsf{end}, \kappa' : T \Rightarrow \Delta} \qquad \frac{\kappa' \notin \mathsf{dom}(\Delta)}{\Delta, \kappa : T \vdash \kappa : T, \kappa' : \mathsf{end} \Rightarrow \Delta}$$

Figure 32: Linear access function

(SNILSD) $\Gamma; \Theta \vdash_{\mathsf{sd}} \mathbf{0} \triangleright \emptyset$

(SACCSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Delta \vdash \mathbf{x} : T' \Rightarrow \Delta' \quad \Gamma \vdash v : [T] \quad T \leq T'}{\Gamma; \Theta \vdash_{\mathsf{sd}} v(\mathbf{x}).P \triangleright \Delta'}$$

(SREQSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Delta \vdash \mathbf{x} : T' \Rightarrow \Delta' \quad \Gamma \vdash v : [T] \quad \overline{T'} \leq T}{\Gamma; \Theta \vdash_{\mathsf{sd}} \overline{v}(\mathbf{x}).P \triangleright \Delta'}$$

(SINSD)
$$\frac{\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Delta \vdash \kappa : T \Rightarrow \Delta'}{\Gamma; \Theta \vdash_{\mathsf{sd}} \kappa?(\tilde{x}).P \triangleright \Delta', \kappa : ?(\tilde{S}).T}$$

(SOUTSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Gamma \vdash_{\mathsf{sd}} \tilde{v} : \tilde{S} \quad \Delta \vdash \kappa : T \Rightarrow \Delta'}{\Gamma; \Theta \vdash_{\mathsf{sd}} \kappa!(\tilde{v}).P \triangleright \Delta', \kappa : !(\tilde{S}).T}$$

(SBRANCHSD)
$$\frac{\begin{array}{l}\emptyset \subset I = \{1, \ldots, n\} \\ \forall i \in I \; (\Gamma; \Theta \vdash_{\mathsf{sd}} P_i \triangleright \Delta_i \quad \Delta_i \vdash \kappa : T_i \Rightarrow \Delta_i' \quad \Delta'_{\downarrow \bigcup_{i=1}^{n} \mathsf{dom}(\Delta_i')} \leq \Delta_i')\end{array}}{\Gamma; \Theta \vdash_{\mathsf{sd}} \Sigma_{i=1}^{n} \kappa?(l_i).P_i \triangleright \Delta', \kappa : \&\{\mathsf{same}(l_j : T_j)\}_{j \in I}}$$

(SCHOICESD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Delta \vdash \kappa : T \Rightarrow \Delta'}{\Gamma; \Theta \vdash_{\mathsf{sd}} \kappa!(l).P \triangleright \Delta', \kappa : \oplus\{l : T\}}$$

(SCATCHSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Delta \vdash \kappa : T, \mathbf{x} : U \Rightarrow \Delta' \quad U' \leq U}{\Gamma; \Theta \vdash_{\mathsf{sd}} \kappa?(\!(\mathbf{x})\!).P \triangleright \Delta', \kappa : ?(U').T}$$

(STHROWSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Delta \vdash \kappa : T \Rightarrow \Delta'}{\Gamma; \Theta \vdash_{\mathsf{sd}} \kappa!\langle\!\langle \kappa' \rangle\!\rangle.P \triangleright \Delta', \kappa : !(U).T, \kappa' : U}$$

(SPARSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta_1 \quad \Gamma; \Theta \vdash_{\mathsf{sd}} Q \triangleright \Delta_2}{\Gamma; \Theta \vdash_{\mathsf{sd}} P | Q \triangleright \Delta_1, \Delta_2}$$

(SIFSD)
$$\frac{(\Gamma \vdash_{\mathsf{sd}} v_i : S \quad \Gamma; \Theta \vdash_{\mathsf{sd}} P_i \triangleright \Delta_i \quad \Delta_{\downarrow \bigcup_{i=1}^{2} \mathsf{dom}(\Delta_i)} \leq \Delta_i) \; i \in \{1, 2\}}{\Gamma; \Theta \vdash_{\mathsf{sd}} \texttt{if } v_1 = v_2 \texttt{ then } P_1 \texttt{ else } P_2 \triangleright \Delta}$$

(SNEWRSD)
$$\frac{\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta \quad \Delta \vdash r^+ : T', r^- : T'' \Rightarrow \Delta' \quad \overline{T''} \leq T'}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\nu r) P \triangleright \Delta'}$$

(SNEWSD)
$$\frac{\Gamma, a : S; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta}{\Gamma; \Theta \vdash_{\mathsf{sd}} (\nu a) P \triangleright \Delta}$$

(SRECSD)
$$\frac{\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash_{\mathsf{sd}} P \triangleright \tilde{\kappa} : \tilde{T'} \quad \tilde{\kappa} : \tilde{T}_{\downarrow\{\tilde{\kappa}\}} \leq \tilde{\kappa} : \tilde{T'} \quad \{\tilde{\kappa}\} = \mathsf{fpn}(P)}{\Gamma; \Theta \vdash_{\mathsf{sd}} \texttt{rec } X.P \triangleright \tilde{\kappa} : \tilde{T}}$$

(SPVARSD) $\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash_{\mathsf{sd}} X \triangleright \tilde{\kappa} : T$

Figure 33: Syntax directed typing rules

$$
\begin{array}{lcl}
\mathsf{same}(l_1 : T_1, \ldots, l_n : T_n) & = & \begin{cases} l_1 : T_1, \mathsf{same}(l_2 : T_2 \ldots, l_n : T_n)) & \text{if } l_1 \notin \{l_2, \ldots, l_n\} \\ \mathsf{same}(l_2 : T_2, \ldots, l_i : U, \ldots, l_n : T_n) & \text{if } l_1 = l_i \text{ and } U \leq T_1, T_i \end{cases} \\
\mathsf{same}(l : T) & = & l : T
\end{array}
$$

Figure 34: same function

Rule (SNEWRSD) uses the linear access to obtain the type of each polarized r and then it checks their compatibility. Finally rule (SRECSD) uses the subtyping relation between linear environments with domain limited to the free polarized names of P.

Example 5.14. Letting $T_\alpha = \mu\alpha.?(int).\alpha$ and $\Theta = X : \mathbf{x}\,T_\alpha, Y : \emptyset$ and $\Gamma = a : [T_\alpha], b : [?(int)]$, $Proxy$ from Example 5.1 is typed as:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x; int; \Theta \vdash_{\mathsf{sd}} X \triangleright \mathbf{x} : T_\alpha \quad \mathbf{x} : T_\alpha \vdash \mathbf{y} : \mathtt{end} \Rightarrow \mathbf{x} : T_\alpha}{\Gamma, x : int; \Theta \vdash_{\mathsf{sd}} \mathbf{y}!(x).X \triangleright \mathbf{x} : T_\alpha, \mathbf{y} :!(int)}\,(\text{SOUTSD})}{\Gamma, x : int; \Theta \vdash_{\mathsf{sd}} \bar{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \mathbf{x} : T_\alpha}\,(\text{SREQSD})}{\Gamma; \Theta \vdash_{\mathsf{sd}} \mathbf{x}?(x).\bar{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \mathbf{x} :?(int).T_\alpha \quad T_\alpha \leq ?(int).T_\alpha}\,(\text{SINSD})}{\Gamma; \Theta \vdash_{\mathsf{sd}} \mathtt{rec}\ X.\mathbf{x}?(x).\bar{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \mathbf{x} : T_\alpha}\,(\text{SRECSD})}{\Gamma; Y : \emptyset \vdash_{\mathsf{sd}} a(\mathbf{x}).\mathtt{rec}\ X.\mathbf{x}?(x).\bar{b}(\mathbf{y}).\mathbf{y}!(x).X \triangleright \emptyset}\,(\text{SACCSD}) \qquad \Gamma; Y : \emptyset \vdash_{\mathsf{sd}} Y \triangleright \mathbf{x} : \emptyset}{\Gamma; Y : \emptyset \vdash_{\mathsf{sd}} a(\mathbf{x}).\mathtt{rec}\ X.\mathbf{x}?(x).\bar{b}(\mathbf{y}).\mathbf{y}!(x).X | Y \triangleright \emptyset}\,(\text{SPARSD})}{\Gamma; \emptyset \vdash_{\mathsf{sd}} Proxy \triangleright \emptyset}\,(\text{SRECSD})
$$

Compare this typing with the one reported in Example 5.4 and notice that each application of the rule (SWEAK) is now checked inline.

We now prove the soundness of the syntax directed typing rules.

Proposition 5.15 (Soundness). *If* $\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta$ *then* $\Gamma; \Theta \vdash P \triangleright \Delta$.

Proof. The proof is by induction on the typing derivation of $\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta$ with case analysis on the last applied rule. Base cases are (SNILSD) and (SRECSD) and they are both straightforward. In the inductive cases when the last applied rule is:

- (SACCSD) and $\dot{P} = v(\mathbf{x}).P$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} v(\mathbf{x}).P \triangleright \Delta$. We have two cases depending whether $\Delta \vdash \mathbf{x} : \mathtt{end} \Rightarrow \Delta$ or not. In the former case the judgment follows directly by induction and rule (SWEAK). In the latter case we have $\Delta \vdash \mathbf{x} : T' \Rightarrow \Delta'$ for some T' or $\Delta = \Delta', \mathbf{x} : T'$ and $\Gamma \vdash v : [T]$ for some $T \leq T'$. The result follows applying rule (SWEAK) on $\Delta', \mathbf{x} : T'$ and rule (SACC) to conclude.

- (SREQSD) similar to the previous case.

- (SIFSD) and $\dot{P} = \texttt{if } v_1 = v_2 \texttt{ then } P_1 \texttt{ else } P_2$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i \triangleright \Delta_i$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} \dot{P} \triangleright \Delta$ for some $\Delta \leq \Delta_i$ (we omitted the domain restriction) for $i \in \{1, 2\}$. The thesis follows applying (SWEAK) in order to have $\Gamma; \Theta \vdash P_i \triangleright \Delta$ and (SIF) to conclude.

- (SINSD) and $\dot{P} = \kappa?(\tilde{x}).P$ and $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta$ and by induction $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P \triangleright \Delta$. In order to apply the rule (SIN) we use (SWEAK) if $\kappa \notin \mathsf{dom}(\Delta)$ and conclude.

- (SOUTSD), (SCHOICESD), (SCATCHSD) and (STHROWSD) similar to the previous case.

- (SRECSD) and $\dot{P} = \texttt{rec } X.P$ and $\Gamma; \Theta, X : \tilde{k}\tilde{T} \vdash_{\mathsf{sd}} P \triangleright \tilde{k} : \tilde{T}'$ and by induction $\Gamma; \Theta, X : \tilde{k}\tilde{T} \vdash P \triangleright \tilde{k} : \tilde{T}'$ where $\tilde{k} : \tilde{T} \leq \tilde{k} : \tilde{T}'$. The conclusion follows applying rules (SWEAK) and (SREC).

- (SNEWRSD) and $\dot{P} = (\nu r)P$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta$ and $\Delta \vdash r^+ : T', r^- : T'' \Rightarrow \Delta'$ and by induction $\Gamma; \Theta \vdash P \triangleright \Delta$. If either $r^+ \notin \mathsf{dom}(\Delta)$ or $r^- \notin \mathsf{dom}(\Delta)$ then it must be the case T', T'' equal to $\texttt{end}$ which conclude applying (SWEAK). Otherwise $\Delta = \Delta', r^+ : T', r^- : T''$ and $\overline{T''} \leq T'$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} \dot{P} \triangleright \Delta'$ and by an application of the the rule (SWEAK) we obtain $\Delta', r^+ : \overline{T''}, r^- : T''$ which allows to conclude $\Gamma; \Theta \vdash \dot{P} \triangleright \Delta'$ with rule (SNEWR).

- (SNEWSD) follows directly by induction.

- (SBRANCHSD) and $\dot{P} = \Sigma_{i=1}^{n} \kappa?(l_i).P_i$ and $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i \triangleright \Delta_i$ and $\Delta_i \vdash \kappa : T_i \Rightarrow \Delta_i'$ and $\Delta \leq \Delta_i'$ for some Δ and by induction $\Gamma; \Theta \vdash P_i \triangleright \Delta_i$ where $i \in \{1, \ldots, n\}$. We can conclude using the rule (SWEAK) on each $\Delta_i = \Delta_i', \kappa : T_i$ to obtain $\Delta, \kappa : U_i$ where U_i is the type returned by same relative to the usage of session κ by the process P_i; notice that by definition of same it holds that $T_i \leq U_i$ for all i. This allows to conclude with (SBRANCH) $\Gamma; \Theta \vdash \dot{P} \triangleright \Delta, \kappa : \&\{\mathsf{same}(l_j : T_j)\}_{j \in J}\}$.

□

The completeness theorem proves that each judgment produced by $\vdash$ is produced also by $\vdash_{\mathsf{sd}}$ with a greater linear environment.

Proposition 5.16 (Completeness). *If* $\Gamma; \Theta \vdash P \triangleright \Delta$ *then there exists* Δ' *s.t.* $\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta'$ *and* $\Delta \leq \Delta'$.

Proof. The proof is by induction on the typing derivation of $\Gamma; \Theta \vdash P \triangleright \Delta$ with case analysis on the last applied rule. Base cases are immediate and follow by the reflexivity of $\leq$ i.e.:$\Delta \leq \Delta$ for any Δ. In the inductive cases when the last applied rule is:

- (SAcc) and $\dot{P} = v(\mathbf{x}).P$ and $\Gamma; \Theta \vdash P \triangleright \Delta_1, \mathbf{x} : T$ and by induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta_2$ for some Δ_2 s.t. $\Delta_1, \mathbf{x} : T \leq \Delta_2$. By definition $\Delta_2 \vdash \mathbf{x} : T' \Rightarrow \Delta_2'$ for some Δ_2' s.t. $\Delta_1 \leq \Delta_2'$ and $T \leq T'$ which allows to conclude by an application of the rule (SAccSD).

- (SIf) and $\dot{P} = \texttt{if } v_1 = v_2 \texttt{ then } P_1 \texttt{ else } P_2$ and $\Gamma; \Theta \vdash P_i \triangleright \Delta$ and by induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i \triangleright \Delta_i$ for some Δ_i s.t. $\Delta \leq \Delta_i$ and $i \in \{1, 2\}$. By definition and by the last equation there exists Δ' s.t. $\Delta'_{\downarrow \bigcup_{i=1}^{2} \mathsf{dom}(\Delta_i)} \leq \Delta_i$ and $\Delta \leq \Delta'$ which allows the application of rule (SIfSD) to conclude with $\Gamma; \Theta \vdash_{\mathsf{sd}} \dot{P} \triangleright \Delta'$.

- (SRec) and $\dot{P} = \texttt{rec } X.P$ and $\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash P \triangleright \tilde{\kappa} : \tilde{T}$ and by induction $\Gamma; \Theta, X : \tilde{\kappa}\tilde{T} \vdash_{\mathsf{sd}} P \triangleright \tilde{\kappa} : \tilde{T}'$ where $\tilde{\kappa} : \tilde{T} \leq \tilde{\kappa} : \tilde{T}'$. (Notice that the set $\{\tilde{\kappa}\}$ is returned also by the inductive hypothesis since each $\tilde{T}$ is different from $\texttt{end}$.) Since $\{\tilde{\kappa}\} = \mathsf{fpn}(P)$, by definition $\tilde{\kappa} : \tilde{T}_{\downarrow \mathsf{fpn}(P)} \leq \tilde{\kappa} : \tilde{T}'$ which concludes by rule (SRecSD).

- (SIn) and $\dot{P} = \kappa?(\tilde{x}).P$ and $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P \triangleright \Delta_1, \kappa : T$ and by induction $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta_2$ with $\Delta_1, \kappa : T \leq \Delta_2$. By definition $\Delta_2 \vdash \kappa : T' \Rightarrow \Delta_2'$ for some Δ_2' s.t. $\Delta_1 \leq \Delta_2'$ and $T \leq T'$. The result follows since $?(\tilde{S}).T \leq ?(\tilde{S}).T'$ holds for any $\tilde{S}$.

- (SOut) and (SThrow) similar to the previous case.

- (SNewR) and $\dot{P} = (\nu r)P$ and $\Gamma; \Theta \vdash P \triangleright \Delta_1, r^+ : T, r^- : \overline{T}$ and by induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta_2$ and $\Delta_1, r^+ : T, r^- : \overline{T} \leq \Delta_2$. By definition we have $\Delta_2 \vdash r^+ : T', r^- : T'' \Rightarrow \Delta_2'$ and $T \leq T'$ and $\overline{T} \leq T''$ and $\Delta_1 \leq \Delta_2'$. As usual $\overline{T''} \leq T$ and by transitivity $\overline{T''} \leq T'$ which allows to conclude with (SNewRSD).

- (SBranch) and $\dot{P} = \Sigma_{i=1}^{n} \kappa?(l_i).P_i$ and $\Gamma; \Theta \vdash P_i \triangleright \Delta_1, \kappa : T_i$ and by induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i \triangleright \Delta_{2,i}$ and $\Delta_1, \kappa : T_i \leq \Delta_{2,i}$ and $i \in \{1, \ldots, n\}$. By definition $\Delta_{2,i} \vdash \kappa : T_i' \Rightarrow \Delta_{2,i}'$, $T_i \leq T_i'$ and $\Delta_1 \leq \Delta_{2,i}'$ and we can find Δ'' s.t. $\Delta''_{\downarrow \bigcup_{i=1}^{n} \mathsf{dom}(\Delta_{2,i}')} \leq \Delta_{2,i}'$ and $\Delta_1 \leq \Delta''$ and as discussed in the proof of Theorem 4.40, $\&\{l_i : T_i\}_{i \in J} \leq \&\{\mathsf{same}(l_i : T_i')_{i \in I}\}$ for each $J \subseteq I$ and $T_i \leq T_i'$ and $i \in J$. The conclusion

follows applying (SBRANCHSD) to obtain $\Gamma; \Theta \vdash \dot{P} \triangleright \Delta'', \kappa : \&\{\mathsf{same}(l_i : T'_i)_{i\in I}\}$.

- (SCATCH) and $\dot{P} = \kappa?(\!(\mathbf{x})\!).P$ and $\Gamma; \Theta \vdash P \triangleright \Delta_1, \kappa : T, \mathbf{x} : U$ and $\Gamma; \Theta \vdash \dot{P} \triangleright \Delta_1, \kappa :?(U).T$ and by induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P \triangleright \Delta_2$ with $\Delta_1, \kappa : T, \mathbf{x} : U \leq \Delta_2$. By definition we have $\Delta_2 \vdash \kappa : T', \mathbf{x} : U' \Rightarrow \Delta'_2$ and $T \leq T'$ and $U \leq U'$ and $\Delta_1 \leq \Delta'_2$. We can conclude applying rule (SCATCHSD) to obtain $\Gamma; \Theta \vdash \dot{P} \triangleright \Delta'_2, \kappa :?(U).T'$.

- (SPAR) and $\dot{P} = P_1|P_2$ and $\Gamma; \Theta \vdash P_i \triangleright \Delta_i$ and by induction $\Gamma; \Theta \vdash_{\mathsf{sd}} P_i \triangleright \Delta'_i$ and $\Delta_i \leq \Delta'_i$ for $i \in \{1, 2\}$. The conclusion follows since Δ_1, Δ_2 implies Δ'_1, Δ'_2 since by definition of $\Delta_i \leq \Delta'_i$, $\mathsf{dom}(\Delta'_i) \subseteq \mathsf{dom}(\Delta_i)$.

- (SNEW) follows directly by induction.

- (SWEAK) follows directly by transitivity of $\leq$ between linear environments.

□

As we have done before we can define an algorithm to extract a set of constraints starting from a process, s.t. if the set of constraints is satisfied for some substitution then the process is typable and vice versa. The algorithm `INF` (Figure 35) takes a process P, a standard typing environment Γ and a process environment Θ and returns a set of constraints $\mathcal{C}$ and a linear typing environment Δ. In the algorithm we use $\Delta_1 \leq_c \Delta_2$ defined as:

$$\Delta_1 \leq_c \Delta_2 = \begin{array}{l} \{(\Delta_1(\kappa) \leq \Delta_2(\kappa)) \mid \kappa \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \\ \cup\{(\Delta_1(\kappa) = \mathsf{end}) \mid \kappa \in \mathsf{dom}(\Delta_1) \setminus \mathsf{dom}(\Delta_2)\} \end{array}$$

$\leq_c$ generates constraints relative to the subtyping relation among linear environments since $\leq_c$ enjoys the following property.

Lemma 5.17. *If $\sigma \vDash \Delta_1 \leq_c \Delta_2$ then $\sigma\Delta_1 \leq \sigma\Delta_2$.*

Proof. Follows directly by the definition of $\Delta_1 \leq_c \Delta_2$ and the definition of $\Delta_1 \leq \Delta_2$. □

Proposition 5.18 (Soundness and Completeness of `INF`)**.** *Let* $\mathtt{INF}(P, \Gamma, \Theta) = (\mathcal{C}, \Delta)$. *$\sigma \vDash \mathcal{C}$ iff $\sigma\Gamma; \sigma\Theta \vdash_{\mathsf{sd}} P \triangleright \sigma\Delta$.*

VALUEINF $(x, \Gamma) = \Gamma(x)$ VALUEINF $(s, \Gamma) = \Gamma(s)$ VALUEINF $(n, \Gamma) = int$

INF $(\mathbf{0}, \Gamma, \Theta) = (\emptyset, \emptyset)$

INF $(v(\mathbf{x}).P, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta) =$ INF (P, Γ, Θ) **in** $\Delta \vdash \mathbf{x} : T \Rightarrow \Delta'$
  in $(\mathcal{C} \cup \{\alpha \leq T\} \cup \{\Gamma(v) = [\alpha]\}, \Delta')$ (where α fresh)

INF $(\overline{v}(\mathbf{x}).P, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta) =$ INF (P, Γ, Θ) **in** $\Delta \vdash \mathbf{x} : T \Rightarrow \Delta'$
  in $(\mathcal{C} \cup \{\overline{T} \leq \alpha\} \cup \{\Gamma(v) = [\alpha]\}, \Delta')$ (where α fresh)

INF $(\kappa?(x_1, \ldots, x_n).P, \Gamma, \Theta) =$
  let $(\mathcal{C}, \Delta) =$ INF $(P, \Gamma \cup \{x_1 : \beta_1, \ldots, x_n : \beta_n\}, \Theta)$ $\beta_1, ., \beta_n$ fresh
  in $\Delta \vdash \kappa : T \Rightarrow \Delta'$ **in** $(\mathcal{C}, \Delta' \cup \{\kappa : ?(\beta_1, \ldots, \beta_n).T\})$

INF $(\kappa!(v_1, \ldots, v_n).P, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta) =$ INF (P, Γ, Θ) **in** $\Delta \vdash \kappa : T \Rightarrow \Delta'$
  in let $S_1 =$ VALUEINF (v_1, Γ) $\ldots\ldots$ $S_n =$ VALUEINF (v_n, Γ)
  in $(\mathcal{C}, \Delta' \cup \{\kappa : !(S_1, \ldots, S_n).T\})$

INF $(\kappa!\langle\!\langle \kappa' \rangle\!\rangle.P, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta) =$ INF (P, Γ, Θ) **in** $\Delta \vdash \kappa : T \Rightarrow \Delta'$
  in $(\mathcal{C}, (\Delta' \cup \{\kappa : !(\alpha).T, \kappa' : \alpha\}))$ (where α fresh)

INF $(\kappa?((\mathbf{x})).P, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta) =$ INF (P, Γ, Θ) **in** $\Delta \vdash \kappa : T, \mathbf{x} : U \Rightarrow \Delta'$
  in $(\mathcal{C} \cup \{\alpha \leq U\}, \Delta' \cup \{\kappa : ?(\alpha).T\})$ (where α fresh)

INF (if $v_1 = v_2$ then P else $Q, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta_1) =$ INF (P, Γ, Θ) **in**
  let $(\mathcal{C}_1, \Delta_2) =$ INF (Q, Γ, Θ) **in**
  let $S_1 =$ VALUEINF (v_1, Γ) **in let** $S_2 =$ VALUEINF (v_2, Γ)
  in let $\Delta = \tilde{\kappa} : \tilde{\alpha}$ where $\tilde{\kappa} = \bigcup_{i=1}^{2} \mathsf{dom}(\Delta_i)$ and $\tilde{\alpha}$ fresh
  in $(\mathcal{C} \cup \mathcal{C}_1 \cup \{S_1 = S_2\} \cup \Delta \leq_c \Delta_1 \cup \Delta \leq_c \Delta_2, \Delta)$

INF $((\nu a)P, \Gamma, \Theta) =$ INF $(P, \Gamma \cup \{a : \beta\}, \Theta)$ (where β fresh)

INF $((\nu r)P, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta) =$ INF (P, Γ, Θ) **in** $\Delta \vdash r^+ : T', r^- : T'' \Rightarrow \Delta'$
  in $(\mathcal{C} \cup \{\overline{T''} \leq T'\}, \Delta')$

INF $(P|Q, \Gamma, \Theta) =$ **let** $(\mathcal{C}_1, \Delta_1) =$ INF (P, Γ, Θ) **in**
  let $(\mathcal{C}_2, \Delta_2) =$ INF (Q, Γ, Θ) **in**
  in $(\mathcal{C}_1 \cup \mathcal{C}_2, \Delta_1, \Delta_2)$

INF $(\Sigma_{i=1}^{n} \kappa?(l_i).P_i, \Gamma, \Theta) =$
  let $(\mathcal{C}_1, \Delta_1) =$ INF (P_i, Γ, Θ) $\ldots\ldots$ $(\mathcal{C}_n, \Delta_n) =$ INF (P_n, Γ, Θ)
  in $\Delta_i \vdash \kappa : T_i \Rightarrow \Delta'_i \quad i \in \{1, \ldots, n\}$
  in let $\Delta = \tilde{\kappa} : \tilde{\alpha}$ where $\tilde{\kappa} = \bigcup_{i=1}^{n} \mathsf{dom}(\Delta'_i)$ and $\tilde{\alpha}$ fresh
  in let $(\mathcal{C}', \widetilde{l' : T'}) = \mathsf{same}(l_1 : T_1, \ldots, l_n : T_n)$
 in $(\mathcal{C}' \cup \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_n \cup \Delta \leq_c \Delta'_1 \cup \ldots \cup \Delta \leq_c \Delta'_n, \Delta \cup \{\kappa : \&\{\widetilde{l' : T'}\}\})$
  where each type variables generated by same is fresh

INF $(\kappa!(l).P, \Gamma, \Theta) =$ **let** $(\mathcal{C}, \Delta) =$ INF (P, Γ, Θ)
  in $\Delta \vdash \kappa : T \Rightarrow \Delta'$
  in $(\mathcal{C}, \Delta' \cup \{\kappa : \oplus\{l : T\}\})$

INF (rec $X.P, \Gamma, \Theta) =$ **let** $\kappa_1, \ldots, \kappa_n = \mathsf{fpn}(P)$ **in**
  let $(\mathcal{C}, \kappa_1 : T'_1, \ldots, \kappa_n : T'_n) =$ INF $(P, \Gamma, \Theta \cup \{X : \kappa_1 \alpha_{X_1}, \ldots, \kappa_n \alpha_{X_n}\})$
  in $(\mathcal{C} \cup \{\alpha_{X_1} \leq T'_1, \ldots, \alpha_{X_n} \leq T'_n\}, \kappa_1 : \alpha_{X_1}, \ldots, \kappa_n : \alpha_{X_n})$

INF $(X, \Gamma, \Theta) =$ $(\emptyset, \tilde{\kappa} : \tilde{T})$ if $\Theta(X) = \tilde{\kappa}\tilde{T}$

Figure 35: The algorithm to extract constraints in Ocaml-like syntax

Proof. $\Rightarrow$) The proof is by induction on the recursive structure of a run of $\mathtt{INF}(P,\Gamma,\Theta) = (\mathcal{C},\Delta)$. Base cases are when $\dot{P} = X$ or $\dot{P} = \mathbf{0}$ and the empty set of constraints is solved by any substitution which allows also the conclusion with both rules (SPVARSD) and (SNILSD). We report some inductive cases, when the case relative to:

- $\dot{P} = v(\mathbf{x}).P$ is applied we have $\mathtt{INF}(P,\Gamma,\Theta) = (\mathcal{C},\Delta)$ and $\Delta \vdash \mathbf{x} : T \Rightarrow \Delta'$ and $\sigma \models \mathcal{C} \cup \{\alpha \leq T, \Gamma(v) = [\alpha]\}$ and by induction $\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} P \triangleright \sigma\Delta$ then
(SACCSD)
$$\frac{\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} P \triangleright \sigma\Delta \quad \sigma\Delta \vdash \mathbf{x} : \sigma T \Rightarrow \sigma\Delta' \quad \sigma\Gamma \vdash v : [\sigma\alpha] \quad \sigma\alpha \leq \sigma T}{\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} v(\mathbf{x}).P \triangleright \sigma\Delta'}\ .$$

- $\dot{P} = \kappa?(\!(\mathbf{x})\!).P$ is applied we have $\mathtt{INF}(P,\Gamma,\Theta) = (\mathcal{C},\Delta)$ and $\Delta \vdash \kappa : T, \mathbf{x} : U \Rightarrow \Delta'$ and $\sigma \models \mathcal{C} \cup \{\alpha \leq U\}$ and by induction $\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} P \triangleright \sigma\Delta$ then
(SCATCHSD)
$$\frac{\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} P \triangleright \sigma\Delta \quad \sigma\Delta \vdash \kappa : \sigma T, \mathbf{x} : \sigma U \Rightarrow \sigma\Delta' \quad \sigma\alpha \leq \sigma U}{\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} \kappa?(\!(\mathbf{x})\!).P \triangleright \sigma\Delta', \kappa : \sigma?(U).T}\ .$$

- $\dot{P} = \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ is applied, letting $i \in \{1,2\}$ we have $\mathtt{INF}(P_i,\Gamma,\Theta) = (\mathcal{C}_i,\Delta_i)$ and $\mathtt{VALUEINF}(v_i,\Gamma) = S_i$ and $\sigma \models \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 = S_2\} \cup \Delta \leq_c \Delta_1 \cup \Delta \leq_c \Delta_2$ and Δ is s.t. $\mathsf{dom}(\Delta) = \mathsf{dom}(\Delta_1) \cup \mathsf{dom}(\Delta_2)$ and for all $\kappa \in \mathsf{dom}(\Delta)$ then $\Delta(\kappa)$ is a fresh variable and by induction $\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} P_i \triangleright \sigma\Delta_i$ which suffices to conclude
(SIFSD)
$$\frac{(\Gamma \vdash_{\mathsf{sd}} v_i : \sigma S_i \quad \sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} P_i \triangleright \sigma\Delta_i \quad \sigma\Delta_{\downarrow \bigcup_{i=1}^{2} \mathsf{dom}(\Delta_i)} \leq \sigma\Delta_i)\ i \in \{1,2\}}{\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \triangleright \sigma\Delta}\ .$$
Notice that $\sigma \models \Delta \leq_c \Delta_i$ implies $\sigma\Delta_{\downarrow \bigcup_{i=1}^{2} \mathsf{dom}(\Delta_i)} \leq \sigma\Delta_i$ follows by Lemma 5.17.

- $\dot{P} = \mathtt{rec}\ X.P$ is applied we have $\mathtt{INF}(P,\Gamma,\Theta \cup \{X : \kappa_1\alpha_{X_1},\ldots,\kappa_n\alpha_{X_n}\}) = (\mathcal{C}, \kappa_1 : T'_1,\ldots,\kappa_n : T'_n)$ and $\{\tilde{\kappa}\} = \mathsf{fpn}(P)$ and $\sigma \models \mathcal{C} \cup \{\alpha_{X_1} \leq T'_1,\ldots,\alpha_{X_n} \leq T'_n\}$ which it suffices to conclude
(SRECSD)
$$\frac{\begin{array}{c}\sigma\Gamma;\sigma\Theta, X : \kappa_1\sigma\alpha_1,\ldots,\kappa_n\sigma\alpha_n \vdash_{\mathsf{sd}} P \triangleright \kappa_1 : \sigma T'_1,\ldots,\kappa_n : \sigma T'_n \\ (\kappa_1 : \sigma\alpha_1,\ldots,\kappa_n : \sigma\alpha_n)_{\downarrow \mathsf{fpn}(P)} \leq \kappa_1 : \sigma T'_1,\ldots,\kappa_n : \sigma T'_n\end{array}}{\sigma\Gamma;\sigma\Theta \vdash_{\mathsf{sd}} \mathtt{rec}\ X.P \triangleright \kappa_1 : \sigma\alpha_1,\ldots,\kappa_n : \sigma\alpha_n}\ .$$

$\Leftarrow$) The proof is similar to the other direction with opposite implications. We report some inductive cases. When the case is relative to:

- $\dot{P} = (\nu r)P$ is applied we have $\mathtt{INF}(P,\Gamma,\Theta) = (\mathcal{C},\Delta)$ and $\sigma\Theta;\sigma\Gamma \vdash P \triangleright \sigma\Delta$ and $\Delta \vdash r^+ : T', r^- : T'' \Rightarrow \Delta'$ and $\sigma T' \leq \sigma\overline{T''}$ or $\sigma \models \mathcal{C} \cup \{T' \leq \overline{T''}\}$ since by induction $\sigma \models \mathcal{C}$ holds.

- $\dot{P} = P_1|P_2$ and letting $i \in \{1,2\}$ we have $\texttt{INF}(P_i, \Gamma, \Theta) = (\mathcal{C}_i, \Delta_i)$ and by induction $\sigma \vDash \mathcal{C}_i$ which allows to conclude $\sigma \vDash \mathcal{C}_1 \cup \mathcal{C}_2$.

$\square$

Example 5.19. Take the three processes of Example 5.1, with $\Gamma = a : [\alpha_a], b : [\alpha_b]$ the set of constraints generated by $\texttt{INF}(Proxy, \Gamma, \emptyset)$ is

$$\overline{!(\beta_x)} \leq \alpha_b \quad \alpha_X \leq ?(\beta_x).\alpha_X \quad \alpha_a \leq \alpha_X$$

the set of constraints generated by $\texttt{INF}(Server, \Gamma, \emptyset)$ is

$$\alpha_b \leq ?(\beta_{x_1})$$

the set of constraints generated by $\texttt{INF}(Client, \Gamma, \emptyset)$ is

$$\alpha_{X_1} \leq !(int).!(int).\alpha_{X_1} \quad \overline{\alpha_{X_1}} \leq \alpha_a$$

Example 5.20. Take the process of Example 5.2, with $\Gamma = a : [\alpha_a], b : [\alpha_b]$ the set of constraints generated by $\texttt{INF}(Nulldel, \Gamma, \emptyset)$ is

$$\alpha_b \leq !(\alpha_{\mathbf{x}}) \quad \alpha_a \leq \alpha_{\mathbf{x}} \quad \texttt{end} \leq \alpha_a \quad \alpha_{\mathbf{x}_1} \leq \texttt{end} \quad \overline{?(\alpha_{\mathbf{x}_1})} \leq \alpha_b$$

5.4.1 The solve algorithm

In this section we try to solve in an automatic manner the set of constraints generated by INF. This time the task is not easy due to the session delegation and cyclic constraint dependencies (generated by recursive processes) which allows a type variable to appear anywhere in each constraint. Next we introduce some context notation which mainly we use to identify the place where a certain type variable appear. We consider the following set of n-holes session type contexts:

$$\mathbb{C} \quad ::= \quad [\![\cdot]\!] \mid \texttt{end} \mid \alpha \mid ?(\tilde{S}).\mathbb{C} \mid !(\tilde{S}).\mathbb{C} \mid \&\{l_i : \mathbb{C}_i\}_{i \in I} \mid \oplus \{l_i : \mathbb{C}_i\}_{i \in I} \mid \mu\alpha.\mathbb{C}$$

The usual operation $\mathbb{C}[\![T_1, \ldots, T_n]\!]$ of filling holes in $\mathbb{C}$ with the tuple $T_1, \ldots, T_n$ is defined only if $(\mathsf{fv}(T_1) \cup \ldots \cup \mathsf{fv}(T_n)) \cap \mathsf{bv}(\mathbb{C}) = \emptyset$. If $\mathbb{C}$ is a n-holes context we write $\vec{T}$ for $T_1, \ldots, T_n$ since each T_i is possibly not distinct, while $\vec{T} \leq \vec{U}$ is the point-wise extension of the subtyping relation.

The following lemma says that the subtyping relation is closed by session type contexts.

Lemma 5.21. *If $\vec{T} \leq \vec{U}$ then $\mathbb{C}[\![\vec{T}]\!] \leq \mathbb{C}[\![\vec{U}]\!]$ for all $\mathbb{C}$.*

Proof. We prove that $R = \{(\mathbb{C}[\![\vec{T}]\!], \mathbb{C}[\![\vec{U}]\!]) | \vec{T} \leq \vec{U}\}$ is a type simulation relation. Note that R contains all pairs (T, U) such that $T \leq U$ (by taking the identity context $\mathbb{C} = [\![\cdot]\!]$) and (T, T) as $T \leq T$. The proof is by inspection of the shape of $\mathbb{C}$, for example if $\mathbb{C} = \mu\alpha.\mathbb{C}'$ we have two cases depending if α if free in $\mathbb{C}'$ or not. In the latter case we have, after unfold is applied, $(\mathbb{C}'[\![\vec{T}]\!], \mathbb{C}'[\![\vec{U}]\!]) \in R$ which concludes. Otherwise we have, after unfold is applied, $(\mathbb{C}''[\![T_1, \ldots, T_i, \vec{T}, T_{i+1}, \ldots, T_n]\!], \mathbb{C}''[\![U_1, \ldots, U_i, \vec{U}, U_{i+1}, \ldots, U_n]\!]) \in R$ for some $\mathbb{C}''$, i and $\vec{T} = T_1, \ldots, T_n$ and $\vec{U} = U_1, \ldots, U_n$ which concludes. $\square$

Next we try to write down a solving algorithm based on the naive intuition that we attempt to insert each constraint of the form $\alpha \leq T$ in each other constraint where α appears under some context $\mathbb{C}$ trying to find some type mismatching. In fact using Lemma 5.21 we can turn a constraint of the form $\alpha \leq T$ into an equal constraint of the form $\mathbb{C}[\![\alpha]\!] \leq \mathbb{C}[\![T]\!]$ and then using the transitivity of $\leq$ merge two constraints, for example if $\alpha \leq T$ and $T_1 \leq \mathbb{C}[\![\alpha]\!]$ we have $\mathbb{C}[\![\alpha]\!] \leq \mathbb{C}[\![T]\!]$ and then $T_1 \leq \mathbb{C}[\![T]\!]$. In the same way we can merge a constraint $\alpha \leq T$ with a constraint $\overline{\mathbb{C}[\![\alpha]\!]} \leq T_1$ applying Lemma 5.21 on $\overline{T} \leq \overline{\alpha}$ instead. The following `solve` algorithm takes a set of constraints and tries to discover if there exists a substitution that satisfies such set:

$\mathtt{solve}(\mathcal{C})$:

1. if $S_1 = S_2, \mathcal{C}'$ then $\mathtt{solve}(\sigma_{S_1=S_2}\mathcal{C})$
2. if $\overline{T_1} \leq T_2, \mathcal{C}'$ when the variables in the set $\mathsf{fv}(\overline{T_1} \leq T_2) \setminus \mathsf{fcv}(\overline{T_1} \leq T_2)$ have unique occurrence in $\overline{T_1} \leq T_2$ and do not appear in $\mathcal{C}'$ then $\mathtt{solve}(\sigma_{\overline{T_1} \leq T_2}\mathcal{C}')$
3. if $\alpha \leq T_1, \ldots, \alpha \leq T_n, T_1' \leq \mathbb{C}_1'[\![\alpha]\!], \ldots, T_m' \leq \mathbb{C}_m'[\![\alpha]\!], \overline{\mathbb{C}_1''[\![\alpha]\!]} \leq T_1'', \ldots, \overline{\mathbb{C}_k''[\![\alpha]\!]} \leq T_k'', \mathcal{C}'$ when either $m > 0$ or $k > 0$ and each T_i' can be optionally overlined and there not exist in $\mathcal{C}', \mathbb{C}_1', \ldots, \mathbb{C}_m', \mathbb{C}_1'', \ldots, \mathbb{C}_k''$, any occurrence of α then $\mathtt{solve}(\mathcal{C}' \cup I \cup J)$ where:

$$I = \left\{ \begin{matrix} T_1' \leq \mathbb{C}_1'[\![\mu\alpha(T_1)]\!], & \ldots, & \ldots, & T_1' \leq \mathbb{C}_1'[\![\mu\alpha(T_n)]\!] \\ \vdots & \ddots & \ddots & \vdots \\ T_m' \leq \mathbb{C}_m'[\![\mu\alpha(T_1)]\!], & \ldots, & \ldots, & T_m' \leq \mathbb{C}_m'[\![\mu\alpha(T_n)]\!] \end{matrix} \right\}$$

$$J = \left\{ \begin{matrix} \overline{\mathbb{C}_1''[\![\mu\alpha(T_1)]\!]} \leq T_1'', & \ldots, & \ldots, & \overline{\mathbb{C}_1''[\![\mu\alpha(T_n)]\!]} \leq T_1'' \\ \vdots & \ddots & \ddots & \vdots \\ \overline{\mathbb{C}_k''[\![\mu\alpha(T_1)]\!]} \leq T_k'', & \ldots, & \ldots, & \overline{\mathbb{C}_k''[\![\mu\alpha(T_n)]\!]} \leq T_k'' \end{matrix} \right\}$$

4. if $\alpha \leq T_1, \ldots, \alpha \leq T_n, \mathcal{C}'$ when there not exist in $\mathcal{C}'$ any constraint involving α then ρ=<u>localsolve</u>($\bigcup_{i,j=1}^{n}\{\mu\alpha(T_i)\overset{c}{\wedge}\mu\alpha(T_j)\}$) and $\mathtt{solve}(\rho\mathcal{C}')$
5. if $\overline{T_1} \leq \alpha, \ldots, \overline{T_n} \leq \alpha, \mathcal{C}'$ when there not exist in $\mathcal{C}'$ any constraint involving α then ρ=<u>localsolve</u>($\bigcup_{i,j=1}^{n}\{\mu\alpha(T_i)\overset{c}{\wedge}\mu\alpha(T_j)\}$) and $\mathtt{solve}(\rho\mathcal{C}')$
6. if $\mathcal{C} = \emptyset$ then *success* else *fail*

We informally describe the algorithm. In line 1 we solve the unification constraints, in line 2 we solve inequalities s.t. $\sigma_{\overline{T_1} \leq T_2}$ is the most general solver (see Proposition 3.30). In line 3 we take all constraints involving α and then we merge them in all possible ways by means of sets I and J. Either the set of constraints with $\mathbb{C}_i'[\![\alpha]\!]$ or the set of constraints with

$\overline{\mathbb{C}''_j[\![\alpha]\!]}$ can be optional but at least one of these is mandatory else line 4 is applied. Intuitively if we cannot mix α with other constraints then we are forced to solve all constraints involving α locally. The condition of each T'_i optionally overlined means that we can have both constraints of the form $T'_i \leq \mathbb{C}'_i[\![\alpha]\!]$ or of the form $\overline{T'_i} \leq \mathbb{C}'_i[\![\alpha]\!]$.

We take care to postpone the solution of constraints involving α where α appears elsewhere in other constraints (e.g. in a context like $\mathbb{C}[\![!(\alpha).T]\!]$) since it means that e.g. α is involved in a session delegation. We use the function $\mu\alpha(T)$ to close the recursion, defined as: $\mu\alpha(T) = \mu\alpha.T$ if $\alpha \in (\mathsf{fv}(T) \setminus \mathsf{fcv}(T))$ or $\mu\alpha(T) = T$ if $\alpha \notin \mathsf{fv}(T)$ or *undefined* otherwise.

Lines 4 and 5 compute locally the meet and the join with the help of localsolve. Notice that in lines 4 and 5 the integer n can be also equal to 1, i.e. there is only one constraint of the form $\alpha \leq T$ or only one constraint of the form $\overline{T} \leq \alpha$.

Lemma 5.22. `solve` *terminates.*

Proof. We define an order on $\mathcal{C}$ which also serves as a measure for termination. We identify the set $\boldsymbol{\alpha}(\mathcal{C}) = \{\alpha | \alpha \leq T \in \mathcal{C}\}$. At each iteration either decrement, the cardinality of $|\boldsymbol{\alpha}(\mathcal{C})|$ (line 3) or the number of total constraints (lines 1,2,4,6). The well founded order $\mathcal{C} < \mathcal{C}'$ is defined if $|\boldsymbol{\alpha}(\mathcal{C})| < |\boldsymbol{\alpha}(\mathcal{C}')|$ or if $|\boldsymbol{\alpha}(\mathcal{C})| = |\boldsymbol{\alpha}(\mathcal{C}')|$ and $|\mathcal{C}| < |\mathcal{C}'|$. Since each recursive invocation of `solve` preserves this order we are authorized also to induct on its recursive structure. □

The following is a technical lemma which allows to extract a type U and its counterpart from a subtyping relation. If U does not give its contribute to the subtyping relation, e.g. because it is under a not considered choice, then the subtyping relation also holds for any type in place of U.

Lemma 5.23. *If $T \leq \mathbb{C}[\![U]\!]$ then either there exists $\mathbb{C}', T'$ s.t. $T = \mathbb{C}'[\![T']\!]$ and $T' \leq U$ or $T \leq \mathbb{C}[\![V]\!]$ for any V.*

Proof. We simply run the subtyping algorithm until a conclusion of the form $T' \leq U$ is encountered. If such conclusion is never encountered then U is unnecessary in the subtyping algorithm. Remember that the context is not allowed do bind any type variables in U so U is never unfolded by a recursion. We also must take care marking U to avoid clashing with alias of U within context $\mathbb{C}$. □

In order to state the completeness of the resolution algorithm we classify some cyclic dependencies among constraints. In particular we extract dependencies among constraints that yield the function $\mu\alpha$ undefined and dependencies that prevent applicability of lines 2,3,4,5 of `solve` due to cyclic dependencies among α and the rest of constraints set $\mathcal{C}'$ e.g. constraints of the form $\alpha \leq !(\alpha)$ and $\alpha_1 \leq !(\alpha_2), \alpha_2 \leq ?(\alpha_1)$ respectively.

Definition 5.24 (Cyclic dependencies)**.** A list of constraints $T_1 \leq T'_1, \ldots, T_{n+1} \leq T'_{n+1}$ is $\alpha_1, \ldots, \alpha_n$-dependent if $\forall i \in 1, \ldots, n$, $\alpha_i \in (\mathsf{fcv}(T_i) \cup \mathsf{fcv}(T'_i))$ and $\alpha_i \in (\mathsf{fv}(T_{i+1}) \cup \mathsf{fv}(T'_{i+1})) \setminus (\mathsf{fcv}(T_{i+1}) \cup \mathsf{fcv}(T'_{i+1}))$. A $\alpha_1, \ldots, \alpha_n$-dependent list of constraints is a cycle if $\alpha_1 = \alpha_n$.

Definition 5.25 (Cyclic dependencies into a set)**.** A set $\mathcal{C}$ *contains a* $\alpha_1, \ldots, \alpha_n$*-dependency* if there exists a $\alpha_1, \ldots, \alpha_n$-dependent list of constraints $T_1 \leq T'_1, \ldots, T_{n+1} \leq T'_{n+1}$ s.t. for all i, $T_i \leq T'_i \in \mathcal{C}$. A set of constraints $\mathcal{C}$ *contains a cycle* if it contains a $\alpha_1, \ldots, \alpha_n$-dependency which is a cycle.

Definition 5.26 (Non-admitted dependencies)**.** A set $\mathcal{C}$ of constraints has a *non-admitted* dependency if applying the resolution steps of `solve` zero or more times it reduces to some $\mathcal{C}'$ which contains a cycle.

In the following we assume sets of constraints that has not non-admitted dependencies. Notice that the `solve` reported in Figure 23 does not need such definitions as they are accounted implicitly by the algorithm (i.e. keeping constraints relative to service invocation).

We now prove that every time `solve` succeeds on a certain set of constraints $\mathcal{C}$ then there exists a substitution that solves $\mathcal{C}$. Vice versa if a substitution that solves $\mathcal{C}$ exists and neither line 4 nor line 5 are applied and line 6 is applied with the empty set then `solve` succeeds. This means that `solve` makes the most general decisions as long as it is not asked to solve locally constraints. In fact, if one of those lines of `solve` is applied it means that we have some constraints containing a session type variable that has no direct dependencies with other constraints. Thus `solve` is complete for closed systems without free names and if for each partner in the system it is also specified the dual counterpart i.e. for each service request there is at least one service accept and vice versa. We call such process *fully specified process*. One can simply verify this fact by inspecting `INF` that with the above hypothesis does not generate constraints with open dependencies. First we note that for each constraint of the form

$V \leq \mathbb{C}[\![\alpha']\!]$ (where V is optionally overlined) or of the form $\overline{\mathbb{C}[\![\alpha']\!]} \leq V$ there exists at least one respective constraint of the form $\alpha' \leq T$. Let us proceed by case inspection on `INF` observing rules that generate free variables that are not free communicated variables.

- The case for the if-then-else has Δ in the conclusions but we generate the set of constraints $\Delta \leq_c \Delta_1 \cup \Delta \leq_c \Delta_2$.
- The case for the external choice is similar to the previous one.
- The case for process variable generates free variables in the conclusion but since the process is closed there exists for each α' a constraint of the form $\alpha' \leq T$ inserted by the relative recursion construct.
- The case for a throw instruction generates a free variable in the conclusion but since the process is fully specified there exists a relative catch instruction that generates a constraint of the form $\alpha'' \leq U$ and a relative constraint (due to a constraint generated for the session, subject of the delegation) that unifies α' with α''. Moreover α' appears only in two constraints, the current one with $\mathbb{C}[\![\alpha']\!]$ and as argument of the the constraint relative to the throw. The same holds for α''.
- The case for service request generate a constraint of the form $\overline{T'} \leq \alpha'$ but there exists a service accept with at least a relative constraint of the form $\alpha' \leq T$.

Notice that also the converse is true, for every constraint of the form $\alpha' \leq T$ there exists a relative constraint either of the form $V \leq \mathbb{C}[\![\alpha']\!]$ (where V is optionally overlined) or of the form $\overline{\mathbb{C}[\![\alpha']\!]} \leq V$ since the process has not free names. Furthermore one can simply prove (by induction on `INF` observing that only fresh variables are generated) that in both $\alpha \leq \mathbb{C}[\![\alpha']\!]$ and in $\overline{\mathbb{C}[\![\alpha']\!]} \leq \alpha$, α' has a single occurrence in $\mathbb{C}[\![\alpha']\!]$ but for recursion. However recursion implies a cycle among free variables that are not free communicated variables (notice that these dependencies are different from those defined in Definition 5.24), then we can solve the cycle in the inverse order, starting from the constraint relative to the recursion construct towards the constraint relative to the process variable. Hence, line 3 of `solve` is applied until a set of constraints with only free variables that are also free communicated variables is obtained and line 2 of `solve` removes all remaining constraints by applying the syntactic unifier.

Theorem 5.27 (Soundness and Completeness of solve)**.** *Let $\mathcal{C}$ s.t. $(\mathcal{C}, \Delta) = \mathtt{INF}(P, \Gamma, \Theta)$. If* solve $(\mathcal{C}) = success$ *then there exists σ such that $\sigma \vDash \mathcal{C}$ and $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(\mathcal{C})$. Vice versa if $\mathcal{C}$ has not non-admitted dependencies and neither lines 4 nor 5 of the algorithm are applied and line 6 is applied with the empty set of constraints then if there exists σ such that $\sigma \vDash \mathcal{C}$ then* solve $(\mathcal{C}) = success$.

Proof. $\Rightarrow$) By induction on the recursive structure of a run of solve with case analysis on the last applied line of the algorithm. In the base case when the line 6 of the algorithm is applied $\mathcal{C} = \emptyset$ and the empty set of constraints is solved by the empty substitution. In the inductive cases when the last line is:

- line 1 and $\mathtt{solve}(S_1 = S_2, \mathcal{C})$, then by induction there exists σ s.t. $\sigma \vDash \sigma_{S_1=S_2}\mathcal{C}$ and by Lemma 3.33, $\sigma_{S_1=S_2}\sigma \vDash \mathcal{C}$ and by definition $\sigma_{S_1=S_2}\sigma \vDash S_1 = S_2, \mathcal{C}$. Notice that the composition is defined since by induction $\mathsf{dom}(\sigma) \subseteq \mathsf{var}(\sigma_{S_1=S_2}\mathcal{C})$ (see the similar proof of Theorem 4.34).

- line 2 it is similar to the previous case.

- line 3 and $\mathcal{C} = \alpha \leq T_1, \ldots, \alpha \leq T_n, T'_1 \leq \mathbb{C}'_1[\![\alpha]\!], \ldots, T'_m \leq \mathbb{C}'_m[\![\alpha]\!], \overline{\mathbb{C}''_1[\![\alpha]\!]} \leq T''_1, \ldots, \overline{\mathbb{C}''_k[\![\alpha]\!]} \leq T''_k, \mathcal{C}'$ and $\mathtt{solve}(\mathcal{C}) = success$ and the two sets I and J defined consequently. By induction solve $(\mathcal{C}' \cup I \cup J) = success$ implies there exists σ s.t. $\sigma \vDash \mathcal{C}' \cup I \cup J$. Applying Lemma 5.23 we define

$$I' = \left\{ \begin{matrix} U_1 \leq \mu\alpha(T_1), & \ldots, & U_1 \leq \mu\alpha(T_n) \\ \vdots & \ddots & \vdots \\ U_m \leq \mu\alpha(T_1), & \ldots, & U_m \leq \mu\alpha(T_n) \end{matrix} \right\}$$

$$J' = \left\{ \begin{matrix} \overline{\mu\alpha(T_1)} \leq U'_1, & \ldots, & \overline{\mu\alpha(T_n)} \leq U'_1 \\ \vdots & \ddots & \vdots \\ \overline{\mu\alpha(T_1)} \leq U'_k, & \ldots, & \overline{\mu\alpha(T_n)} \leq U'_k \end{matrix} \right\}$$

where each element is optional and depends on the result of Lemma 5.23 if or not $\mu\alpha(T_1)$ contributes to the subtyping relation. For simplicity we consider only the case for I' alone other cases are

similar. Set I' gives the following inequalities:

$$\sigma(U_i \leq \bigvee_{j=1}^{n} (\mu\alpha(T_j))) \text{ for all } i \in \{1, \ldots, m\} \tag{5.1}$$

$$\sigma(\bigwedge_{i=1}^{m} (U_i) \leq \mu\alpha(T_j)) \text{ for all } j \in \{1, \ldots, n\} \tag{5.2}$$

The result follows substituting α either with $\sigma(\bigvee_{j=1}^{n}(\mu\alpha(T_j)) \vee \bigwedge_{i=1}^{m}(U_i))$ or with $\sigma(\bigvee_{j=1}^{n}(\mu\alpha(T_j)) \wedge \bigwedge_{i=1}^{m}(U_i))$, e.g. letting $T_\alpha = \sigma(\bigvee_{i=j}^{n}(\mu\alpha(T_j)) \vee \bigwedge_{i=1}^{m}(U_i))$ we have $[^{T_\alpha}/_\alpha]\sigma(U_i \leq \alpha)$ (by equation (5.1)) for all $i \in \{1, \ldots, m\}$ and $[^{T_\alpha}/_\alpha]\sigma(\alpha \leq T_j)$ for all $j \in \{1, \ldots, n\}$. Notice that the algorithm does not apply any alpha renaming since at each iteration the set of free names is different, then we can safely reverse the function $\mu\alpha(T)$. However, in general the converse of Lemma 4.24, which is used to reverse $\mu\alpha(T)$, does not hold.

- line 4 and $\alpha \leq T_1, \ldots, \alpha \leq T_n, \mathcal{C}$ and by induction there exists σ s.t. $\sigma \vDash \rho\mathcal{C}$. By Lemma 3.34 we know that exists ρ and the intersection among all the pairs holds then we can use Lemma 3.33 to conclude $\rho\sigma \vDash \alpha \leq T_1, \ldots, \alpha \leq T_n, \mathcal{C}$.

- line 5 this case is similar to the previous one but in addition we compute the intersection of each $\overline{T_i}$ which is the least upper bound of each T_i.

$\Leftarrow$) We proceed by induction on the resolution steps of `solve` with case analysis on the last applied line of the algorithm. As we have done in the proof of Theorem 4.34 we must also prove that the substitution used by the inductive hypothesis is implied by the premise. In the base case the empty substitution solves the empty set of constraints (which is empty by hypothesis). In the inductive cases when the last line is:

- lines 1 and 2 similar to the relative cases in the proof of Theorem 4.34.

- line 3 and $\mathcal{C} = \underline{\alpha \leq T_1, \ldots, \alpha} \leq T_n, T_1' \leq \mathbb{C}_1'[\![\alpha]\!], \ldots, T_m' \leq \mathbb{C}_m'[\![\alpha]\!], \overline{\mathbb{C}_1''[\![\alpha]\!]} \leq T_1'', \ldots, \overline{\mathbb{C}_k''[\![\alpha]\!]} \leq T_k'', \mathcal{C}'$ and $\sigma \vDash \mathcal{C}$ and the two sets I and J defined consequently. We show that a generic element of I and J is solved by σ. Take $\sigma \vDash \alpha \leq T_i, T_j' \leq \mathbb{C}'[\![\alpha]\!]$, by Lemma 5.21

and Lemma 4.24 (which is applicable by hypothesis as $\mathcal{C}$ has not non-admitted dependencies) $\sigma \vDash \mathbb{C}'[\![\alpha]\!] \leq \mathbb{C}'[\![\mu\alpha(T_i)]\!], T'_j \leq \mathbb{C}'[\![\alpha]\!]$ and by transitivity $\sigma \vDash T'_j \leq \mathbb{C}'[\![\mu\alpha(T_i)]\!]$. Take $\sigma \vDash \alpha \leq T_i, \overline{\mathbb{C}''[\![\alpha]\!]} \leq T''_j$ and $\sigma \vDash \overline{\mathbb{C}''[\![\mu\alpha(T_i)]\!]} \leq \overline{\mathbb{C}''[\![\alpha]\!]}, \overline{\mathbb{C}''[\![\alpha]\!]} \leq T''_j$ and we can conclude by transitivity.

□

Example 5.28. We apply `solve` to the set of constraints generated in the Example 5.19. In the first column we report the line of `solve` applied at each step. After two iterations we have

Step	Constraints
3	$\alpha_a \leq \mu\alpha_X.?(\beta_x).\alpha_X \quad \overline{!(\beta_x)} \leq \alpha_b$ $\mu\alpha_{X_1}.?(int).?(int).\alpha_{X_1} \leq \alpha_a \quad \alpha_b \leq ?(\beta_{x_1})$
2	$\mu\alpha_X.?(\beta_x).\alpha_X \leq \mu\alpha_{X_1}.?(int).?(int).\alpha_{X_1}$ $\overline{!(\beta_x)} \leq \alpha_b \quad \alpha_b \leq ?(\beta_{x_1})$
3	$\overline{!(int)} \leq \alpha_b \quad \alpha_b \leq ?(\beta_{x_1})$
2	$?(int) \leq ?(\beta_{x_1})$
6	$success$

Example 5.29. We apply `solve` to the set of constraints generated in the Example 5.20

Step	Constraints
3	$\alpha_b \leq !(\alpha_{\mathbf{x}}) \quad \alpha_a \leq \alpha_{\mathbf{x}} \quad \mathtt{end} \leq \alpha_a \quad \alpha_{\mathbf{x}_1} \leq \mathtt{end} \quad \overline{?(\alpha_{\mathbf{x}_1})} \leq \alpha_b$
3	$\mathtt{end} \leq \alpha_{\mathbf{x}} \quad \alpha_b \leq !(\alpha_{\mathbf{x}}) \quad \alpha_{\mathbf{x}_1} \leq \mathtt{end} \quad \overline{?(\alpha_{\mathbf{x}_1})} \leq \alpha_b$
2	$\overline{?(\alpha_{\mathbf{x}_1})} \leq !(\alpha_{\mathbf{x}}) \quad \mathtt{end} \leq \alpha_{\mathbf{x}} \quad \alpha_{\mathbf{x}_1} \leq \mathtt{end}$
3	$\mathtt{end} \leq \alpha_{\mathbf{x}} \quad \alpha_{\mathbf{x}} \leq \mathtt{end}$
2	$\mathtt{end} \leq \mathtt{end}$
6	$success$

5.5 Further issues on the completeness of `solve`

Theorem 5.27 proves a general result on the completeness of `solve` without making any assumption on the type of constraints it can handle.

We showed in Section 3.4.1 that in order to achieve the completeness of `solve` in the general case at least one has to implement the exponential version of localsolve (and to prove Claim 3.4.1). Let us now make an assumption about the constraint set in order to do better with respect to the completeness. We consider a closed process P typed without assumptions in Γ (i.e. the set of constraints returned by $(\mathcal{C}, \Delta) = \texttt{INF}(P, \emptyset, \emptyset)$) and we show that in this manner `solve` is complete up-to non-admitted dependencies. In order to prove that, we first observe that localsolve reported in Figure 11 is also complete in the following sense:

Proposition 5.30. *If $\rho \vDash \mathcal{C}$ and variables in $\mathsf{fv}(\mathcal{C}) \setminus \mathsf{fcv}(\mathcal{C})$ have a unique occurrence in $\mathcal{C}$ then there exists σ s.t. $\sigma =$localsolve$(\mathcal{C})$ and $\rho = \sigma_*\sigma'$ for some σ_*, σ' s.t. $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma_*)$ and for all $\alpha \in \mathsf{dom}(\sigma)$ if $\alpha \in \mathsf{fcv}(\mathcal{C})$ then $\sigma(\alpha) = \sigma_*(\alpha)$.*

Proof. The proof is by induction on the resolution steps of localsolve$(\mathcal{C})$. If line 1 is applied the proof is similar to the proof of Theorem 4.34. If line 2 is applied $\rho \vDash \alpha \overset{c}{\wedge} T, \mathcal{C}$ and we must prove that this implies $\rho' \vDash [^T/_\alpha]\mathcal{C}$. Since by hypothesis $\alpha \notin \mathsf{fv}(\mathcal{C})$ then any substitution for α holds so take ρ' obtained from ρ removing the substitution for α then $[^T/_\alpha]\rho' \vDash \mathcal{C}$, which concludes by an application of Lemma 3.33. If the line 3 is applied we can conclude with Lemma 3.32. □

Finally we can states the completeness of `solve`.

Theorem 5.31. *Let $\mathcal{C}$ s.t. $(\mathcal{C}, \Delta) = \texttt{INF}(P, \emptyset, \emptyset)$ and $\mathcal{C}$ has not non-admitted dependencies. If $\sigma \vDash \mathcal{C}$ then* `solve` $(\mathcal{C}) =$ *success.*

Proof. The proof is by induction on the resolution steps of `solve`. We proceed considering the excluded cases from the completeness stated in Theorem 5.27. In particular each constraint of the form $V \leq \mathbb{C}[\![\alpha']\!]$ (where V optionally overlined) or of the form $\overline{\mathbb{C}[\![\alpha']\!]} \leq V$ where $\mathbb{C} \neq [\![\cdot]\!]$ that does not have a respective constraint of the form $\alpha' \leq T$ is due to a throw instruction. In this case α' has linear occurrence in $\mathbb{C}$. Hence when lines 4 and 5 of `solve` are applied we can use Proposition 5.30.

To conclude we must prove that line 6 is applied with the empty set of constraints which is immediate since every remaining constraint is of the form $\overline{T_1} \leq T_2$ with T_1 and T_2 different from a type variable. However $\overline{T_1} \leq T_2$ fits the pre-condition of line 2 because it can only be created using line 3. □

The previous theorem says that if `solve` fails on a certain process P then the set of constraints generated by `INF` for P has non-admitted dependencies. We now characterize all such processes P. For simplicity we do not consider delegation; construction with delegation is more involved and does not add nothing interesting. First of all we observe that every service accept and every service request relative to a service v is constrained with the same fresh session type variable exploiting assumptions in Γ. Consequently we can refer a service by its associated session type variable and vice versa we can use a session type variable to refer a service.

The support function $\mathsf{unified}(v)$ returns w if the session type variable relative to v is unified with the session type variable relative to w during the running of `solve` on a certain set of constraints or it returns v otherwise (we omit the set of constraints which is clear from the context from time to time). Given a process P with all free and bound names different we build the graph $G_P = (V, E)$ such that:

- the set of vertexes V is the set $\{w | v(\mathbf{x}).Q \text{ or } \overline{v}(\mathbf{x}).Q \text{ is a subprocess of } P \quad \text{for some} \quad Q, \mathbf{x} \quad \text{and} \quad \mathsf{unified}(v) = w\}$

- the set of edges E is the set $\{(v', w') | v(\mathbf{x}).Q$ or $\overline{v}(\mathbf{x}).Q$ is a subprocess of P and w is either sent or received through $\mathbf{x}$ and $\mathsf{unified}(v) = v'$ and $\mathsf{unified}(w) = w'$ and $w' \in V\}$

The construction of G_P is simple, we reason up-to unification in order to identify variables with the real service they are instantiated with. It is easy to see that the set of constraints generated by $\mathtt{INF}(P, \Gamma, \Theta)$ has not non-admitted dependencies iff G_P is acyclic. In order to prove that we introduce a further definition which extends dependency to a generic set of constraints.

Definition 5.32 ($\alpha_1, \ldots, \alpha_n$-dependency into a set)**.** A set $\mathcal{C}$ of constraints has an $\alpha_1, \ldots, \alpha_n$-dependency if applying the resolution steps of `solve` zero or more times it reduces to some $\mathcal{C}'$ which contains a set $\alpha_1, \ldots, \alpha_n$-dependent.

Lemma 5.33. *Let P a process s.t. $G_P = (V, E)$ is acyclic and there exist $\mathcal{C}, \Delta$ s.t. $(\mathcal{C}, \Delta) = \mathtt{INF}(P, \Gamma, \Theta)$ and $v \in \mathsf{dom}(\Gamma)$ implies $\Gamma(v) = \beta$ for some β fresh. For each edge in E corresponds an α, α'-dependency in $\mathcal{C}$ and vice-versa.*

Proof. $\Rightarrow$) The proof is by induction on the recursive structure of a run of $\texttt{INF}(P, \Gamma, \Theta)$ with case analysis on the last applied rule. We sketch service accept case when $\dot{P} = v(\mathbf{x}).P$ and $\dot{\mathcal{C}} = \mathcal{C} \cup \{\alpha \leq T\} \cup \{\Gamma(v) = [\alpha]\}$. The unification constraint unifies the type of α with every service accept and service request relative to v. Applying the resolution steps of `solve`, $\alpha \leq T$ is turned to a constraint of the form $\alpha' \leq U$ where U is the type of $\mathbf{x}$ and the session type variable α' is relative to the service w s.t. $\mathsf{unified}(v) = w$. In particular an α', α''-dependency is created if the service associated with α'' is either sent or received through $\mathbf{x}$ and an α''', α'-dependency is created if w is either received or sent through the service associated with α'''. We can conclude by induction since the proof mimes the construction of $G_{v(\mathbf{x}).P}$ relative to v.
$\Leftarrow$) The proof directly follow by the construction of G_P and the definition of α, α'-dependency. □

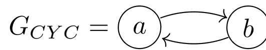
Corollary 5.34. *Let P a process. If G_P is acyclic and there exist $\mathcal{C}, \Delta$ s.t. $(\mathcal{C}, \Delta) = \texttt{INF}(P, \emptyset, \emptyset)$ then $\mathcal{C}$ does not contain non-admitted dependencies.*

Proof. The proof follows directly applying Lemma 5.33. □

Let us now show some examples of the graph construction.

Example 5.35. In this example we show a process CYC and its graph G_{CYC} which has a cycle $(a, b), (b, a)$.

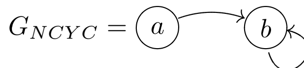
$$CYC = a(\mathbf{x}).\mathbf{x}!(b) \mid \overline{b}(\mathbf{y}).\mathbf{y}?(x) \mid \overline{a}(\mathbf{x}).\mathbf{x}?(y).y(\mathbf{y}).\mathbf{y}!(a)$$



The cycle is relative to the non-admitted dependency $\overline{?([\alpha_b])} \leq \alpha_a, \alpha_b \leq\ !([\alpha_a])$ (we simplified the unification constraints where y has been unified with b) generated by $\texttt{INF}(CYC, \emptyset, \emptyset)$.

Example 5.36. Consider the process

$$NCYC = a(\mathbf{x}).\mathbf{x}!(b) \mid \overline{a}(\mathbf{x}).\mathbf{x}?(x).\overline{x}(\mathbf{y}).\mathbf{y}!(b)$$



Since G_{NCYC} has a cycle by Lemma 5.33, $\texttt{INF}(NCYC, \emptyset, \emptyset) = (\mathcal{C}, \emptyset)$ generates non-admitted dependencies.

```
1  let rec solve c =
2    let dis = solveunification(c) in
3      while (!zero <>[]) do
4        while (!alpha <>[]) do
5          alpha:= get_alpha diseq;
6            (match !alpha
7               with Subtype(Vart(id), _):: l ->
8                 lalpha:= find_alpha id diseq;
9                 diseq:=remove_list diseq lalpha
10                lc:=  find_free_alpha id diseq;
11                lc1:= find_free_alpha1 id diseq;
12                lnc:= find_free_not_alpha id diseq;
13                if (!lc=[] and !lc1=[] and !lnc=[])
14                then
15                  let sub=localsolve lalpha in
16                  diseq:= apply_sub sub (remove_list diseq lalpha)
17                else
18                  res:= create_i_matrix !lc !lalpha;
19                  res1:=create_i_matrix !lc1 !lalpha;
20                  res2:=create_j_matrix !lnc !lalpha;
21                  diseq:=(remove_list diseq (lc1@lc@lnc))@res@res1@res2;
22            | _ -> ());
23         done;
24         zero:= find_non_constr_var diseq
25         (match !zero with
26           | Dsubtype(t1, t2):: l -> let sub = subtype(dual t1, t2) in
27            diseq:= apply_sub sub (remove_list diseq [Dsubtype(t1, t2)])
28           | _ -> ());
29           alpha:=[Subtype(Endt, Endt)];
30     done;
```

Figure 36: An implementation of the solve algorithm

5.5.1 Solve implementation

The `solve` algorithm is given by pattern matching on the set of constraints in input. We now briefly discuss a possible implementation used in TypSes which we believe better clarify how `solve` works in practice. The code of the algorithm is reported in Figure 36. For an elegant implementation we use the imperative features of Ocaml. Apart from the line 2 which solves the unification constraints, the main code is composed of two nested while. The internal while, lines 4-22, simulates the cases 3 and 4 of `solve` and it is applied as many time as possible. In line 5 we find a constraint of the form $\alpha \leq T$ such that α does not appear elsewhere as the argument of either an input or an output action. If at least one of such constraint is found we extract `id` the identifier of type variable in the constraint (line 7). With the help of `id` we collect four lists: `lalpha` which is a list of all the constraints of the form $\texttt{id} \leq T_i$, `lc` which is a list

of all constraints of the form $T'_i \leq \mathbb{C}[\![\mathtt{id}]\!]$, `lc1` which is a list of all constraints of the form $\overline{T''_j} \leq \mathbb{C}'[\![\mathtt{id}]\!]$ and `lnc` which is a list of all constraints of the form $\overline{\mathbb{C}''[\![\mathtt{id}]\!]} \leq T'''_k$. Notice in case 3 of `solve` we do not have the analogous of the list `lc1` since we handle this case imposing the condition "each T'_i can be optionally overlined". These four fresh lists `lalpha`, `lc`, `lc1` and `lnc` are used to create the sets I and J of the `solve` which are called `res`, `res1` (for I) and `res2` (for J) here. If I and J are empty we use the `localsolve` function (lines 15-16). Just before exiting the internal while (line 21) we update the set of constraints (`diseq`) adding the newly created ones and removing the existing ones. In lines 24-28 we solve a subtyping constraint that fits the premise of case 2 of `solve`, in order to obtain the syntactic unifier (`sub`) computed by means of the function `subtype`. We have two types of possible subtype constraints depending if the first member is or not overlined, indicated respectively with `Subtype` and `Dsubtype`. Thus in line 26 we handle the `Dsubtype` constraints (the case 2 of `solve`) using the function `dual` to compute the dual of a session type. In few words we remove the overline from a constraint effectively computing its dual without caring about free variables. Exit from the internal while is guaranteed because in line 9 we remove `lalpha` from `diseq` while exit from the external while is guaranteed because we remove the `Dsubtype` constraint from `diseq`. For sake of simplicity we omit in Figure 36, line 5 of `solve` since its implementation is simple and should be placed after line 29.

5.6 Encodings

In this section we introduce two encodings: from the standard π-calculus to HVK-X and from CST to HVK-X. In order to prove the validity of our encodings we proceed differently from the standard way (35; 62). We do prove the validity of an encoding showing that each encoded process is typable in HVK-X if and only if it is typable in the source calculus.

$$\frac{}{\Gamma, x : \boldsymbol{\pi} \vdash x : \boldsymbol{\pi}}\ (\text{PI-VAR}) \qquad \frac{}{\Gamma \vdash \mathbf{0}}\ (\text{PI-NIL}) \qquad \frac{\Gamma, \tilde{y} : \tilde{\boldsymbol{\pi}} \vdash \mathbf{P} \quad \Gamma \vdash v : [\tilde{\boldsymbol{\pi}}]}{\Gamma \vdash v(\tilde{y}).\mathbf{P}}\ (\text{PI-IN})$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash v : [\tilde{\boldsymbol{\pi}}] \quad \Gamma \vdash \tilde{w} : \tilde{\boldsymbol{\pi}}}{\Gamma \vdash v\langle\tilde{w}\rangle.\mathbf{P}}\ (\text{PI-OUT}) \qquad \frac{}{\Gamma, a : \boldsymbol{\pi} \vdash a : \boldsymbol{\pi}}\ (\text{PI-CHAN}) \qquad \frac{\Gamma \vdash \mathbf{P}\ \Gamma \vdash \mathbf{Q}}{\Gamma \vdash \mathbf{P}|\mathbf{Q}}\ (\text{PI-PAR})$$

$$\frac{\Gamma, a : \boldsymbol{\pi} \vdash P}{\Gamma \vdash (\nu a)P}\ (\text{PI-REST}) \qquad \frac{}{\Gamma \vdash X}\ (\text{PI-PVAR}) \qquad \frac{\Gamma \vdash \mathbf{P}}{\Gamma \vdash \texttt{rec}\ X.\mathbf{P}}\ (\text{PI-REC})$$

Figure 37: Simply typed π-calculus

$$\begin{array}{l} enc(\mathbf{0}) = \mathbf{0} \qquad enc(v(\tilde{y}).\mathbf{P}) = v(\mathbf{x}).\mathbf{x}?(\tilde{y}).enc(\mathbf{P}) \\ enc(v\langle\tilde{w}\rangle.\mathbf{P}) = \overline{v}(\mathbf{x}).\mathbf{x}!(\tilde{w}).enc(\mathbf{P}) \\ enc(\mathbf{P}|\mathbf{Q}) = enc(\mathbf{P})|enc(\mathbf{Q}) \qquad enc((\nu a)\mathbf{P}) = (\nu a)enc(\mathbf{P}) \\ enc(X) = X \qquad enc(\texttt{rec}\ X.\mathbf{P}) = \texttt{rec}\ X.enc(\mathbf{P}) \end{array}$$

Figure 38: Encoding of the π-calculus processes

$$enc([]) = [?(\texttt{end})] \qquad enc([\boldsymbol{\pi}_1, \dots, \boldsymbol{\pi}_n]) = [?(enc(\boldsymbol{\pi}_1), \dots, enc(\boldsymbol{\pi}_n))]$$

Figure 39: Encoding of the π-calculus types

5.6.1 Encoding the π-calculus

We report the syntax of a π-calculus process $\mathbf{P}$ and the syntax of a π-calculus simply types π.

$$\begin{array}{l} \mathbf{P}, \mathbf{Q} ::= \mathbf{0} \mid v(\tilde{y}).\mathbf{P} \mid v\langle\tilde{w}\rangle.\mathbf{P} \mid \mathbf{P}|\mathbf{Q} \mid (\nu a)\mathbf{P} \mid X \mid \texttt{rec}\ X.\mathbf{P} \\ \boldsymbol{\pi} ::= [] \mid [\tilde{\boldsymbol{\pi}}] \end{array}$$

Syntax is standard: we only use recursion instead of replication and the distinction between variables and channels ranged over by $x, y, \dots$ and by $a, b, \dots$ respectively, instead of using a unique syntactic category for names. Values which are both channel and variables are ranged over by $v, w, \dots$. Types $\boldsymbol{\pi}$ can be a channel in which one can output a null tuple or a channel in which one can output a tuple of type $\tilde{\boldsymbol{\pi}}$. In the simply typed π-calculus each channel can be used to send and receive values of a unique type. Typing rules of the simply typed π-calculus are reported in Figure 37. Typing judgments are of the form $\Gamma \vdash \mathbf{P}$ (we use an overload-

ing of Γ without risk of clash). The encoding function enc from π-calculus processes to HVK-X processes is shown in Figure 38. The only non-trivial cases are those relative to channel input, which is modeled as a service accept that inputs a tuple, and to channel output which is modeled as a service request which outputs a tuple. Since types have a different syntax we also encode them in Figure 39. Remember that in HVK-X type discipline we assume only the type of the service accept so the null tuple channel is encoded as a service that expects to *input* an ended session and the same for $[\tilde{\boldsymbol{\pi}}]$ which is encoded as a service that reads a tuple.

The following theorem proves the soundness of enc but one can think of the theorem as the fact that the session type discipline used in a simple request-response manner has the same power as the simply typed π-calculus.

Theorem 5.37 (Soundness and Completeness of $enc(\mathbf{P})$)**.** $\Gamma \vdash \mathbf{P}$ *iff* $enc(\Gamma); \Theta \vdash enc(\mathbf{P}) \triangleright \emptyset$ *for some* Θ *s.t.* $X \in \mathsf{fpv}(P)$ *implies* $\Theta = \Theta', X : \emptyset$*.*

Proof. $\Rightarrow$) The proof is by induction on the typing derivation of $\Gamma \vdash \mathbf{P}$ with case analysis on the last applied rule. Base cases are when the rules (PI-NIL) and (PI-PVAR) are applied and they are immediate to check. In the inductive cases, when the last applied rule is:

- (PI-IN) and $\dot{P} = v(\tilde{y}).\mathbf{P}$, and $\dfrac{\Gamma, \tilde{y} : \tilde{\boldsymbol{\pi}} \vdash \mathbf{P} \quad \Gamma \vdash v : [\tilde{\boldsymbol{\pi}}]}{\Gamma \vdash v(\tilde{y}).\mathbf{P}}$ (PI-IN), by induction $enc(\Gamma), \tilde{y} : enc(\tilde{\boldsymbol{\pi}}); \Theta \vdash enc(P) \triangleright \emptyset$ for some suitable Θ which allows us to conclude by

$$\dfrac{\dfrac{\dfrac{enc(\Gamma), \tilde{y} : enc(\tilde{\boldsymbol{\pi}}); \Theta \vdash enc(P) \triangleright \emptyset}{enc(\Gamma), \tilde{y} : enc(\tilde{\boldsymbol{\pi}}); \Theta \vdash enc(P) \triangleright \mathbf{x} : \mathtt{end}}\ (\textsc{SWeak})}{enc(\Gamma); \Theta \vdash \mathbf{x}?(\tilde{y}).enc(P) \triangleright \mathbf{x} : ?(enc(\tilde{\boldsymbol{\pi}})) \quad enc(\Gamma) \vdash v : [?(enc(\tilde{\boldsymbol{\pi}}))]}\ (\textsc{SIn})}{enc(\Gamma); \Theta \vdash v(\mathbf{x}).\mathbf{x}?(\tilde{y}).enc(P) \triangleright \emptyset}\ (\textsc{SAcc})$$

- (PI-OUT) and $\dot{P} = v\langle\tilde{w}\rangle.\mathbf{P}$ and $\dfrac{\Gamma \vdash P \quad \Gamma \vdash v : [\tilde{\boldsymbol{\pi}}] \quad \Gamma \vdash \tilde{w} : \tilde{\boldsymbol{\pi}}}{\Gamma \vdash v\langle\tilde{w}\rangle.\mathbf{P}}$ (PI-OUT), by induction we have $enc(\Gamma); \Theta \vdash enc(P) \triangleright \emptyset$ for some suitable Θ which allows us to conclude by:

$$\dfrac{\dfrac{\dfrac{enc(\Gamma); \Theta \vdash enc(P) \triangleright \emptyset}{enc(\Gamma); \Theta \vdash enc(P) \triangleright \mathbf{x} : \mathtt{end}}\ (\textsc{SWeak}) \quad enc(\Gamma) \vdash \tilde{w} : enc(\tilde{\boldsymbol{\pi}})}{enc(\Gamma); \Theta \vdash \mathbf{x}!(\tilde{w}).enc(P) \triangleright \mathbf{x} : \overline{!(enc(\tilde{\boldsymbol{\pi}}))} \quad enc(\Gamma) \vdash v : [?(enc(\tilde{\boldsymbol{\pi}}))]}\ (\textsc{SOut})}{enc(\Gamma); \Theta \vdash \overline{v}(\mathbf{x}).\mathbf{x}!(\tilde{w}).enc(P) \triangleright \emptyset}\ (\textsc{SReq})$$

- (PI-REC) and $\dot{P} = \texttt{rec}\ X.\mathbf{P}$ and $\dfrac{\Gamma \vdash \mathbf{P}}{\Gamma \vdash \texttt{rec}\ X.\mathbf{P}}$ (PI-REC), by induction we have that $enc(\Gamma); \Theta, X : \emptyset \vdash enc(P) \triangleright \emptyset$ for some suitable Θ then we can directly conclude with rule (SREC) $\Gamma; \Theta \vdash \texttt{rec}\ X.enc(P) \triangleright \emptyset$ since $\mathsf{fpn}(P) = \emptyset$, by Lemma 5.8.

- (PI-PAR), (PI-REST) the result follows directly by induction.

$\Leftarrow$)The proof is by induction on $enc(\mathbf{P})$ with case analysis on the last applied rule. Base cases: when $P = X$ and $enc(\Gamma); \Theta, X : \emptyset \vdash X \triangleright \emptyset$ implies $\Gamma \vdash X$ and when $P = \mathbf{0}$ and $enc(\Gamma); \Theta \vdash \mathbf{0} \triangleright \emptyset$ implies $\Gamma \vdash \mathbf{0}$. In the inductive case when the last applied rule is:

- $enc(v(\tilde{y}).\mathbf{P})$ is similar to the case for (PI-IN) with inverse implications.

- $enc(v\langle\tilde{w}\rangle.\mathbf{P})$ is similar to the case for (PI-OUT) with inverse implications.

- $enc(P|Q)$, $enc((\nu a)\mathbf{P})$ follow directly by induction.

- $enc(\texttt{rec}\ X.\mathbf{P})$ and $\dfrac{enc(\Gamma); \Theta, X : \emptyset \vdash enc(\mathbf{P}) \triangleright \emptyset}{enc(\Gamma); \Theta \vdash \texttt{rec}\ X.enc(\mathbf{P}) \triangleright \emptyset}$ (SREC) and by induction $\Gamma \vdash \mathbf{P}$ and we can conclude with rule (PI-REC).

□

Example 5.38. The ping server takes a channel and sends a null tuple in response to advise the client that it still correctly working. We show how its encoding by means of enc is typed in HVK-X. The π-calculus process relative to the ping server is the following $P = \texttt{rec}\ X.(ping(r).r\langle\rangle|X) \mid (\nu r_1)ping(r_1).r_1()$ and it can be typed using the rule if Figure 37 as $ping : [[]] \vdash P$. We now consider its encoding $enc(P) = \texttt{rec}\ X.(ping(\mathbf{x}).\mathbf{x}?(r).\overline{r}(\mathbf{x}).\mathbf{x}!() \mid X)(\nu r_1)\overline{ping}(\mathbf{x}).\mathbf{x}!(r_1).r_1(\mathbf{x}).\mathbf{x}?()$ which is a well-typed HVK-X process $ping : [?([?()])]; \emptyset \vdash enc(P) \triangleright \emptyset$. In particular the following is a fragment of the proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{ping : [?([?()])], r : [?()]; X : \emptyset \vdash \mathbf{0} \triangleright \emptyset}{ping : [?([?()])], r : [?()]; X : \emptyset \vdash \mathbf{0} \triangleright \mathbf{x} : \texttt{end}}\ (\text{SWEAK})}{ping : [?([?()])], r : [?()]; X : \emptyset \vdash \mathbf{x}!() \triangleright \mathbf{x} :!()}\ (\text{SOUT})}{ping : [?([?()])], r : [?()]; X : \emptyset \vdash \overline{r}(\mathbf{x}).\mathbf{x}!() \triangleright \emptyset}\ (\text{SREQ})}{ping : [?([?()])], r : [?()]; X : \emptyset \vdash \overline{r}(\mathbf{x}).\mathbf{x}!() \triangleright \mathbf{x} : \texttt{end}}\ (\text{SWEAK})}{ping : [?([?()])]; X : \emptyset \vdash \mathbf{x}?(r).\overline{r}(\mathbf{x}).\mathbf{x}!() \triangleright \mathbf{x} :?([?()])}\ (\text{SIN})}{ping : [?([?()])]; X : \emptyset \vdash ping(\mathbf{x}).\mathbf{x}?(r).\overline{r}(\mathbf{x}).\mathbf{x}!() \triangleright \emptyset}\ (\text{SACC})$$

5.6.2 Encoding CST

We now try to encode CST in HVK-X proving the the correspondence between type system defined in Figure 16 and type system defined in Figure 31. This time the encoding is more tricky, due to the different session management policies taken in each calculus.

The first problem arises with recursion. Consider for example the CST process $\mathtt{rec}\ X.\langle 5\rangle.a.\langle 5\rangle.X$ of infinite nested declarations of a which outputs 5 two times once invoked. Notice that the process is well-typed assuming $a : [!(int).!(int)]$. The obvious encoding $\mathtt{rec}\ X.\kappa!(5).a(\mathbf{x}).\mathbf{x}!(5).X$ does not work since after the unfolding of the recursion we have $\langle 5\rangle.a.\langle 5\rangle.\mathtt{rec}\ X.\langle 5\rangle.a.\langle 5\rangle.X$ from one side and $\kappa!(5).a(\mathbf{x}).\mathbf{x}!(5).\mathtt{rec}\ X.\kappa!(5).a(\mathbf{x}).\mathbf{x}!(5).X$ from the other side which are two very different processes: in the former case only two outputs are performed in the current session while in the latter case infinite outputs are performed on κ. The solution to the problem is using a process definition with at most two parameters (the current session c and the parent session p) for each $\mathtt{rec}$ instruction. Thus the correct encoding for the previous process is $X(\kappa) \stackrel{\text{def}}{=} c!(5).a(\mathbf{x}).\mathbf{x}!(5).X(\mathbf{x})\ \mathtt{in}\ X(c)$ which uses a process definition that takes only the current session c. The parent session is ignored since no actions are performed and no assumptions about processes with ended session are allowed.

The encoding of process definitions starting from the general recursion (that we already have) is standard (for instance see (60) Section 9.5) and the technique uses the replication together with a new fresh name in place of each recursive call, for example the definition of $X(c)$ above is modeled as:

$$\mathtt{rec}\ Y.a_X(\mathbf{y}).\mathbf{y}?(\!(c)\!).a(\mathbf{x}).\mathbf{x}!(5).\overline{a_X}(\mathbf{y}_1).\mathbf{y}_1!\langle\!\langle \mathbf{x}\rangle\!\rangle.\mathbf{0} \mid Y$$

in which we create a new replicated service a_X used to catch the current c and after writing the integer 5 on both the current session and the new created session $\mathbf{x}$, it recursively invokes a_X throwing $\mathbf{x}$ as parameter, i.e. $\mathbf{x}$ will become the new parent session. It is simple to see that the process is typed with $\Gamma = a_X : [?(!(int))], a : [!(int).!(int)]$ and $\Theta = Y : \emptyset$.

For simplicity however we take (only for this subsection) as built in process definitions of the form $X(\tilde{\kappa}) \stackrel{\text{def}}{=} P\ \mathtt{in}\ Q$ in which the declaration of X with the list of parameters $\tilde{\kappa}$ has Q as its scope. (More generally, process definitions in (76) allows passing also variables besides sessions which could be easily accommodated here as well.) We present the new

typing rules for process definitions:

$$\frac{\Gamma; \Theta, X : \tilde{T} \vdash P \triangleright \tilde{\kappa} : \tilde{T} \quad \Gamma; \Theta, X : \tilde{T} \vdash Q \triangleright \Delta}{\Gamma; \Theta \vdash X(\tilde{\kappa}) \stackrel{\text{def}}{=} P \text{ in } Q \triangleright \Delta} \text{(SDEF)} \qquad \Gamma; \Theta, X : \tilde{T} \vdash X(\tilde{\kappa}) \triangleright \tilde{\kappa} : \tilde{T} \text{ (SDEFVAR)}$$

These rules are the same as in (76) where differently from our type system assumptions on process variables are only a list of types since the relative subjects will be extracted from the list of parameters. Interestingly the problem due to unguarded recursion returns also here e.g. $\texttt{rec}\ X(\kappa_1).X(\kappa_1)$ allows a linear environment of the form $\kappa_1 : T$ where T can be any session types. We assume the consistency checks also here, so for example when we encode the CST process $\texttt{rec}\ X.X$ (which is typed with end) as $X(\kappa_1, \kappa_2) \stackrel{\text{def}}{=} X(\kappa_1, \kappa_2)\ \texttt{in}\ X(c, p)$ we assume X is typed also with two ends. For the sake of completeness we report also the syntax directed version of the rules

$$\frac{\Gamma; \Theta, X : \tilde{T} \vdash_{\mathsf{sd}} P \triangleright \tilde{k} : \tilde{T}' \quad \tilde{\kappa} : \tilde{T} \leq \tilde{\kappa} : \tilde{T}' \quad \Gamma; \Theta, X : \tilde{T} \vdash_{\mathsf{sd}} Q \triangleright \Delta}{\Gamma; \Theta \vdash_{\mathsf{sd}} X(\tilde{\kappa}) \stackrel{\text{def}}{=} P \text{ in } Q \triangleright \Delta} \text{(SDEFSD)} \qquad \Gamma; \Theta, X : \tilde{T} \vdash_{\mathsf{sd}} X(\tilde{\kappa}) \triangleright \tilde{\kappa} : \tilde{T} \text{ (SDEFVARSD)}$$

and the code for constraint extraction:

```
INF(X(κ1,...,κn) =def P in Q, Γ, Θ) =
      in let (C, κ1 : T1,...,κn : Tn) =
                        INF(P, Γ, Θ ∪ {X : αX1,...,αXn})
      in let (C1, Δ) = INF(Q, Γ, Θ ∪ {X : αX1,...,αXn})
      in (C ∪ C1 ∪ {αX1 ≤ T1,...,αXn ≤ Tn}, Δ)
INF(X, Γ, Θ) = (∅, κ1 : T1,...,κn : Tn) where Θ(X) = T1,...,Tn
```

Since the number of parameters in a process definition can vary from 0 to 2 and strictly depend of the opened sessions we need a further environment to store assumptions about the number of parameters required to call a certain process definition. The *param-environment* Λ is defined by the following grammar:

$$\Lambda ::= \Lambda, X : 0 \mid \Lambda, X : 1 \mid \Lambda, X : 2 \mid \emptyset$$

then the usual operation of $\Lambda, X : n$ and the definition of $\mathsf{dom}(\Lambda)$ are standard. An assumption of the form $X : n$ indicates we expect the process definition X to take n parameters with $0 \leq n \leq 2$.

We present our encoding in Figure 40. Without risk of confusion we use P, Q to refer both CST and HVK-X processes: it will be the context to

$$
\begin{array}{lcl}
enc(\mathbf{0}, c, p, \Lambda) & = & \mathbf{0} \\
enc(v.P, c, p, \Lambda) & = & v(\mathbf{x}).enc(P, \mathbf{x}, c, \Lambda) \quad \mathbf{x} \text{ fresh} \\
enc(\overline{v}.P, c, p, \Lambda) & = & \overline{v}(\mathbf{x}).enc(P, \mathbf{x}, c, \Lambda) \quad \mathbf{x} \text{ fresh} \\
enc(r^q \rhd P, c, p, \Lambda) & = & enc(P, r^p, c, \Lambda) \\
enc(\begin{array}{l}\texttt{if } v = w \texttt{ then} \\ P \texttt{ else } Q\end{array}, c, p, \Lambda) & = & \begin{array}{l}\texttt{if } v = w \texttt{ then} \\ enc(P, c, p, \Lambda) \texttt{ else } enc(Q, c, p, \Lambda)\end{array} \\
enc((\tilde{x}).P, c, p, \Lambda) & = & c?(\tilde{x}).enc(P, c, p, \Lambda) \quad c \neq \epsilon \\
enc(\Sigma_{i=1}^{n}(l_i).P_i, c, p, \Lambda) & = & \Sigma_{i=1}^{n} c?(l_i).enc(P_i, c, p, \Lambda) \quad c \neq \epsilon \\
enc(\langle \tilde{v} \rangle.P, c, p, \Lambda) & = & c!(\tilde{v}).enc(P, c, p, \Lambda) \quad c \neq \epsilon \\
enc(\langle l \rangle.P, c, p, \Lambda) & = & c!(l).enc(P, c, p, \Lambda) \quad c \neq \epsilon \\
enc(\texttt{return } \tilde{v}.P, c, p, \Lambda) & = & p!(\tilde{v}).enc(P, c, p, \Lambda) \quad p \neq \epsilon \\
enc(P|Q, c, p, \Lambda) & = & \left\{\begin{array}{ll} enc(P, c, p, \Lambda)|enc(Q, \epsilon, \epsilon, \Lambda) & \text{if } noactions(Q) \\ enc(P, \epsilon, \epsilon, \Lambda)|enc(Q, c, p, \Lambda) & \text{if } noactions(P) \end{array}\right. \\
enc(P > \tilde{x} > Q, c, p, \Lambda) & = & \left\{\begin{array}{ll} enc(P, c, p, \Lambda) & \text{if } noactions(P) \\ (\nu r)\left(\begin{array}{l} enc(P, r^+, \epsilon, \emptyset)| \\ r^-?(\tilde{x}).enc(Q, c, p, \Lambda) \end{array}\right) & \begin{array}{l}\text{if } noretact(P) \text{ and} \\ \mathsf{fpv}(P) = \emptyset \text{ and} \\ r \text{ fresh and} \\ \mathsf{fpn}(Q) = \emptyset \end{array} \end{array}\right. \\
enc((\nu m)P, c, p, \Lambda) & = & (\nu m)enc(P, c, p, \Lambda) \\
enc(\texttt{rec } X.P, c, p, \Lambda) & = & \left\{\begin{array}{lcl} X & \stackrel{\text{def}}{=} & \begin{array}{l} enc(P, \epsilon, \epsilon, \Lambda, X:0) \texttt{ in } X \\ \text{if } noactions(P) \end{array} \\ X(\kappa) & \stackrel{\text{def}}{=} & \begin{array}{l} enc(P, \kappa, \epsilon, \Lambda, X:1) \texttt{ in } X(c) \\ \text{if } noretact(P) \end{array} \\ X(\kappa_1, \kappa_2) & \stackrel{\text{def}}{=} & \begin{array}{l} enc(P, \kappa_1, \kappa_2, \Lambda, X:2) \texttt{ in } X(c, p) \\ otherwise \end{array} \end{array}\right. \\
enc(X, c, p, \Lambda, X:2) & = & X(c, p) \\
enc(X, c, p, \Lambda, X:1) & = & X(c) \\
enc(X, c, p, \Lambda, X:0) & = & X
\end{array}
$$

Figure 40: Encoding CST in HVK-X

disambiguate their meaning from time to time. Also we use c, p to range over both session variables and a special mark ϵ that stays indicating no communication operations are possible on a certain session. enc takes a CST process and two session variables (possibly ϵ) that represent both the current and the parent session, a param-environment Λ with assumptions about the number of parameters in a process definition and returns an HVK-X process which (as we shall prove) is well typed if and only if the original process is well typed.

Some comments about the encoding function follow. The function $noactions(P)$ holds if both $nocuract(P)$ and $noretact(P)$ hold which are the same predicates defined in Figure 17. In the case of service definitions, $enc(v.P, c, p)$ creates a new service accept and a new session variable $\mathbf{x}$, used as current session, in the encoding of the body P. The encoding of a service invocation is similar, while the encoding of a session end

$(r^q \triangleright P)$ simply uses r^q as the current session and c as the new parent session. Communication primitives are encoded as communications on the current session c while a return instruction corresponds to an output in the parent session p. The encoding of $P|Q$ checks which subprocess between P or Q does not perform actions. On one hand two ϵ's are used to encode the subprocess with no actions. On the other hand the good sessions c, p are used to encode the process which performs some actions. The encoding of a pipe has two subcases depending of the function pipe, in the first subcase when P does not perform any actions then we just encode P with the correct sessions c, p. In the second case of the pipe when P outputs a tuple we create a new fresh session r: r^+ is used to encode P so as to receive the tuple while a reading on r^- blocks the execution of Q until a tuple is received. Notice that in both the encoding of a parallel composition and of a pipe we use the same syntactic restrictions used in the relative typing rules of CST. Hence if enc fails due to the premises of rules for parallel composition and pipe the process is not well-typed.

The encoding of a process definitions adds an assumption in the param-environment based on the predicate $noactions(P)$ and $noretact(P)$ and the encoding of a process variable uses these assumptions in order to build the correct function call. Following lemmas are the equivalent of the Weakening Lemma and of the Strengthening Lemma, used to add and remove assumptions from Λ in the encoding.

Lemma 5.39. *If* $enc(P, c, p, \Lambda) = Q$ *and* $X \notin \mathsf{fpv}(P)$ *then* $enc(P, c, p,\ \Lambda, X : n) = Q$. *Vice versa if* $enc(P, c, p,\ \Lambda, X : n) = Q$ *and* $X \notin \mathsf{fpv}(P)$ *then* $enc(P, c, p, \Lambda) = Q$.

Proof. By straightforward induction on the definitions of $enc(P, c, p, \Lambda) = Q$ and $enc(P, c, p,\ \Lambda, X : n) = Q$ respectively. □

The following is an example of the encoding of a recursive process.

Example 5.40. Consider the process $NEST = \texttt{rec}\, X.a.\langle 5\rangle.X$ and $NEST' = \texttt{rec}\, X.a.\langle 5\rangle.\texttt{return}\ 5.X$ then $enc(NEST, c, p, \emptyset) = X \stackrel{\text{def}}{=} a(\mathbf{x}).\mathbf{x}!(5).X\ \texttt{in}\ X$ and $enc(NEST', c, p, \emptyset) = X(\kappa) \stackrel{\text{def}}{=} a(\mathbf{x}).\mathbf{x}!(5).\kappa!(5).X(\mathbf{x})\ \texttt{in}\ X(c)$.

In the encoding we use functions $noretact$ and $noactions$ to statically guess if the process has type $\texttt{end}$. Unfortunately this reasoning works only in the following direction:

Lemma 5.41. *Let* $\Gamma; \Theta \vdash P : T; U; L$. *If* $U = \texttt{end}$ *then* $noretact(P)$.

Proof. The proof is by induction on the typing derivation of $\Gamma; \Theta \vdash P : T; U; L$ with case analysis on the last applied rule. We prove the inductive case when the last applied rule is (TPARL). By hypothesis $\Gamma; \Theta \vdash P|Q : T; U; L_1 \uplus L_2$ and $\Gamma; \Theta \vdash P : T; U; L_1$ and $\Gamma; \Theta \vdash Q : \texttt{end}; \texttt{end}; L_2$ and by induction $noretact(Q)$ and if $U = \texttt{end}$ then $noretact(P)$ which concludes. $\square$

Lemma 5.42. *Let $\Gamma; \Theta \vdash P : T; U; L$. If $T = \texttt{end}$ and $U = \texttt{end}$ then $noactions(P)$.*

Proof. The proof is by induction on the typing derivation of $\Gamma; \Theta \vdash P : T; U; L$ with case analysis on the last applied rule. We prove the inductive case when the last applied rule is (TPIPE). By hypothesis $\Gamma; \Theta \vdash P > \tilde{x} > Q : T; U; L$ and $\Gamma; \Theta \vdash Q : T_1; U_1; \emptyset$ for some T_1 and U_1 by induction if $T_1 = U_1 = \texttt{end}$ then $nocuract(Q)$ and $noretact(Q)$ which concludes since $(T, U) = (\texttt{end}, \texttt{end}) = \mathsf{pipe}(T_2, \texttt{end}, \texttt{end}, \tilde{S})$ for any T_2 and $\tilde{S}$. $\square$

The converse of the previous two lemmas does not hold, this means that we cannot statically guess when the type of a process (especially an open one) is `end`. As a simple counterexample consider the process $a : [\texttt{end}]; X : \texttt{end}; !(int) \vdash a.X : \texttt{end}; !(int); \emptyset$ in which $noactions(a.X)$ holds but the process has a type different from `end`; `end`.

Think of to the `INF` algorithm in Figure 25 where we used the `or` keyword to overcome this limitation. Here we use another solution instead of trying, we assume the following hypotheses.

Hypothesis 1. Let $\Gamma; \Theta \vdash P : T; U; L$. If $noretact(P)$ then $U = \texttt{end}$.

Hypothesis 2. Let $\Gamma; \Theta \vdash P : T; U; L$. If $noactions(P)$ then $T = \texttt{end}$ and $U = \texttt{end}$.

Notice that previous hypotheses are Lemmas for closed processes which trivially hold due to the consistency checks in the rule for recursion.

Lemma 5.43. *Let $\Gamma; \emptyset \vdash P : T; U; L$. If $noretact(P)$ then $U = \texttt{end}$.*

Lemma 5.44. *Let $\Gamma; \emptyset \vdash P : T; U; L$. If $noactions(P)$ then $T = \texttt{end}$ and $U = \texttt{end}$.*

We believe that these hypotheses are not an issue since if a process does not perform observable actions then it should be typed in a consistent manner. Assuming these two hypotheses we prove the soundness and completeness of our encoding. We now define a mapping from

CST environments to HVK-X environments used in the proof of Theorem 5.45. Given a CST standard typing environment Γ, we indicate with $\breve{\Gamma}$ the same standard typing environment without assumptions on session names:

$$\breve{\Gamma} = \begin{cases} \breve{\Gamma}(a) & = & \Gamma(a) \text{ if } a \in \mathsf{dom}(\Gamma) \\ \breve{\Gamma}(x) & = & \Gamma(x) \text{ if } x \in \mathsf{dom}(\Gamma) \\ \breve{\Gamma}(r^p) & & undefined \end{cases}$$

Given a CTS process environment Θ we indicate with $\breve{\Theta}_\Lambda$ the following mapping, where we assume $\mathsf{dom}(\Lambda) = \mathsf{dom}(\Theta)$:

$$\breve{\Theta}_{\Lambda'} = \begin{cases} \breve{\Theta}_{\Lambda, X:0}(X) & = & \emptyset & \text{if} & \Theta(X) = \mathsf{end}; \mathsf{end} \\ \breve{\Theta}_{\Lambda, X:1}(X) & = & T & \text{if} & \Theta(X) = T; \mathsf{end} \\ \breve{\Theta}_{\Lambda, X:2}(X) & = & T, U & \text{if} & \Theta(X) = T; U \\ undefined & & & & \text{otherwise} \end{cases}$$

$\breve{\Gamma}$ and $\breve{\Theta}_\Lambda$ will be our standard environments for the encoded process (notice now they are valid HVK-X typing environments), removed assumptions from Γ are used together with the triple $T; U; L$ to build the linear environment for the encoded process.

Theorem 5.45 (Soundness and Completeness of *enc*)**.** *Let P a process and c, p two fresh session variables s.t. $c \notin \mathsf{fn}(P)$ and $p \notin \mathsf{fn}(P)$. $\Gamma; \Theta \vdash P : T; U; L$ iff*

1. $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, c : T, p : U$
2. *if* $U = \mathsf{end}$ *then* $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, c, \epsilon, \Lambda) \triangleright \Delta, c : T$
3. *if* $T = \mathsf{end}$ *and* $U = \mathsf{end}$ *then* $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, \epsilon, \epsilon, \Lambda) \triangleright \Delta$

where Δ is s.t. $L = \mathsf{dom}(\Delta)$ and for each $r^p \in \mathsf{dom}(\Delta)$ then $\Gamma(r^p) = [\Delta(r^p)]$.

Proof. $\Rightarrow$) The proof is by induction on the typing derivation of $\Gamma; \Theta \vdash P : T; U; L$ we prove only the first statement the others are similar. Base cases are $\frac{(\text{TNIL})}{\Gamma; \Theta \vdash \mathbf{0} : \mathsf{end}; \mathsf{end}; \emptyset}$ and using rule (SWEAK) we can conclude $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash \mathbf{0} \triangleright c : \mathsf{end}, p : \mathsf{end}$ and $\frac{(\text{TPVAR})}{\Gamma; \Theta, X : T; U \vdash X : T; U; \emptyset}$ and we can conclude with $\frac{(\text{SDEFVAR})}{\breve{\Gamma}; \breve{\Theta},\ X : T, U_{\Lambda, X:2} \vdash X(c, p) \triangleright c : T; p : U}$. In the inductive case when the last applied rule is:

- (TNEWR) and $\dot{P} = (\nu r)P$ and $\Gamma, r^+ : [T], r^- : [\overline{T}]; \Theta \vdash P : T'; U; L$ and by induction either $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, c : T', p : U; r^+ : T, r^- : \overline{T}$ if $L \cap \{r^+, r^-\} = \{r^+, r^-\}$ or $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, c : T', p : U$ if $L \cap \{r^+, r^-\} = \emptyset$. In the first case we can directly conclude applying rule (SNEWR) in the second case we can conclude after applying rule (SWEAK) to obtain $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, c : T', p : U, r^+ : \mathtt{end}, r^- : \mathtt{end}$ and then conclude by rule (SNEWR).

- (TIF) and $\dot{P} = \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ and $\Gamma; \Theta \vdash \dot{P} : T; U; L$ and by hypothesis, letting $i \in \{1, 2\}$, $\Gamma; \Theta \vdash P_i : T; U; L$ and by induction $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_i, c, p, \Lambda) \triangleright \Delta, c : T, p : U$. We can conclude by using rule (SIF) to obtain $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ enc(P_1, c, p, \Lambda)\ \mathtt{else}\ enc(P_2, c, p, \Lambda) \triangleright \Delta, c : T, p : U$.

- (TIN) and $\dot{P} = (\tilde{x}).P$ and $\Gamma; \Theta \vdash \dot{P} :?(\tilde{S}).T; U; L$ and $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash P : T; U; L$ and by induction $\breve{\Gamma}, \tilde{x} : \tilde{S}; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, c : T, p : U$ and we can conclude by an application of the rule (SIN) with $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, c :?(\tilde{S}).T, p : U$.

- (TOUT),(TRET), (TCHOICE) and (TBRANCH) similar to the previous case.

- (TPARL) and $\dot{P} = P_1 | P_2$ and $\Gamma; \Theta \vdash P_1 : T; U; L_1$ and $\Gamma; \Theta \vdash P_2 : \mathtt{end}; \mathtt{end}; L_2$ and by Lemma 5.42 $noations(P_2)$ holds. By induction $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_1, c, p, \Lambda) \triangleright \Delta_1, c : T; p : U$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_2, \epsilon, \epsilon, \Lambda) \triangleright \Delta_2$ and we can conclude with rule (SPAR) $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_1, c, p, \Lambda) | enc(P_2, \epsilon, \epsilon, \Lambda) \triangleright \Delta_1, \Delta_2, c : T, p : U$, where Δ_1, Δ_2 is defined because $L_1 \uplus L_2$ and c, p are disjoint with $\mathsf{dom}(\Delta_2)$ since their freshness.

- (TPARR) similar to the previous case.

- (TREC) and $\dot{P} = \mathtt{rec}\ X.P$ and $\Gamma; \Theta, X : T; U \vdash P : T; U; \emptyset$. If $\neg noactions(P)$ and $\neg noretact(P)$ by induction we have $\breve{\Gamma}; \breve{\Theta}, X : T, U_{\Lambda, X:2} \vdash enc(P, \kappa_1, \kappa_2, \Lambda, X : 2) \triangleright \kappa_1 : T, \kappa_2 : U$ and we can use rules (SDEF) and (SDEFVAR) to conclude $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash X(\kappa_1, \kappa_2) \stackrel{\text{def}}{=} enc(P, \kappa_1, \kappa_2, \Lambda, X : 2)\ \mathtt{in}\ X(c, p) \triangleright c : T, p : U$. The other case when $noactions(P)$ and when $noretact(P)$ hold are similar.

- (TDEF) and $\dot{P} = v.P$ and $\Gamma; \Theta \vdash P : T; U; L$ and $\Gamma \vdash v : [T]$ and by induction $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, \mathbf{x}, c, \Lambda) \triangleright \Delta, \mathbf{x} : T, c : U$ and by definition

$\breve{\Gamma} \vdash v : [T]$ and applying (SWEAK) and (SACC) we conclude $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash a(\mathbf{x}).enc(P, c, p, \Lambda) \triangleright \Delta, c : U, p : \texttt{end}$.

- (TINV) similar to the previous case
- (TSES) and $\dot{P} = r^q \triangleright P$ and $\Gamma; \Theta \vdash P : T; U; L$ and by induction $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, r^q, c, \Lambda) \triangleright \Delta, r^p : T, c : U$ and we can conclude applying (TWEAK) to add the assumption $p : \texttt{end}$ in the linear environment. (Notice that r^q becomes part of the linear environment of the conclusion since it is added to L in the conclusion of rule (TSES)).
- (TPIPE) and $\dot{P} = P > \tilde{x} > Q$ we have two cases depending of the type of P. If $\Gamma; \emptyset \vdash P :!(\tilde{S}); \texttt{end}$ then $noactions(P)$ does not hold (since it would contradict the Lemma 5.42) and $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash Q : T; U; \emptyset$ and by induction $\breve{\Gamma}; \emptyset \vdash enc(P, r^+, \epsilon, \emptyset) \triangleright \Delta_1, r^+ :!(\tilde{S})$ and $\breve{\Gamma}, \tilde{x} : \tilde{S}; \breve{\Theta}_\Lambda \vdash enc(Q, c, p, \Lambda) \triangleright c : T, p : U$. By Lemma 5.39 and the Weakening Lemma we have $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, r^+, \epsilon, \Lambda) \triangleright \Delta_1, r^+ :!(\tilde{S})$ and we can use the following derivation tree:

$$\dfrac{\dfrac{\breve{\Gamma}, \tilde{x} : \tilde{S}; \breve{\Theta}_\Lambda \vdash enc(Q, c, p, \Lambda) \triangleright c : T, p : U}{\breve{\Gamma}, \tilde{x} : \tilde{S}; \breve{\Theta}_\Lambda \vdash enc(Q, c, p, \Lambda) \triangleright c : T, p : U, r^- : \texttt{end}}}{\dfrac{\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash r^-?(\tilde{x}).enc(Q, c, p, \Lambda) \triangleright c : T, p : U, r^- :?(\tilde{S})}{\star\star}}$$

$$\dfrac{\dfrac{\overbrace{\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, r^+, \epsilon, \Lambda) \triangleright \Delta_1, r^+ :!(\tilde{S})}^{\text{Lemma 5.8 and Weakening Lemma}} \qquad \star\star}{\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, r^+, \epsilon, \Lambda) | r^-?(\tilde{x}).enc(Q, c, p, \Lambda) \triangleright \Delta_1, r^+ :!(\tilde{S}), c : T, p : U, r^- :?(\tilde{S})}}{\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash (\nu r) enc(P, r^+, \epsilon, \Lambda) | r^-?(\tilde{x}).enc(Q, c, p, \Lambda) \triangleright \Delta_1, c : T, p : U}$$

- (TWEAK) follows by induction and by rule (SWEAK).
- (TNEW) follows directly by induction.

$\Leftarrow$) The proof is by induction on the recursive structure of $enc(P, c, p, \Lambda)$ with case analysis on the last applied rule, we prove only the first statement the other are similar. In the base case relative to $\mathbf{0}$ we have $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(\mathbf{0}, c, p, \Lambda) \triangleright c : \texttt{end}, p : \texttt{end}$ and $\Gamma; \Theta \vdash \mathbf{0} : \texttt{end}; \texttt{end}; \emptyset$. In the base case relative to a process variable X we have $\breve{\Gamma}; \breve{\Theta}, X : T; U_{\Lambda, X:2} \vdash enc(X, c, p, \Lambda, X : 2) \triangleright c : T, p : U$ and $\Gamma; \Theta, X : T; U \vdash X : T; U; \emptyset$. In the inductive case when the last applied rule is:

- $enc(v.P, c, p, \Lambda)$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash v(\mathbf{x}).enc(P, \mathbf{x}, c, \Lambda) \triangleright \Delta, c : U, p : \texttt{end}$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, \mathbf{x}, c, \Lambda) \triangleright \Delta, \mathbf{x} : T, c : U, p : \texttt{end}$ and by induction $\Gamma; \Theta \vdash P : T; U; L$ and $L = \mathsf{dom}(\Delta)$ and $\Gamma \vdash v : [T]$. Applying (TDEF) we obtain $\Gamma; \Theta \vdash v.P : U; \texttt{end}; L$ which concludes.

- $enc(\overline{v}.P, p, c, \Lambda)$ similar to the previous case.

- $enc(r^q \triangleright P, c, p, \Lambda)$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(r^q \triangleright P, c, p, \Lambda) \triangleright \Delta, r^q : T, c : U, p : \texttt{end}$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, r^q, c, \Lambda) \triangleright \Delta, c : U, r^q : T$ and by induction $\Gamma; \Theta \vdash P : T; U; L$ and $\Gamma \vdash r^q : [T]$ which allows to conclude by rule (TSES) $\Gamma; \Theta \vdash r^q \triangleright P : U; \texttt{end}; L \uplus \{r^q\}$.

- $enc(\texttt{if}\ v_1 = v_2\ \texttt{then}\ P_1\ \texttt{else}\ P_2, c, p, \Lambda)$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_i, c, p, \Lambda) \triangleright \Delta, c : T, p : U$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(\texttt{if}\ v_1 = v_2\ \texttt{then}\ P_1\ \texttt{else}\ P_2, c, p, \Lambda) \triangleright \Delta, c : T, p : U$. By induction $\Gamma; \Theta \vdash P_i : T; U; L$ and we can conclude with rule (TIF).

- $enc(P_1|P_2, c, p, \Lambda)$ and we have two similar cases. Consider $noactions(P_2)$ holds, we have $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_1, c, p, \Lambda) \triangleright \Delta_1, c : T, p : U$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_2, \epsilon, \epsilon, \Lambda) \triangleright \Delta_2$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P_1|P_2, c, p, \Lambda) \triangleright \Delta_1, \Delta_2, c : T, p : U$. By induction and by Hypothesis 2 we have $\Gamma; \Theta \vdash P_1 : T; U; L_1$ and $\Gamma; \Theta \vdash P_2 : \texttt{end}; \texttt{end}; L_2$ and we can conclude with rule (TPARL).

- $enc(P > \tilde{x} > Q, c, p, \Lambda)$ and we have two cases. Consider the case $noretact(P)$. Using the Strengthening Lemma (since $\mathsf{fpv}(P) = \emptyset$) and Lemma 5.39 we obtain $\breve{\Gamma}; \emptyset \vdash enc(P, r^+, \epsilon, \emptyset) \triangleright \Delta_1, r^+ : T$. By induction and by Hypothesis 1, $\Gamma; \emptyset \vdash P : T'; \texttt{end}; L$ also $\breve{\Gamma}, \tilde{x} : \tilde{S}; \breve{\Theta}_\Lambda \vdash enc(Q, c, p, \Lambda) \triangleright \Delta_2, c : T, p : U$ implies by induction $\Gamma, \tilde{x} : \tilde{S}; \Theta \vdash Q : T; U; L_1$. Since $\mathsf{fpn}(Q) = \emptyset$ and Q is well typed by the induction hypothesis in order not to contradict Lemma 4.13 it must be $L_1 = \emptyset$ and consequently $\Delta_2 = \emptyset$. Also since by hypothesis $enc(P > \tilde{x} > Q, c, p, \Lambda)$ is well typed and r is fresh, $T' = !(\tilde{S})$. Finally since $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P > \tilde{x} > Q, c, p, \Lambda) \triangleright \Delta_1, c : T, p : U$ we can apply (TPIPE) to conclude $\Gamma; \Theta \vdash P > \tilde{x} > Q : T; U; L$. The case where $noactions(P)$ holds is simpler.

- $enc(\texttt{rec}\ X.P, c, p, \Lambda)$ and if $\neg noactions(P)$ and $\neg noretact(P)$ then $\breve{\Gamma}; \breve{\Theta}, X : T; U_{\Lambda, X:2} \vdash enc(P, \kappa_1, \kappa_2, \Lambda, X : 2) \triangleright \kappa_1 : T, \kappa_2 : U$ and by induction $\Gamma; \Theta, X : T, U \vdash P : T; U; \emptyset$ which allows to conclude by an application of the rule (TREC). The other cases when $noactions(P)$ and $noretac(P)$ are similar.

- $enc(\langle l\rangle.P, c, p, \Lambda)$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash P \triangleright \Delta, c : T, p : U$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash \langle l\rangle.P \triangleright \Delta, c : \oplus\{l_i : T_i\}_{i\in I}, p : U$ and by induction $\Gamma; \Theta \vdash P \triangleright T; U; L$ and we can conclude by an application of the rule (TCHOICE).

- $enc(\langle\tilde{v}\rangle.P, c, p, \Lambda)$, $enc(\langle(\tilde{x}).P, c, p, \Lambda)$, $enc(\Sigma_{i=1}^{n}(l_i).P_i, c, p, \Lambda)$ and $enc(\texttt{return}\ \tilde{v}.P, c, p, \Lambda)$ are similar to the previous case.

- $enc((\nu r)P, c, p, \Lambda)$ and $\breve{\Gamma}; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, r^+ : T, r^- : \overline{T}, c : T, p : U$ and by induction $\Gamma, r^+ : [T], r^- : [\overline{T}], \Theta \vdash P : T; U; L$ and we can conclude with an application of the rule (TNEWR).

- $enc((\nu a)P, c, p, \Lambda)$ and $\breve{\Gamma}, a : [T]; \breve{\Theta}_\Lambda \vdash enc(P, c, p, \Lambda) \triangleright \Delta, c : T, p : U$ and by induction $\Gamma, a : [T], \breve{\Theta} \vdash P : T; U; L$ and we can conclude with an application of the rule (TNEW).

□

Example 5.46. Consider the process $P = P_1|P_2$ where $P_1 = (x).\texttt{return}\ x$ and $P_2 = a.\langle 5\rangle$. Process P communicates in the current session and at the same time it offers a service a. Process P_1 is correctly typed with $\Gamma; \emptyset \vdash P_1 :?(int); !(int); \emptyset$ while P_2 is correctly typed with $\Gamma; \emptyset \vdash P_2 : \texttt{end}; \texttt{end}; \emptyset$ where, $\Gamma = a : [!(int)], x : int$. Now take $enc(P, c, p)$ since $noactions(P_2)$ holds $enc(P_1, c, p) = c?(x).p!(x)$ is typed with $\Gamma; \emptyset \vdash enc(P_1, c, p) \triangleright c :?(int), p :!(int)$ and $enc(P_2, \epsilon, \epsilon) = a(\mathbf{x}).\mathbf{x}!(5)$ is typed with $\Gamma; \emptyset \vdash enc(P_2, c', p') \triangleright \emptyset$. Finally $\Gamma; \emptyset \vdash P :?(int); !(int); \emptyset$ and $\Gamma; \emptyset \vdash enc(P, c, p) \triangleright c :?(int), p :!(int)$.

The previous theorem can be used to check the typability of every CST process on the base of an existing type checker for HVK-X.

5.7 Concluding remarks on HVK-X and encoding functions

In this chapter we have introduced a full session calculus with delegation (which we call HVK-X) inspired to (76), but with general recursion. We have modified the original type system adding a rule to weaken assumptions (SWEAK): both substituting a type of a session with a direct subtype and adding ended sessions. Due to these differences and the additional possibility of service extrusion we have proved the subject reduction, which does not hold directly from (76). Subsequently we have introduced a syntax directed type system, which replaces rule (SWEAK) adding

an inline subtype checking and a function, which we call linear access, to access variables in a linear environment. We have shown both the correctness and the completeness of this new syntax directed type system. Given its syntax directed nature we have developed an algorithm to extract a set of constraints which are satisfied if and only if the original process is well-typed. Finding a solution of these automatically generated constraints is not easy due to the presence of session delegation, name extrusion and cyclic recursive constraints. We have introduce our solving algorithm `solve`, that exploits transitivity and context closure of the subtyping relation to merge couple of constraints. We have shown that if `solve` succeeds then there exists a solution to the constraint sets given in input. On the other hand the completeness holds only for closed processes without free names.

Finally we have encoded both the standard π-calculus and CST in HVK-X. These encodings have been made keeping in mind that a typable process should imply the encoded process typable and vice versa. Of course this is a weak result and does not imply any relation on the operational semantics of two calculi, but it is a useful way to debug type systems as well as a way to use a unique implementation of the type checker. For example with the first encoding we have witnessed that using session communications to communicate only a value in each session, degenerate to the simply typed π-calculus. The second encoding instead shows the disciplinated nature of CST with respect to the session instantiation.

Chapter 6

Progress and comparison with related works

We prove in this Chapter the progress property of CST which turns out to be a direct consequence of the subject reduction. The progress property for HVK-X can be proved used the same technique proposed in (26).

We also give a detailed comparison with our source of inspiration and we discuss the other related works. We describe CaSPiS by means of examples, we show the particularity of the calculus proposed in (37) with the special type $\perp$ and then we show the π-calculus with sessions proposed by Gay and Hole. We also discuss the similarities with the other existent literature.

6.1 A more powerful guarantee: The Progress Property

In this section we aim to exploit our results so to prove more powerful guarantees than session safety. The first property we are interested in is called *progress* in literature (26) and it is used to check deadlock of opened sessions. In that work authors provide a type system to type-check processes with the progress property for the calculus in (76). They first introduce a type system that handles the interleaving of opened session in such a manner that no acyclic dependencies within opened ses-

sion arise. The problem of checking deadlock among opened sessions is simpler than the deadlock freedom in general due to the session linearity. To this aim it suffices to check the acyclicity of the order induced by the sequence of session subjects in a process.

The remaining problem is relative to check that each service request and each service accept actually succeeds. Since we are in a service oriented scenario this problem is faced differently and the progress theorem is stated consequently. In fact each time a process gets stuck either on a service request or on a service accept we can provide specification of the needed partner building a process using only the type information. The new partner composed in parallel with the stuck process allows the entire system to proceed.

Let us do the same with CST. We already proved a similar property in (15) but here we have the recursion operator missed in that work and we do not have an LTS to rely on, for the observation of the actions taking place in a specific session. First of all we define a relation $P \stackrel{\lambda}{\longrightarrow}$ that allows the inspection of the active actions within a certain session where λ is defined by the following grammar:

$$\lambda ::= r^p : (\tilde{x}) \mid r^p : \langle \tilde{v} \rangle \mid r^p : (l_i)_{i \in I} \mid r^p : \langle l \rangle \mid \iota$$

In order to avoid usual problems with the alpha renaming of bound names we define $P \stackrel{\lambda}{\longrightarrow}$ only on process without restrictions, using the CST evaluation contexts:

$$\begin{array}{llll} \mathbb{C} & ::= \llbracket \cdot \rrbracket \mid \mathbb{C}|P \mid r^p \triangleright \mathbb{C} \mid \mathbb{C} > \tilde{x} > P & \qquad \mathbb{C}_{r^p} & ::= r^p \triangleright (\llbracket \cdot \rrbracket | P) \\ \mathbb{D} & ::= \mathbb{C}\llbracket \mathbb{C}' \mid \mathbb{C}'' \rrbracket & \qquad \mathbb{D}_r & ::= \mathbb{D}\llbracket \mathbb{C}'_{r^p}, \mathbb{C}''_{r^{\overline{p}}} \rrbracket \end{array}$$

Definition 6.1. Let $P \stackrel{\lambda}{\longrightarrow}$ the least relation s.t.

- If $P \equiv \mathbb{C}\llbracket \mathbb{C}_{r^p} \llbracket (\tilde{x}).P' \rrbracket \rrbracket$ then $P \stackrel{r^p:(\tilde{x})}{\longrightarrow}$
- If $P \equiv \mathbb{C}\llbracket \mathbb{C}_{r^p} \llbracket \langle \tilde{v} \rangle.P' \rrbracket \rrbracket$ then $P \stackrel{r^p:\langle \tilde{v} \rangle}{\longrightarrow}$
- If $P \equiv \mathbb{C}\llbracket \mathbb{C}_{r^p} \llbracket \mathbb{C}_{r_1^q} \llbracket \mathtt{return}\ \tilde{v}.P' \rrbracket \rrbracket \rrbracket$ then $P \stackrel{r^p:\langle \tilde{v} \rangle}{\longrightarrow}$
- If $P \equiv \mathbb{C}\llbracket \mathbb{C}_{r^p} \llbracket \Sigma_{i=1}^n.(l_i).P_i \rrbracket \rrbracket$ then $P \stackrel{r^p:(l_i)_{i \in I}}{\longrightarrow}$ and $I = \{1, \ldots, n\}$
- If $P \equiv \mathbb{C}\llbracket \mathbb{C}_{r^p} \llbracket \langle l \rangle.P' \rrbracket \rrbracket$ then $P \stackrel{r^p:\langle l \rangle}{\longrightarrow}$
- If $P \equiv \mathbb{C}\llbracket a.P' \rrbracket$ then $P \stackrel{\iota}{\longrightarrow}$

- If $P \equiv \mathbb{C}[\![\overline{a}.P']\!]$ then $P \stackrel{\iota}{\longrightarrow}$

All cases are simple but the case for external choice that returns the entire set of offered labels. Notice that the definition of $P \stackrel{\lambda}{\longrightarrow}$ does not account for the pipe since it will be accounted for by Proposition 6.9.

We also define $P \stackrel{\mathbf{0}}{\longrightarrow}$ to characterize all processes that are terminated, think of for example to the process $\mathbf{0} > \tilde{x} > P$.

Definition 6.2. Let P be a process. We write $P \stackrel{\mathbf{0}}{\longrightarrow}$ if there do not exist $P', \tilde{m}$ and λ s.t. $P \equiv (\nu\tilde{m})P'$ and $P' \stackrel{\lambda}{\longrightarrow}$.

As usual we write $\rightarrow^*$ for the transitive and reflexive closure of the transition relation $\rightarrow$ (the transition relation of CST defined in Section 4.2) and we call P' a derivative of P if $P \rightarrow^* P'$. Also we write $P \rightarrow$ if there exists Q s.t. $P \rightarrow Q$ and we write $P \nrightarrow$ if not $P \rightarrow$. We now define formally the deadlock freedom property.

Definition 6.3 (Deadlock-free)**.** A process P is *deadlock-free* if for each Q s.t. $P \rightarrow^* Q$ then either $Q \rightarrow$ or $Q \stackrel{\mathbf{0}}{\longrightarrow}$.

In few words in each step the process must be able either to proceed or it must be ended with success. For example the process $a.\overline{b} \mid \overline{a}.b$ is deadlock free but $\overline{b}.a \mid \overline{a}.b$ is not.

Definition 6.4 (Progress)**.** A well-typed process P has the progress property if for each Q s.t. $P \rightarrow^* Q$ then either $Q \rightarrow$ or there exists a well-typed Q' s.t. $Q|Q'$ is well-typed with $Q' \nrightarrow$ and $Q'|Q \rightarrow$ or $Q \stackrel{\mathbf{0}}{\longrightarrow}$.

Formally progress provides a third choice, the process Q can be itself stuck but composed in parallel with an another process Q' (also stuck) allows the system to evolve. The condition on the fact that Q' has no derivative is important since allows only those Q' which actually participate to the progress of the entire process. Also, since the type system already checks the linearity of opened sessions, the only possibility for Q' to participate to the progress is to offer either a service accept or a service request to locally solve the deadlock.

The progress property is weaker than the deadlock freedom property. For example the process $(\nu a)(a.r^- \triangleright (x))|r^+ \triangleright \langle 5\rangle$ has not the progress property (since a is restricted) and it is deadlocked while $a.r^- \triangleright (x)|r^+ \triangleright \langle 5\rangle$ has the progress property (since a is not restricted) but it is deadlocked. Notice that all the previous examples are well-typed CST processes.

We define a relation that captures all pairs of opened session which are in father/children relation and we also take its transitive closure.

Definition 6.5 (Session nesting relation)**.** Let $\mathbb{R}$ be a generic syntactic CST context and $\mathbb{S}$ a generic syntactic CST context without the production $r^p \triangleright [\![\cdot]\!]$ and P a process with all bound names and free names different then we let $r_1 \prec_P r_2$ iff $P \equiv \mathbb{S}[\![r_1^p \triangleright \mathbb{R}[\![r_2^q \triangleright Q]\!]]\!]$ for some contexts $\mathbb{S}, \mathbb{R}$ and let $<_P$ the transitive closure of $\prec_P$.

We are interested to all processes that have P s.t. $\prec_P$ is irreflexive since it means that do not exist cyclic dependencies among sessions that can cause deadlock. We prove that since the management of sessions is left to the operational semantics it is not the case that a process without deadlock due to the session nesting relation introduces a deadlock during its evaluation.

Lemma 6.6. *Let P a process s.t. $\Gamma; \Theta \vdash P : T; U; L$ and $<_P$ irreflexive. If $P \to Q$ then $<_Q$ is irreflexive.*

Proof. The proof is by induction on the derivation of $P \to Q$ with case analysis on the last applied rule. All cases are simple, and in particular the case for rule (INV) holds because it chooses a fresh session name. □

The next lemma relates the $P \xrightarrow{\lambda}$ relation to the actual process redex.

Lemma 6.7. *If $P \xrightarrow{r^p:(\tilde{x})}$ and $P \xrightarrow{r^{\overline{p}}:\langle\tilde{v}\rangle}$ then either $P \equiv \mathbb{D}_r[\![(\tilde{x}).P', \langle\tilde{v}\rangle.Q']\!]$ or $P \equiv \mathbb{D}_r[\![(\tilde{x}).P, \mathbb{C}_{r_1^q}[\![\mathtt{return}\ \tilde{v}.Q]\!]]\!]$ for some $\mathbb{D}_r$, $\mathbb{C}_{r_1^q}$, P', and Q'. If $P \xrightarrow{r^p:(l_i)_{i\in I}}$ and $P \xrightarrow{r^p:\langle l\rangle}$ then $P \equiv \mathbb{D}_r[\![\Sigma_{i=1}^n(l_i).P_i, \langle l\rangle.Q']\!]$ for some $\mathbb{D}_r$, $P_1, \ldots, P_n$, Q' and $\{l_1, \ldots, l_n\} = \{l_i | i \in I\}$.*

Proof. The proof comes directly from the definition of $P \xrightarrow{\lambda}$ and from the definition of a context $\mathbb{D}_r$. □

The following is an important lemma which says that in a recursive process the first action described by T and U is actually a real action in the body of the recursion P. This fact is important since in Proposition 6.9 we require unfolded processes in order to disallow recursion constructs appearing during the induction. To this end in fact we consider always an unfolded version of the process and this lemma says that we always encounter the actions we need before encounter the next recursion construct.

Lemma 6.8. *Let* $\texttt{rec}\ X.P$ *s.t.* $\Gamma, \emptyset \vdash \texttt{rec}\ X.P : T; U; L$ *then the first action in either* T *or* U *(an input or an output or an internal choice or an external choice) is relative to a corresponding action in* P.

Proof. Since the process we are considering is closed we have by the consistency check that either T or U different from $\texttt{end}$ implies either $\neg nocuract(P)$ or $\neg noretact(P)$. □

Finally we are now ready to formulate the progress of CST valid for the outermost (in terms of $\prec$-relation) active sessions. In fact, if one of such sessions has a pending action enabled then it is either guaranteed that after a finite number of steps a suitable synchronization is accomplished or a service invocation/declaration is issued.

Proposition 6.9. *Let* P *a process s.t. all recursions are unfolded at least once and* $P \equiv (\nu\tilde{m})Q$ *and* $\Gamma; \emptyset \vdash Q : T; U; L$ *and* $<_Q$ *irreflexive. For any session* r *in* Q *if* $\nexists \mathbb{C}, \mathbb{C}_{r^p}, P', \tilde{m}$ *and* $\tilde{w}$ *such that* $Q \equiv (\nu\tilde{m})\mathbb{C}[\![\mathbb{C}_{r^p}[\![\texttt{return}\ \tilde{w}.P']\!]]\!]$ *all the following hold:*

(I) *if* $Q \xrightarrow{r^p:(\tilde{x})}$ *then* $Q \to^* \xrightarrow{r^{\overline{p}}:\langle\tilde{v}\rangle} \vee Q \to^* \xrightarrow{\iota}$

(II) *if* $Q \xrightarrow{r^p:\langle\tilde{v}\rangle}$ *then* $Q \to^* \xrightarrow{r^{\overline{p}}:(\tilde{x})} \vee Q \to^* \xrightarrow{\iota}$

(III) *if* $Q \xrightarrow{r^p:(l_i)_{i\in I}}$ *then there exists* $l_j \in \{l_i | i \in I\}$ *s.t.* $Q \to^* \xrightarrow{r^{\overline{p}}:\langle l_j\rangle} \vee Q \to^* \xrightarrow{\iota}$

(IV) *if* $Q \xrightarrow{r^p:\langle l\rangle}$ *then* $Q \to^* \xrightarrow{r^{\overline{p}}:(l_i)_{i\in I}}$ *and* $l \in \{l_i | i \in I\} \vee Q \to^* \xrightarrow{\iota}$

(V) *if* $Q \equiv (\nu\tilde{m})\mathbb{C}[\![\mathbb{C}_{r^p}[\![P' > \tilde{x} > Q']\!]]\!]$ *for some* $\tilde{m}, \tilde{x}, \mathbb{C}, \mathbb{C}_{r^p}, P', Q'$ *s.t.* $\Gamma'; \emptyset \vdash P' :!(\tilde{S}); U; L$ *for some* Γ' *then* $\mathbb{C}[\![\mathbb{C}_{r^p}[\![P']\!]]\!] \to^* \xrightarrow{r^p:\langle\tilde{v}\rangle} \vee \mathbb{C}[\![\mathbb{C}_{r^p}[\![P']\!]]\!] \to^* \xrightarrow{\iota}$

Proof. The proof is by induction on the length $llns(r, Q)$ of the longest nesting sequence induced by $\prec_Q$ and starting with r; that is the longest sequence of the form $r \prec_Q r_1 \prec_Q r_2 \prec_Q \ldots \prec_Q r_{n-1} \prec_Q r_n$, and then on the structure of the processes. Notice that the well foundedness is due to the fact that by hypothesis $<_Q$ is irreflexive so no cyclic dependencies are possible. More precisely, the well-founded order we consider for the induction is defined on pairs (r^p, Q) by letting $(r_1^{p_1}, Q_1) < (r_2^{p_2}, Q_2)$ be the least transitive relation satisfying:

- $(r_1^{p_1}, Q_1) < (r_2^{p_2}, Q_2)$ if $llns(r_1, Q_1) < llns(r_2, Q_2)$,
- $(r^p, \mathbb{C}[\![\mathbb{C}_{r^p}[\![Q_1]\!]]\!]) < (r^p, \mathbb{C}[\![\mathbb{C}_{r^p}[\![Q_2]\!]]\!])$ if $llns(r, Q_1) = llns(r, Q_2)$ and Q_1 is a subterm of $Q_2[^{\tilde{v}}/_{\tilde{x}}]$ for suitable $\tilde{v}$

We start proving *(I)* which together with subject reduction is

(I) if $Q \xrightarrow{r^p:(\tilde{x})}$ and $\Gamma_1, r^p : [?(\tilde{S}).\overline{T}], r^{\overline{p}} : [!(\tilde{S}).T] \vdash Q : V_1; V_2; L$ then $Q \rightarrow^* \xrightarrow{r^{\overline{p}}:\langle\tilde{v}\rangle} \vee Q \rightarrow^* \xrightarrow{\iota}$

We can read previous statements as "a session side must respect, after a certain number of steps, the obligation imposed by its type unless it postpones the obligation with a new service action". If $r^{\overline{p}} : [?(\tilde{S}).\overline{T}]$ it means that $\Gamma; \emptyset \vdash Q' :!(\tilde{S}).T; U; L'$ for some U, L' where $Q \equiv \mathbb{C}[\![r^{\overline{p}} \triangleright Q']\!]$ for some Γ,$\mathbb{C}$ and Q'. The entire proof is completely type-driven, the key idea is that we consider only rules able to yield $!(\tilde{S}).T$ in the conclusion. For ease of readability and since a runtime process we are considering, cannot have free process variables we use a triple $\Gamma \vdash P : T; U$ as a typing judgment, and we also utilize in addition variables W,R to range over processes.

Base cases: The bases cases are those prefixes compatible with an $r^{\overline{p}}$ output action.

$$\text{(TOUT)}\ \frac{\Gamma', \tilde{v} : \tilde{S} \vdash W : T; U}{\Gamma', \tilde{v} : \tilde{S} \vdash \langle\tilde{v}\rangle.W :!(\tilde{S}).T; U} \qquad \text{(TSES)}\ \frac{\Gamma', r_1^q : [T'], \tilde{v} : \tilde{S} \vdash \texttt{return}\ \tilde{v}.W : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'], \tilde{v} : \tilde{S} \vdash r_1^q \triangleright \texttt{return}\ \tilde{v}.W :!(\tilde{S}).T; \texttt{end}}$$

$$\text{(TINV)}\ \frac{\Gamma', v : [T'] \vdash W : \overline{T'}; !(\tilde{S}).T}{\Gamma', v : [T'] \vdash \overline{v}.W :!(\tilde{S}).T; \texttt{end}} \qquad \text{(TDEF)}\ \frac{\Gamma', v : [T'] \vdash W : T'; !(\tilde{S}).T}{\Gamma', v : [T'] \vdash v.W :!(\tilde{S}).T; \texttt{end}}$$

in these cases we have either $Q \xrightarrow{r^{\overline{p}}:\langle\tilde{v}\rangle}$ or $Q \xrightarrow{\iota}$ which concludes.

Inductive cases:

- When Q' is a parallel composition:

$$\text{(TPARL)}\ \frac{\Gamma \vdash W :!(\tilde{S}).T; U \quad \Gamma \vdash R : \texttt{end}; \texttt{end}}{\Gamma \vdash W|R :!(\tilde{S}).T; U} \qquad \text{(TPARR)}\ \frac{\Gamma \vdash W : \texttt{end}; \texttt{end} \quad \Gamma \vdash R :!(\tilde{S}).T; U}{\Gamma \vdash W|R :!(\tilde{S}).T; U}$$

  The thesis follows by induction hypothesis on $\mathbb{C}[\![r^{\overline{p}} \triangleright W]\!]$ for (TparL) and $\mathbb{C}[\![r^{\overline{p}} \triangleright R]\!]$ for (TparR).

- When Q' is an if-then-else:

$$\frac{\Gamma \vdash v_i : S_1 \quad i = 1,2 \quad \Gamma \vdash W :!(\tilde{S}).T;U \quad \Gamma \vdash R :!(\tilde{S}).T;U}{\Gamma \vdash \texttt{if } v_1 = v_2 \texttt{ then } W \texttt{ else } R :!(\tilde{S}).T;U} \text{(TIF)}$$

The thesis follows by induction hypothesis either on $\mathbb{C}[\![r^{\overline{p}} \triangleright W]\!]$ or $\mathbb{C}[\![r^{\overline{p}} \triangleright R]\!]$ depending on the evaluation of the if guard.

- When Q' is a pipe:

$$\frac{\Gamma \vdash W :!(\tilde{S}'); \texttt{end} \quad \Gamma, \tilde{x} : \tilde{S}' \vdash R :!(\tilde{S}).T;U}{\Gamma \vdash W > \tilde{x} > R :!(\tilde{S}).T;U} \text{(TPIPE)}$$

We apply the induction hypothesis case *(V)* on $\mathbb{C}[\![r^{\overline{p}} \triangleright W]\!]$. If $\mathbb{C}[\![r^{\overline{p}} \triangleright W]\!] \rightarrow^* \xrightarrow{r^{\overline{p}}:\langle \tilde{v}' \rangle} \mathbb{C}[\![r^{\overline{p}} \triangleright W']\!]$ then $\mathbb{C}[\![r^{\overline{p}} \triangleright (W > \tilde{x} > R)]\!] \rightarrow^* \mathbb{C}[\![r^{\overline{p}} \triangleright (W' > \tilde{x} > R) | R[^{\tilde{v}'}/_{\tilde{x}}]]\!]$, and then the thesis follows by induction hypothesis on $\mathbb{C}[\![r^{\overline{p}} \triangleright R[^{\tilde{v}'}/_{\tilde{x}}]]\!]$. The thesis follows directly from the inductive hypothesis, if $\mathbb{C}[\![r^{\overline{p}} \triangleright W]\!] \rightarrow^* \xrightarrow{\iota}$.

- When Q' is a restriction:

$$\frac{\Gamma, s : S' \vdash W :!(\tilde{S}).T;U}{\Gamma \vdash (\nu s)W :!(\tilde{S}).T;U} \text{(TNEW)} \qquad \frac{\Gamma, r_1^+ : [T'], r_1^- : [\overline{T'}] \vdash W :!(\tilde{S}).T;U}{\Gamma \vdash (\nu r)W :!(\tilde{S}).T;U} \text{(TNEWR)}$$

Follows directly by inductive hypothesis on $\mathbb{C}[\![r^{\overline{p}} \triangleright W]\!]$.

- When Q' is a recursion (where $\star$ is the consistency check):

$$\frac{\Gamma \vdash (W :!(\tilde{S}).T;U)^\star}{\Gamma \vdash \texttt{rec } X.W :!(\tilde{S}).T;U} \text{(TREC)}$$

By Lemma 6.8 this case cannot happen since we should have found the output action relative to $!(\tilde{S})$ before encountering the recursion operator remember in fact we have unfolded all recursions once.

- In case of a nested session r_1^q we have W with different shapes.

$$\frac{\Gamma', r_1^q : [T'] \vdash W : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'] \vdash r_1^q \triangleright W :!(\tilde{S}).T; \texttt{end}} \text{(TSES)}$$

– In case W is an if-then-else, a recursion and a name restriction, the thesis follows directly by the application of the inductive hypothesis; the other cases follow.
– In case of an input inside a nested session

$$\frac{\Gamma', r_1^q : [T'] \vdash (\tilde{y}).W : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'] \vdash r_1^q \rhd (\tilde{y}).W :!(\tilde{S}).T; \mathsf{end}}\ (\text{TSES})$$

Since $Q \xrightarrow{r_1^q:(\tilde{y})}$ and $r \prec_Q r_1$ then by induction hypothesis we have two cases. $Q \to^* \xrightarrow{r_1^q:\langle \tilde{v}' \rangle}$ and then by Lemma 6.7 we have a further reduction step. The thesis follows by another application of the induction hypothesis on $\mathbb{C}[\![r^{\overline{p}} \rhd (r_1^q \rhd W)]\!]$. Otherwise if $Q \to^* \xrightarrow{\iota}$ the thesis follows directly.
– Cases relative to other actions within a nested session

$$\frac{\Gamma', r_1^q : [T'] \vdash \langle \tilde{w} \rangle.W : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'] \vdash r_1^q \rhd \langle \tilde{w} \rangle.W :!(\tilde{S}).T; \mathsf{end}}\ (\text{TSES}) \qquad \frac{\Gamma', r_1^q : [T'] \vdash \langle l \rangle.W : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'] \vdash r_1^q \rhd \langle l \rangle.W :!(\tilde{S}).T; \mathsf{end}}\ (\text{TSES})$$

$$\frac{\Gamma', r_1^q : [T'] \vdash \sum_{i=1}^n (l_i).W_i : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'] \vdash r_1^q \rhd \sum_{i=1}^n (l_i).W_i :!(\tilde{S}).T; \mathsf{end}}\ (\text{TSES})$$

are similar to the previous one.
– In case of a parallel composition within a nested session:

$$\cfrac{\cfrac{\Gamma', r_1^q : [T'] \vdash W : \mathsf{end}; \mathsf{end} \qquad \Gamma', r_1^q : [T'] \vdash R : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'] \vdash (W|R) : T'; !(\tilde{S}).T}\ (\text{TPARR})}{\Gamma', r_1^q : [T'] \vdash r_1^q \rhd (W|R) :!(\tilde{S}).T; \mathsf{end}}\ (\text{TSES})$$

$$\cfrac{\cfrac{\Gamma', r_1^q : [T'] \vdash W : T'; !(\tilde{S}).T \qquad \Gamma', r_1^q : [T'] \vdash R : \mathsf{end}; \mathsf{end}}{\Gamma', r_1^q : [T'] \vdash (W|R) : T'; !(\tilde{S}).T}\ (\text{TPARL})}{\Gamma', r_1^q : [T'] \vdash r_1^q \rhd (W|R) :!(\tilde{S}).T; \mathsf{end}}\ (\text{TSES})$$

Thesis follows directly by induction on the process producing the output.
– In case of a pipe within a nested session:

$$\cfrac{\cfrac{\Gamma', r_1^q : [T'] \vdash W :!(\tilde{S}'); \mathsf{end} \qquad \Gamma', r_1^q : [T'], \tilde{y} : \tilde{S}' \vdash R : T'; !(\tilde{S}).T}{\Gamma', r_1^q : [T'] \vdash W > \tilde{y} > R : T'; !(\tilde{S}).T}\ (\text{TPIPE})}{\Gamma', r_1^q : [T'] \vdash r_1^q \rhd (W > \tilde{y} > R) :!(\tilde{S}).T; \mathsf{end}}\ (\text{TSES})$$

We apply the induction hypothesis case *(V)* on $\mathbb{C}[\![r^{\overline{p}} \triangleright (r_1^q \triangleright W)]\!]$. If $\mathbb{C}[\![r^{\overline{p}} \triangleright (r_1^q \triangleright W)]\!] \rightarrow^* \xrightarrow{r_1^q\langle\tilde{w}\rangle}$ $\mathbb{C}[\![r^{\overline{p}} \triangleright (r_1^q \triangleright W')]\!]$ then $\mathbb{C}[\![r^{\overline{p}} \triangleright (r_1^q \triangleright (W > \tilde{y} > R))]\!] \rightarrow^*$ $\mathbb{C}[\![r^{\overline{p}} \triangleright (r_1^q \triangleright (W' > \tilde{y} > R)|R[^{\tilde{v}}/_{\tilde{y}}])]\!]$, and then the thesis follows by induction hypothesis on $\mathbb{C}[\![r^{\overline{p}} \triangleright (r_1^q \triangleright W')]\!]$. If instead $\mathbb{C}[\![r^{\overline{p}} \triangleright (r_1^q \triangleright W)]\!] \rightarrow^* \xrightarrow{\iota}$ then the thesis follows directly.

- The case of a nested recursion cannot happen since by Lemma 6.8 we should have found the return output before encounter the recursion operator.

Statement *(II)* together with subject reduction is

(II) if $Q \xrightarrow{r^p:\langle\tilde{v}\rangle}$ and $\Gamma_1, r^p : [!(\tilde{S}).\overline{T}], r^{\overline{p}} : [?(\tilde{S}).T]; \emptyset \vdash Q : V_1, V_2; L$ then $Q \rightarrow^* \xrightarrow{r^{\overline{p}}:(\tilde{x})} \vee Q \rightarrow^* \xrightarrow{\iota}$

Base cases: The base case is the prefix compatible with an $r^{\overline{p}}$ input action.

$$\frac{\Gamma, \tilde{x} : \tilde{S} \vdash W : T; U}{\Gamma \vdash (\tilde{x}).W : ?(\tilde{S}).T; U} \ (\text{TIN})$$

in this case we have $Q \xrightarrow{r^{\overline{p}}:(\tilde{x})}$ which concludes.
Inductive cases: The inductive cases comprise parallel composition, if-then-else, pipe and restrictions all similar to previous case.
Statement *(III)* together with subject reduction is

(III) if $Q \xrightarrow{r^p:(l_i)_{i\in I}}$ and $\Gamma_1, r^p : [\overline{\oplus\{l_j : T_j\}_{j\in J}}], r^{\overline{p}} : [\oplus\{l_j : T_j\}_{j\in J}]; \emptyset \vdash Q : V_1, V_2; L$ and $\{l_j | j \in J\} \subseteq \{l_i | i \in I\}$ then there exists k s.t. $Q \rightarrow^* \xrightarrow{r^{\overline{p}}:\langle l_k\rangle}$ and $l_k \in \{l_j | j \in J\} \vee Q \rightarrow^* \xrightarrow{\iota}$

Base cases: The base case is the prefix compatible compatible with an $r^{\overline{p}}$ label choice action.

$$\frac{l = l_k \in \{l_j | j \in J\} \quad \Gamma \vdash W : T_k; U}{\Gamma \vdash \langle l\rangle.W : \oplus\{l_j : T_j\}_{j\in J}; U} \ (\text{TCHOICE})$$

in this case we have $Q \xrightarrow{r^{\overline{p}}:\langle l\rangle}$ which concludes.
Inductive cases: The inductive cases comprise parallel composition, if-then-else, pipe and restrictions all similar case (I).
Statement *(IV)* together with subject reduction is

```
let transitiveclousure g l=
    (List.iter (fun i->
       (List.iter (fun j->
            (List.iter (fun k->
            if ((Hashtbl.mem g (i,k))=true) &&
               ((Hashtbl.mem g (k,j))=true) &&
               ((Hashtbl.mem g (i,j))=false)
               then  (Hashtbl.add g  (i,j) true)
               else ()) l) )l) )l)
let loop p=let g=getsessgraph p in
   let l=listsession p in  (transitiveclousure g l);
     (List.fold_right (fun s y->(Hashtbl.mem g  (s,s)) or y) l false)
```

Figure 41: The transitive closure of the session relation

(IV) if $Q \stackrel{r^p:\langle l \rangle}{\longrightarrow}$ and $\Gamma_1, r^p : [\overline{\&\{l_j : T_j\}_{j\in J}}], r^{\overline{p}} : [\&\{l_j : T_j\}_{j\in J}]; \emptyset \vdash Q : V_1; V_2; L$ then $Q \rightarrow^* \stackrel{r^{\overline{p}}:(l_i)_{i\in I}}{\longrightarrow}$ and $l \in \{l_j | j \in J\} \subseteq \{l_i | i \in J\} \vee Q \rightarrow^* \stackrel{\iota}{\longrightarrow}$

Base cases: The base case is the prefix compatible with an $r^{\overline{p}}$ label guarded sum action.

$$\frac{\text{(TBRANCH)} \quad \emptyset \subset J \subseteq I = \{1, \ldots, n\} \quad \forall i \in I \Gamma \vdash W_i : T_i; U}{\Gamma \vdash \Sigma_{i=1}^{n} (l_i).W_i : \&\{l_j : T_j\}_{j\in J}; U}$$

The inductive cases comprise parallel composition, if-then-else, pipe and restrictions all similar case (I).

Inductive cases: The inductive cases comprise rules: (TPARL), (TPARR), (TIF), (TPIPE), (TNEW), (TNEWR), (TREC) and they are all similar to previous case. Finally, statement (V) we have $Q \equiv (\nu\tilde{m})\mathbb{C}[\![\mathbb{C}_{r^p}[\![P' > \tilde{x} > Q']\!]]\!]$ and it is similar to statement (I) with case analysis on the typing of $\Gamma'; \emptyset \vdash P' : !(\tilde{S}); U; L$. □

The previous proposition ensures a very powerful property: a process can stuck only on either service invocations or either service definitions. With the progress property we introduce into the stuck process respectively either the service definition or the service invocation in order to allow the communications within opened sessions to complete. Let us show the progress property for CST.

Theorem 6.10 (CST Progress). *Let P a process without service restrictions. If $\Gamma; \emptyset \vdash P : T; U; L$ and $<_P$ is irreflexive then P has the progress property.*

Proof. Let $P \rightarrow^* P'$ since each process and its unfoldings have the same reductions by rule (REC) of the operational semantics the Proposition 6.9 says that if $<'_P$ irreflexive (which is the case by Lemma 6.6) and $\Gamma'; \emptyset \vdash P' : T; U; L$ (which is the case by Subject Reduction) then P' should need either a service invocation or either a service definition. The result follows providing a mapping similar to that defined in (26) (which is simpler here due the lack of session delegation) that takes a type and returns a process. For simplicity we show such mapping for a monadic CST.

$$
\begin{array}{lcll}
\mathcal{B}(!(int).T) & = & \langle 5\rangle.\mathcal{B}(T) & \\
\mathcal{B}(!([U]).T) & = & \langle a\rangle.\mathcal{B}(T) & a \text{ fresh} \\
\mathcal{B}(?(S).T) & = & (x).\mathcal{B}(T) & \\
\mathcal{B}(\oplus\{l_i : T_i\}_{i\in I}) & = & \langle l_k\rangle.\mathcal{B}(T_k) & l_k \in \{l | l_i \in I\} \\
\mathcal{B}(\&\{l_i : T_i\}_{i\in I}) & = & \Sigma_{i\in I}(l_i).\mathcal{B}(T_i) & \\
\mathcal{B}(\mu\alpha.T) & = & \texttt{rec}\ X_\alpha.\mathcal{B}(T) & \\
\mathcal{B}(\alpha) & = & X_\alpha & \\
\mathcal{B}(\texttt{end}) & = & \mathbf{0} &
\end{array}
$$

Hence if P' is stuck on the definition of a then $P'|\overline{a}.\mathcal{B}(\overline{T}) \rightarrow$ otherwise if P' is stuck on the invocation of a then $P'|a.\mathcal{B}(T) \rightarrow$. □

The statement of the Progress Theorem can be generalized removing the condition on the service restrictions modifying the type system to collect the set of bound service names. Hence if a service is bound, the body of the service definition and the body of the service invocation are required to be typed with L equal to the empty set.

Next we tackle the problem of algorithmically check irreflexivity of the transitive closure of a session nesting relation. We can make such check using an adaption of the Bellman-Ford algorithm reported in Figure 41. Such algorithm is a dynamic programming algorithm and has a complexity of n^3 where n is the number of sessions we are considering. The algorithm takes `g` which is the adjacency matrix that contains the session nesting relation and a list `l` that contains all the sessions in the session nesting relation. We report the algorithm which is interestingly straightforward since each session name is distinct. When the algorithm ends if `g` does not contain elements in the diagonal then the transitive closure is irreflexive otherwise not. The same algorithm can be used to compute the transitive closure required in (4; 26) to check the acyclicity of the dependencies among session types. If the aim is only checking the acyclicity of a graph the Depth First Search (DFS) algorithm can be used instead which has a lower complexity bound.

6.2 Revisiting our sources of inspiration

In this section we compare our work with our main sources of inspiration. We describe CaSPiS (9) by means of an example to have a flavor of the calculus, which it suffices since the proposal has not any type discipline. We present the original calculus introduced in (37), which we call SL, since it checks linearity of session without polarity information and finally we introduce the π-calculus for session types, which we call π-ST, introduced in (33).

6.2.1 CaSPiS

CaSPiS is a calculus described in (9) and directly derived from SCC (8). We gently introduce the calculus by means of a few simple examples since the calculus inspired CST but in this manner we can outline the additions the presence of session types require. First of all the operational semantics in the original proposal is given by means of a labeled transition system. We find reductions more suitable in a context where types are used. For example in CaSPiS the operational semantics is given in a early style, where the rule for input locally guesses the received value. Of course the subject reduction does not hold for this rule (one can modify the calculus to allow type annotations in each binder in order to have typed labels in the LTS, like the proposal in (1)) since one can receive any possible value with a very different type.

Within CaSPiS service definitions are rendered as

$$a.P$$

where a is the service name and P is the *body* defining the service behavior. P can be seen as a process that receives/sends values from/to the client side and then activates the corresponding computational activities. For instance,

$$\texttt{succ}.(?x)\langle x+1\rangle$$

models a service that, after receiving an integer, sends back the successor of the received value.

Service invocations can be seen as specific instances of output prefixed processes and they are rendered as

$$\overline{a}.P$$

where a is the name of the service to invoke while P is the process implementing the client-side protocol for interacting with the new instance of a. As an example, a client for the simple service described above will be written in CaSPiS as

$$\overline{\texttt{succ}}.\langle 5\rangle(?y)\langle y\rangle^{\uparrow}$$

After succ is invoked, argument 5 is passed on to the service side and the client waits for a value from the server: the received value will be substituted for y and returned as the overall result of the service invocation.

A service invocation leads to the activation of a new session and a fresh, private, name r is used to bind the two sides of the session. For instance, the interaction of the client and of the service described above triggers the session

$$(\nu r)\big(r \rhd \langle 5+1\rangle \,|\, r \rhd (?y)\langle y\rangle^{\uparrow}\big)$$

Notice that r is created without polarity annotations. Value 6 is computed at the service-side and then received at the client side; the remaining activity is then performed by the client-side of the session

$$r \rhd \langle 6\rangle^{\uparrow}$$

that emits the value 6 outside of the session and becomes

$$r \rhd \mathbf{0}$$

where $\mathbf{0}$ denotes the empty process.

More generally, within sessions, communication is bi-directional, in the sense that the interacting peers can exchange data in both directions. Values returned outside of the session (to the enclosing environment) with the return operator $\langle . \rangle^{\uparrow}$ can be used for invoking other services. Indeed, processes can be composed by using the *pipe* operator

$$P > Q$$

A new instance of process Q is activated in correspondence of each of the values produced by P that Q can receive. For instance, what follows is a client that invokes the service succ and then prints the obtained result:

$$\langle 5\rangle > (?x)\overline{\texttt{succ}}.\langle x\rangle(?y)\langle y\rangle^{\uparrow} > (?z)\overline{\texttt{print}}.\langle z\rangle$$

In CST the syntax of the pipe operator is $P > \tilde{x} > Q$ we syntactically constraints Q to input the value communicated by P.

To improve usability, structured values are permitted; services are invoked using structured values that, via pattern matching, drive usage of the exchanged information.

Using this approach, each service can provide different *methods* corresponding to the exposed activities. For instance:

$$\begin{array}{rl} calculator. & (\text{“}sum\text{”}, ?x, ?y)\langle \text{“}result\text{”}, x + y\rangle \\ + & (\text{“}sub\text{”}, ?x, ?y)\langle \text{“}result\text{”}, x - y\rangle \\ + & (\text{“}mul\text{”}, ?x, ?y)\langle \text{“}result\text{”}, x * y\rangle \\ + & (\text{“}div\text{”}, ?x, ?y)\langle \text{“}result\text{”}, x/y\rangle \end{array}$$

models a service $calculator$ that exposes the methods for computing the basic arithmetic operations. Notice in CST we have a limited form of pattern matching supported by means of choices. This service can be invoked as follows:

$$\overline{calculator}.\langle \text{“}sum\text{”}, 2, 4\rangle(\text{“}result\text{”}, ?y)\langle y\rangle^{\uparrow}$$

A similar approach is used for session interaction. Indeed, thanks to tags and pattern matching, more sophisticated protocols can be programmed for both the server and client side of a session. For instance, a service-side can reply to a client request with different values denoting the status of the execution:

$$r \triangleright (\text{“}fail\text{”}, ?x)P_1 + (\text{“}result\text{”}, ?y)P_2$$

Finally CaSPiS is equipped also with primitives for handling session closure. These primitives are useful to garbage-collect terminated sessions and, most importantly, to explicitly program session termination in order to manage abnormal events or timeouts. We do not describe this part of CaSPiS since it is not supported in the type discipline.

6.2.2 The Honda-Vasconcelos-Kubo Session Typing System

In this section we review the original calculus described in (76). Actually the work describes two different calculi the one reported below and another one with session polarity similar to HVK-X in Chapter 5 but

$$
\begin{array}{llll}
P, Q & ::= & \texttt{request } a(k) \texttt{ in } P & \text{(session request)} \\
& | & \texttt{accept } a(k) \texttt{ in } P & \text{(session acceptance)} \\
& | & k![\tilde{v}]; P & \text{(data sending)} \\
& | & k?(\tilde{x}) \texttt{ in } P & \text{(data reception)} \\
& | & k \triangleleft l; P & \text{(label selection)} \\
& | & k \triangleright \{l_1 : P_1 \| \ldots \| l_n : P_n\} & \text{(label choice)} \\
& | & \texttt{throw } k[k']; P & \text{(session sending)} \\
& | & \texttt{catch } k(k'); P & \text{(session reception)} \\
& | & \texttt{if } v_1 = v_2 \texttt{ then } P \texttt{ else } Q & \text{(if-then-else)} \\
& | & P|Q & \text{(parallel)} \\
& | & \texttt{inact} & \text{(inaction)} \\
& | & (\nu m)P & \text{(restriction)} \\
& | & \texttt{def } D \texttt{ in } P & \text{(recursion)} \\
& | & X[\tilde{v}\tilde{k}] & \text{(process variable)} \\
\\
D & ::= & \begin{array}{l} X_1(\tilde{x}_1\tilde{k}_1) = P_1 \texttt{ and} \ldots \\ \texttt{and } X_n(\tilde{x}_n\tilde{k}_n) = P_n \end{array} & \text{(declarations)}
\end{array}
$$

Figure 42: Syntax of SL

without subtyping and general recursion. We report only the first calculus which is interesting since it keeps linearity without annotating any polarity information just using a special type $\perp$ to indicate session on which no further interaction is possible. We refer to this calculus as Session Language or SL for short. Regarding the other calculus with polarity we prefer presenting the π-calculus with session types by Gay and Hole instead, which is the first work to introduce the idea of polarity.

The syntax of SL assumes the usual set of infinite collections but in addition uses $k, \ldots$ for session names.

Then processes ranged over by $P, Q, \ldots$ are given by the grammar in Figure 42. We stress the fact that k is very different from the κ of HVK-X here k is actually a session not a session variable. Binders are $k?(\tilde{x})$ `in` P for $\tilde{x}$ in P, $X(\tilde{x}, \tilde{k}) = P$ for $\tilde{x}$ and $\tilde{k}$ in P, $(\nu m)P$ for m in P. Furthermore `accept` $a(k)$ `in` P and `accept` $a(k)$ `in` P and `catch` $k'(k); P$ bind k in P and binders for process variables are each process definition in `def` D `in` P. The differences with HVK-X are the presence of process definition which allows both a tuple of variables and a tuple of sessions as parameters but the lack of name extrusion i.e. SL has a service a in both

$$P|\texttt{inact} \equiv P \quad P|Q \equiv Q|P \quad (P|Q)|R \equiv P|(Q|R)$$
$$(\nu m)P|Q \equiv (\nu m)(P|Q) \text{ if } m \notin \mathsf{fn}(Q)$$
$$(\nu m)\mathbf{0} \equiv \mathbf{0}$$
$$\texttt{def } D \texttt{ in inact} \equiv \texttt{inact}$$
$$(\nu m)\texttt{def } D \texttt{ in } P \equiv \texttt{def } D \texttt{ in } (\nu m)P \text{ if } m \notin \mathsf{fn}(Q)$$
$$(\texttt{def } D \texttt{ in } P)|Q \equiv \texttt{def } D \texttt{ in } (P|Q) \text{ if } \mathsf{bpv}(D) \cap \mathsf{fpv}(Q) = \emptyset$$
$$\texttt{def } D \texttt{ in } (\texttt{def } D' \texttt{in } P)|Q \equiv \texttt{def } D \texttt{ and def } D' \texttt{ in } P \text{ if } \mathsf{bpv}(D) \cap \mathsf{bpv}(D') = \emptyset$$

Figure 43: The structural congruence of SL

$$\text{(LINK)} \; (\texttt{accept } a(k) \texttt{ in } P_1)|(\texttt{request } a(k) \texttt{ in } P_2) \rightarrow (\nu k)(P_1|P_2)$$
$$\text{(COM)} \; (k![\tilde{v}]; P_1)|(k?(\tilde{x}) \texttt{ in } P_2) \rightarrow (\nu k)(P_1|P_2)$$
$$\text{(LABEL)} \; (k \triangleleft l_i; P)|(k \triangleright \{l_1 : P_1 \| \ldots \| l_n : P_n\}) \rightarrow P|P_i \quad (1 \leq i \leq n)$$
$$\text{(PASS)} \; (\texttt{throw } k[k']; P)|(\texttt{catch } k(k'); P) \rightarrow P_1|P_2$$
$$\text{(IF1)} \; \texttt{if } v = v \texttt{ then } P_1 \texttt{ else } P_2 \rightarrow P_1$$
$$\text{(IF2)} \; \texttt{if } v = w \texttt{ then } P_1 \texttt{ else } P_2 \rightarrow P_2 \quad (v \neq w)$$
$$\text{(DEF)} \; \texttt{def } D \texttt{ in } (X[\tilde{v}\tilde{k}]|Q) \rightarrow \texttt{def } D \texttt{ in } (P[{}^{\tilde{v}}/_{\tilde{x}}]|Q) \quad (X(\tilde{x}\tilde{k}) = P \in D)$$
$$\text{(SCOP)} P \rightarrow P' \Rightarrow (\nu m)P \rightarrow (\nu m)P'$$
$$\text{(PAR)} P \rightarrow P' \Rightarrow P|Q \rightarrow P'|Q$$
$$\text{(DEFIN)} P \rightarrow P' \Rightarrow \texttt{def } D \texttt{ in } P \rightarrow \texttt{def } D \texttt{ in } P'$$
$$\text{(STR)} P \equiv P' \text{ and } P' \rightarrow Q' \text{ and } Q' \equiv Q \Rightarrow P \rightarrow Q$$

Figure 44: The operational semantics of SL

the accept and the request instruction instead of a value v. The structural congruence in Figure 43 allows floating of restrictions and associativity of process definition with respect to the parallel composition. The last rule groups together nested process definitions and bpv is the set of bound process variables. The operational semantics of SL is reported in Figure 44. Notice the difference in rule (LINK) which directly creates a binder for k as a private session for P_1 and P_2. Rule (PASS) expects k' to be *the same session* in order to synchronize. The essence of this rule is related to a trick in a rule of the operational semantics of a variant of the π-calculus called the πI-calculus (67). This calculus restricts name passing to the bound private name passing. Moreover if k' is free in P_2 the communication never happens since the impossibility of alpha renaming

$$
\begin{array}{lcl}
S & ::= & int \mid (T,\overline{T}) \\
T & ::= & ?(\tilde{S}).T \mid ?(T).U \mid \&\{l_1 : T_1, \ldots, l_n : T_n\} \mid \texttt{end} \mid \bot \mid \\
 & & !(\tilde{S}).T \mid !(T).U \mid \oplus\{l_1 : T_1, \ldots, l_n : T_n\} \mid \alpha \mid \mu\alpha.T
\end{array}
$$

Figure 45: Syntax of session types for SL

the bound sessions to syntactically match k'.

Eventually we now introduce the set of types employed in SL in Figure 45 and the type system in Figure 46. The type relative to a service collects both the dual types relative to both ends of a communication. A special type $\bot$ is introduced in the session type syntax. The idea behind the $\bot$ type is simple, since we want a session k to appear at most twice in a process we substitute the type of k with $\bot$ if k appears dual in the linear environment of both P and Q during the typing of $P|Q$. Only restrictions of sessions with type $\bot$ are allowed in rule (CRES). The following definition used in rule (CONC) formally defines this intuition:

Definition 6.11. Linear environments Δ_1 and Δ_2 are compatible, written $\Delta_1 \asymp \Delta_2$, if $\Delta_1(k) = \overline{\Delta_1(k)}$ for all $k \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$. When $\Delta_1 \asymp \Delta_2$, the composition of Δ_1 and Δ_2, written $\Delta_1 \circ \Delta_2$, is given as a linear environment such that $(\Delta_1 \circ \Delta_2)(k)$ is (1) $\bot$, if $k \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$; (2), $\Delta_i(k)$ if $k \in \mathsf{dom}(\Delta_i) \setminus \mathsf{dom}(\Delta_{i+1 \bmod 2})$ for $i \in \{1,2\}$; and (3) undefined otherwise.

We now outline the differences with our type system. Axioms (VAR) and (INACT) introduce all the ended sessions necessary during the typing derivation. We do the same but using (SWEAK) and the subtyping relation among linear environments. Rule (BOT) allows transformation of ended sessions to $\bot$ sessions. The case is tricky, and without this rule Subject Congruence fails in presence of delegation of ended sessions. We show an example of typing derivation of this case which we believe it is very useful to better understand how this type system works.

Example 6.12. Take the process `throw` $k[k']$; `inact`|`inact` structural congruent to `throw` $k[k']$; `inact`. One possible typing is the following:

$$
\cfrac{\cfrac{\emptyset;\emptyset \vdash \texttt{inact} \rhd k : \texttt{end}}{\emptyset;\emptyset \vdash \texttt{throw}\ k[k'];\texttt{inact} \rhd k' : \texttt{end}, k : !(\texttt{end})}\ (\text{THR}) \qquad \emptyset;\emptyset \vdash \texttt{inact} \rhd k' : \texttt{end}}{\emptyset;\emptyset \vdash \texttt{throw}\ k[k'];\texttt{inact}|\texttt{inact} \rhd k' : \bot, k : !(\texttt{end})}\ (\text{CONC})
$$

$$\text{(Bot)}\ \frac{\Gamma;\Theta \vdash P \triangleright \Delta, k : \mathtt{end}}{\Gamma;\Theta \vdash P \triangleright \Delta, k : \bot} \qquad \text{(Inact)}\ \Gamma;\Theta \vdash \mathtt{inact} \triangleright \tilde{k} : \widetilde{\mathtt{end}}$$

$$\text{(Acc)}\ \frac{\Gamma \vdash a : (T, \overline{T}) \quad \Gamma;\Theta \vdash P \triangleright \Delta, k : T}{\Gamma;\Theta \vdash \mathtt{accept}\ a(k)\ \mathtt{in}\ P \triangleright \Delta} \qquad \text{(Req)}\ \frac{\Gamma \vdash a : (T, \overline{T}) \quad \Gamma;\Theta \vdash P \triangleright \Delta, k : \overline{T}}{\Gamma;\Theta \vdash \mathtt{request}\ a(k)\ \mathtt{in}\ P \triangleright \Delta}$$

$$\text{(Send)}\ \frac{\Gamma \vdash \tilde{v} : \tilde{S} \quad \Gamma;\Theta \vdash P \triangleright \Delta, k : T}{\Gamma;\Theta \vdash k![\tilde{v}]; P \triangleright \Delta, k : !(\tilde{S}).T} \qquad \text{(Rcv)}\ \frac{\Gamma, \tilde{x} : \tilde{S};\Theta \vdash P \triangleright \Delta, k : T}{\Gamma;\Theta \vdash k?(\tilde{x})\ \mathtt{in}\ P \triangleright \Delta, k : ?(\tilde{S}).T}$$

$$\text{(Br)}\ \frac{\Gamma;\Theta \vdash P_1 \triangleright \Delta, k : T_1 \quad \dots \quad \Gamma;\Theta \vdash P_n \triangleright \Delta, k : T_n}{\Gamma;\Theta \vdash k \triangleright \{l_1 : P_1 \| \dots \| l_n : P_n\} \triangleright \Delta, \&\{l_1 : T_1, \dots, l_n : T_n\}}$$

$$\text{(Sel)}\ \frac{\Gamma;\Theta \vdash P \triangleright \Delta, k : T_j \quad (1 \leq j \leq n)}{\Gamma;\Theta \vdash k \triangleleft l_j; P \triangleright \Delta, \oplus\{l_1 : T_1, \dots, l_n : T_n\}}$$

$$\text{(Thr)}\ \frac{\Gamma;\Theta \vdash P \triangleright \Delta, k : T}{\Gamma;\Theta \vdash \mathtt{throw}\ k[k']; P \triangleright \Delta, k : !(U).T, k' : U} \qquad \text{(Cat)}\ \frac{\Gamma;\Theta \vdash P \triangleright \Delta, k : T, k' : U}{\Gamma;\Theta \vdash \mathtt{catch}\ k(k')\ \mathtt{in}\ P \triangleright \Delta, k : ?(U).T}$$

$$\text{(Conc)}\ \frac{\Gamma;\Theta \vdash P \triangleright \Delta \quad \Gamma;\Theta \vdash Q \triangleright \Delta' \quad \Delta \asymp \Delta'}{\Gamma;\Theta \vdash P|Q \triangleright \Delta \circ \Delta'} \qquad \text{(If)}\ \frac{\Gamma \vdash v_i : S \quad \Gamma;\Theta \vdash P \triangleright \Delta \quad \Gamma;\Theta \vdash Q \triangleright \Delta}{\Gamma;\Theta \vdash \mathtt{if}\ v_1 = v_2\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta}$$

$$\text{(NRes)}\ \frac{\Gamma, a : S;\Theta \vdash P \triangleright \Delta}{\Gamma;\Theta \vdash (\nu a)P \triangleright \Delta} \qquad \text{(CRes)}\ \frac{\Gamma;\Theta, k : \bot \vdash P \triangleright \Delta}{\Gamma;\Theta \vdash (\nu k)P \triangleright \Delta} \qquad \text{(Var)}\ \frac{\Gamma \vdash \tilde{v} : \tilde{S}}{\Gamma;\Theta, X : \tilde{S}\tilde{T} \vdash X[\tilde{v}\tilde{k}] \triangleright \tilde{k} : \tilde{T}, \tilde{k}' : \widetilde{\mathtt{end}}}$$

$$\text{(Def)}\ \frac{\Gamma;\Theta, X : \tilde{S}\tilde{T} \vdash \Gamma, \tilde{x} : \tilde{S} \triangleright \tilde{k} : \tilde{T} \quad \Gamma;\Theta, X : \tilde{S}\tilde{T} \vdash Q \triangleright \Delta}{\Gamma;\Theta \vdash \mathtt{def}\ X(\tilde{x}\tilde{k}) = P\ \mathtt{in}\ Q \triangleright \Delta}$$

Figure 46: The SL type system

$$\cfrac{\cfrac{\emptyset;\emptyset \vdash \mathtt{inact} \triangleright k : \mathtt{end}}{\emptyset;\emptyset \vdash \mathtt{throw}\ k[k']; \mathtt{inact} \triangleright k' : \mathtt{end}, k : !(\mathtt{end})}\ \text{(Thr)}}{\emptyset;\emptyset \vdash \mathtt{throw}\ k[k']; \mathtt{inact} \triangleright k' : \bot, k : !(\mathtt{end})}\ \text{(Bot)}$$

Notice how thanks to the rule (Bot) both processes are typed in the same linear environment. We shown the pathological typing, the process can also be typed assuming k' different from `end` and in these other cases typing is simpler since k' cannot be part of the linear environment relatives to `inact`.

Another peculiarity of this type system is that subtyping relation is not considered explicitly. The subtyping is implicitly achieved with rule (Sel) which adds arbitrary branches in order to satisfy the duality condition in both rules (Acc) and (Req). With this limitation (Br) is too constrained and for example it disallows the presence of replicas of a service with different behaviors. We report the statement of the subject reduction

theorem.

Theorem 6.13 (see Theorem 2.10 in (76)). *If* $\Gamma; \Theta \vdash P \triangleright \Delta$ *and* $P \rightarrow Q$ *then* $\Gamma; \Theta \vdash Q \triangleright \Delta$

The particularity of this type system is that the linear environment stays unchanged during reductions.

As a final observation it is simple to use the same techniques we employed for HVK-X to extract the set of constraints from this type system without polarities. The main change is in the rule for parallel composition which should generate the same constraints that in HVK-X we generate for session restriction. We think that it should be simple to adapt the type discipline of HVK-X in order to extract constraints for this type system too. Also it would not hard to prove that each well-typed process in the second of the calculi reported in (76) is also a well typed HVK-X process (assuming the encoding of process definition).

6.2.3 The Gay-Hole Session Typing System

Gay and Hole studied the subtyping relation directly in the π-calculus with session types, πST for short. As usual for the π-calculus they assume only the existence of a collection of names, together with a collection of labels. However names may be polarized, occurring as either x^+ or x^- or simply as x. The syntax of πST is reported in Figure 47, and here x^p can be only x to represent a standard π-calculus channel. The definition of free names is slightly non-standard. Binding occurrences of names are x in $(\nu x : S)P$ and $\tilde{y}$ in $x^p?[\tilde{y} : \tilde{S}].P$, with the particularity that in $(\nu x : S)P$ either x or both x^+ and x^- may occur in P and both are bound. In $x^p?[\tilde{y} : \tilde{S}].P$ for each $y \in \{\tilde{y}\}$ only y unpolarized may occur in P. Binders are annotated with only sort types S since in this calculus each T can be also an S (Figure 48). The type $\hat{}[S_1, \ldots, S_n]$ represents the type of a standard π-calculus channel in which we can either transmit or receive values of type $S_1, \ldots, S_n$. Recursion is provided also at level of sorts since for example one can write $x![x]$ and x is allowed to be typed as $\mu\alpha.\hat{}[\alpha]$. As we showed in the typing of Example 4.5 we do not need this addition because in our case the recursion is always guarded by a session type even when using a service definition in its own body.

The structural congruence is reported in Figure 49. It allows removing ended session from only the nil process (since as usual the nil process allows only ended sessions in its typing) and it forbids the floating of

$$
\begin{array}{rll}
P, Q ::= & \mathbf{0} & \text{(terminated process)} \\
| & P|Q & \text{(parallel combination)} \\
| & !P & \text{(replication)} \\
| & x^p?[y_1 : S_1, \ldots, y_n : S_n].P & \text{(input)} \\
| & x^p![y_1^{p_1}, \ldots, y_n^{p_n}].P & \text{(output)} \\
| & (\nu x : S)P & \text{(channel creation)} \\
| & x^p \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} & \text{(branch)} \\
| & x^p \triangleleft l.P & \text{(choice)}
\end{array}
$$

Figure 47: Syntax of πST

$$
\begin{array}{rcl}
T & ::= & \alpha \mid \texttt{end} \mid ?(S_1, \ldots, S_n).T \mid !(S_1, \ldots, S_n).T \\
 & & \&\{l_1 : T_1, \ldots, l_n : T_n\} \mid \oplus \{l_1 : T_1, \ldots, l_n : T_n\} \mid \mu\alpha.T \\
S & ::= & \alpha \mid T \mid \hat{}[S_1, \ldots, S_n] \mid \mu\alpha.S
\end{array}
$$

Figure 48: Syntax of πST types

a binder if the binder is relative to an ended session since no further communications are allowed. The operational semantics given in Figure 50 is the standard reductions semantics enriched with labels. In $P \xrightarrow{\lambda, l} Q$, λ is the subject of the communication (rule (R-Com)) and l is a label in case the reduction is relative to a label selection (rule (R-Select)). These two parameters of the reduction relation are used to reduce the session type annotated in the relative binder (rule (R-News)) by means of the function $tail$. As special labels they use τ for silent action in which the name is hidden and $_$ to stay for any label l. Other rules are standard.

The type system for πST is reported in Figure 51 and type judgments are of the form $\Gamma \vdash P$. In rule (T-Nil) the premise Γ completed, means that all the session types contained in Γ must be ended which is similar to require a linear environment of the form $\tilde{\kappa} : \widetilde{\texttt{end}}$ and in rule (T-Rep) the premise Γ unlimited means that Γ must not contain any session type which is similar to require an empty linear environment. Linearity of session types is kept by means of the operator of environment composition $\Gamma_1 + \Gamma_2$ (first introduced in (47)), which definition is rather long but simple, it disallows a session with the same polarity to appear twice in Γ_1, Γ_2. Rule (T-In) allows the actual type annotated in the binder to be a

$$P|\mathbf{0} \equiv P \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)|R \quad !P \equiv P|!P$$
$$(\nu x : S)P|Q \equiv (\nu x : S)(P|Q) \text{ if } x, x^+, x^- \notin \mathsf{fn}(Q) \text{ and } T \neq \mathtt{end}$$
$$(\nu x : T)\mathbf{0} \equiv \mathbf{0} \text{ if } T \text{ is not a session type}$$
$$(\nu x : \mathtt{end})\mathbf{0} \equiv \mathbf{0} \quad (\nu x : S)(\nu y : S')P \equiv (\nu y : S')(\nu x : S)P$$

Figure 49: Structural congruence of πST

(R-COM)
$$x^p![\tilde{y} : \tilde{T}].P|x^{\overline{p}}![\tilde{z}^{\tilde{q}}].Q \xrightarrow{x,_} P[^{\tilde{z}^{\tilde{q}}}/_{\tilde{y}}]|Q$$

(R-SELECT)
$$\frac{p \text{ is either } + \text{ or } - \quad 1 \leq i \leq n}{x^p \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}|x^{\overline{p}} \triangleleft l_i.P \xrightarrow{x,l_i} P_i|Q}$$

(R-NEW)
$$\frac{P \xrightarrow{\lambda,l} P' \quad \lambda \neq x \quad S \text{ is not a session type}}{(\nu x : S)P \xrightarrow{\lambda,l} (\nu x : T)P'}$$

(R-NEWS)
$$\frac{P \xrightarrow{x,l} P'}{(\nu x : T) \xrightarrow{\tau,_} (\nu x : tail(T,l))P'}$$

(R-PAR)
$$\frac{P \xrightarrow{\lambda,l} P'}{P|Q \xrightarrow{\lambda,l} P'|Q}$$

(R-CONG)
$$\frac{P' \equiv P \quad P \xrightarrow{\lambda,l} Q \quad Q \equiv Q'}{P' \xrightarrow{\lambda,l} Q'}$$

$$tail(?(\tilde{S}).T, _) = T \qquad tail(!(\tilde{S}).T, _) = T \qquad tail(\&\{l_i : T_i\}_{i\in I}, l_i) = T_i$$
$$tail(\oplus\{l_i : T_i\}_{i\in I}, l_i) = T_i \qquad tail(\mu\alpha.T, l) = tail(T[^{\mu\alpha.T}/_\alpha], l)$$

Figure 50: Operational semantics of πST

subtype of the assumed type since we can use a type less than to what is prescribed without errors. Rule (T-OUT) behaves contravariant with respect to the type annotated since we can annotate a less type than the actual one without incurring in any errors. In our type system we obtain a similar effect by using rule (SWEAK). Moreover rule (T-OUT) makes a clever usage of the environment composition, the fact that it appears in the premise disallows P to use any of the outputted sessions. In this type system rule (T-OFFER) allows a selection of a subset of the real offered labels but differently from us, they allow the remaining branch to not be typed at all. Instead we type all the processes appearing in a choice in order to allow achieving the completeness of the syntax directed rules.

We report the subtyping relation which is slight different from the one we used, since it allows both S and T to be in the relation (remember that every S is also a T here).

Definition 6.14 (Subtyping). A relation $\mathcal{R} \subseteq Type \times Type$ is a type simulation if $(T, U) \in \mathcal{R}$ implies the following conditions:

1. If $\mathsf{unfold}(S) = \hat{}[S_1, \ldots, S_n]$ then $\mathsf{unfold}(S') = \hat{}[S'_1, \ldots, S'_n]$ and for

$$\frac{\Gamma \text{ completed}}{\Gamma \vdash \mathbf{0}}\ (\text{T-NIL}) \qquad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P|Q}\ (\text{T-PAR}) \qquad \frac{\Gamma \vdash P \quad \Gamma \text{ unlimited}}{\Gamma \vdash !P}\ (\text{T-REP})$$

$$\frac{\Gamma, x : T \vdash P \quad T \text{ is not a session type}}{\Gamma \vdash (\nu x : T)P}\ (\text{T-NEW}) \qquad \frac{\Gamma, x^+ : T, x^- : U \vdash P \quad T \bot_c U}{\Gamma \vdash (\nu x : T)P}\ (\text{T-NEWS})$$

$$\frac{\Gamma, x^p : T, \tilde{y} : \tilde{S}' \vdash P \quad \tilde{S}'' \leq \tilde{S}'}{\Gamma, x^p : ?(\tilde{S}'').T \vdash x?[\tilde{y} : \tilde{S}'].P}\ (\text{T-INS}) \qquad \frac{\Gamma, x^p : T \vdash P \quad \tilde{S}' \leq \tilde{S}''}{(\Gamma, x^p : !(\tilde{S}'').T) + \tilde{y}^{\tilde{q}} : \tilde{S}' \vdash x^p![\tilde{y}^{\tilde{q}}].P}\ (\text{T-OUTS})$$

$$\frac{\Gamma, x : \hat{}[\tilde{S}''], \tilde{y} : \tilde{S}' \vdash P \quad \tilde{S}'' \leq \tilde{S}'}{\Gamma, x : \hat{}[\tilde{S}''] \vdash x?[\tilde{y} : \tilde{S}'].P}\ (\text{T-IN}) \qquad \frac{\Gamma, x : \hat{}[S''] \vdash P \quad \tilde{S}' \leq \tilde{S}''}{(\Gamma, x : \hat{}[S'']) + \tilde{y}^{\tilde{q}} : \tilde{S}' \vdash x^p![\tilde{y}^{\tilde{q}}].P}\ (\text{T-OUTS})$$

$$\frac{J \subseteq I \quad \forall i \in J.(\Gamma, x^p ; T_j \vdash P_j)}{\Gamma, x^p : \&\{l_j : T_j\}_{j \in J} \vdash x^p \triangleright \{l_i : P_i\}_{i \in I}}\ (\text{T-OFFER}) \qquad \frac{l = l_i \in \{l_1, \ldots, l_n\} \quad \Gamma, x^p : T_i \vdash P}{\Gamma, x^p : \oplus\{l_1 : T_1, \ldots, l_n : T_n\} \vdash x^p \triangleleft l.P}\ (\text{T-CHOOSE})$$

Figure 51: The type system of πST

all $i \in \{1, \ldots, n\}$, $(S_i, S_i') \in R$ and $(S_i', S_i) \in R$.

2. If $\mathsf{unfold}(S) = ?(S_1, \ldots, S_n).T$ then $\mathsf{unfold}(S') = ?(S_1', \ldots, S_n').U$ and $(T, U) \in R$ and for all $i \in \{1, \ldots, n\}$, $(S_i, S_i') \in R$.
3. If $\mathsf{unfold}(S) = !(S_1, \ldots, S_n).T$ then $\mathsf{unfold}(S') = !(S_1', \ldots, S_n').U$ and $(T, U) \in R$ and for all $i \in \{1, \ldots, n\}$, $(S_i', S_i) \in R$.
4. If $\mathsf{unfold}(S) = \&\{l_1 : T_1, \ldots, l_m : T_m\}$ then $\mathsf{unfold}(S') = \&\{l_1 : U_1, \ldots, l_n : U_n\}$ and $m \leq n$ and for all $i \in \{1, \ldots, m\}$, $(T_i, U_i) \in R$.
5. If $\mathsf{unfold}(S) = \oplus\{l_1 : T_1, \ldots, l_m : T_m\}$ then $\mathsf{unfold}(S') = \oplus\{l_1 : U_1, \ldots, l_n : U_n\}$ and $n \leq m$ and for all $i \in \{1, \ldots, n\}$, $(T_i, U_i) \in R$.
6. If $\mathsf{unfold}(S) = \mathtt{end}$ then $\mathsf{unfold}(S') = \mathtt{end}$.

The co-inductive subtyping relation $\leq$ is defined by $S \leq S'$ if and only if there exists a type simulation R such that $(S, S') \in R$.

Finally the rule (T-NEWS) uses $T \bot_c U$ to checks the duality of two session types allowing infinite types with different representation to be dual e.g. $\mu\alpha.!(int).\alpha \bot_c \mu\alpha.?(int).?(int).\alpha$. The duality relation is defined on only session types (which set is pointed with $SType$) by the following co-inductive definition:

Definition 6.15 (Duality)**.** A relation $R \subseteq SType \times SType$ is a duality relation if $(T, U) \in R$ implies the following conditions:

1. If $\mathsf{unfold}(T) = ?(S_1, \ldots, S_n).T'$ then $\mathsf{unfold}(U) = ?(S'_1, \ldots, S'_n).U'$ and $(T', U') \in R$ and for all $i \in \{1, \ldots, n\}$, $(S_i, S'_i) \in R$ and $(S'_i, S_i) \in R$.

2. If $\mathsf{unfold}(T) = !(S_1, \ldots, S_n).T'$ then $\mathsf{unfold}(U) = !(S'_1, \ldots, S'_n).U'$ and $(T', U') \in R$ and for all $i \in \{1, \ldots, n\}$, $(S_i, S'_i) \in R$ and $(S'_i, S_i) \in R$.

3. If $\mathsf{unfold}(S) = \&\{l_1 : T_1, \ldots, l_n : T_n\}$ then $\mathsf{unfold}(S') = \&\{l_1 : U_1, \ldots, l_n : U_n\}$ and for all $i \in \{1, \ldots, n\}$, $(T_i, U_i) \in R$.

4. If $\mathsf{unfold}(S) = \oplus\{l_1 : T_1, \ldots, l_n : T_n\}$ then $\mathsf{unfold}(S') = \oplus\{l_1 : U_1, \ldots, l_n : U_n\}$ and for all $i \in \{1, \ldots, n\}$, $(T_i, U_i) \in R$.

5. If $\mathsf{unfold}(S) = \mathtt{end}$ then $\mathsf{unfold}(S') = \mathtt{end}$.

The co-inductive duality relation $\perp_c$ is defined by $T \perp_c U$ if and only if there exists a duality relation R such that $(T, U) \in R$.

Again we do the same by means of the rule (SWEAK) since one can easily prove that $T \perp_c U$ iff $T \leqslant \overline{U}$. We end this section reporting the Subject Reduction Theorem for this type system.

Theorem 6.16 (see Theorem 1 in (33)). *Subject Reduction*

1. *If* $\Gamma \vdash P$ *and* $P \xrightarrow{\tau,-} Q$ *then* $\Gamma \vdash Q$*.*

2. *If* $\Gamma, x^+ : T, x^- : U \vdash P$ *and* $T \perp_c U$ *and* $P \xrightarrow{x,l} Q$ *then* $\Gamma, x^+ : tail(T, l), x^- : tail(\overline{U}, l) \vdash Q$*.*

3. *If* $\Gamma, x : T \vdash P$ *and* $P \xrightarrow{x,-} Q$ *then* $\Gamma, x : T \vdash Q$*.*

The first point is relative to restricted communications, the second point is relative to session types and the third point is relative to standard π-calculus communications. Notice that after the reduction $\perp_c$ degenerates to the syntactic duality since it holds that $T \perp_c \overline{T}$.

6.3 Other related works

There is a lot work on type reconstruction for the π-calculus (70), Tyco (71; 73), and lambda-calculus with records and recursive types, however the problem addressed here is slight different. The novelty of this work is that it discovers the typability of a process using session types that are

specified in width (due to the possibly infinite sequence of actions) and in depth (due to the types of the values exchanged). For example in a object calculus the type of each object comprises only the set of its methods (similar to an unique external choice) and the type discipline uses kinds to exhibit the principal type. Kinds used in (73) permit constraining the type of an object using the set of its methods. It is not clear how to adopt kinds for the principal typing of a session type. Here we are in presence of dual communications so the kinds must account for both parties in the conversation. Moreover one has to introduce something similar to (72) constraining the entire (possible) infinite tree built from a session type. What we do here is to check a set of constraints at the type system level using the subtyping relation.

Regarding the works done on π-calculus they cannot be directly used here since the presence of subtyping while type discipline for the π-calculus uses unification (either on finite or on infinite trees) to reconstruct channel sorts. The same is applied to the lambda calculus with functional type (74).

Also existing reconstruction algorithms for recursion are relative to channels that recursively use themselves, it is something like the type inference of the type of a recursive data structure (list,tree,...). These algorithms use unification among infinite trees, here we address the problem of discovering the behavior of a recursion intended as a construct to build unbounded sequences of actions. The case is similar to the solution of recursive behaviors given in (45) but they use the channel sorting discipline of the simply typed π-calculus as basis and they have replication in place of the general recursion. The theory of unification of infinite trees can be used also here (as we have showed in Example 4.37) for a service that recursively uses itself.

Using co-inductive definitions in the type system is certainly not new (40; 45; 46; 49) but here we can use the power of the co-induction proof method (34) conscious of the presence of a simple algorithm for checking it.

There are a lot of works on session types. In (32) they used session types in BASS, an ambient like calculus. It also interesting how they modified the operational semantics in order to count the number of opened sessions: then an ambient is allowed to move only if it does not have opened sessions with its parent. In (6) they studied the correspondence assertion for session types. The notion of correspondence assertions was introduced by Woo and Lam (75) for stating expected authenticity properties formally. In (42) they studied the problem of type inference of cor-

respondence assertions for the π-calculus. It is not clear how to adapt type inference of correspondence assertions and correspondence assertions studied for session types with our results. The main point is that the type system proposed in (6) is an ordered type system in which the order of assumptions in a typing environments is relevant. In particular assumptions depend on each other since a substitution is used in the rule for output to track the exchanged values. Tracking values allows the automatic verification of assertions. Also session types are used in (12) to represent component interface and to study the problem of component adaptation resulting in a high level notation for writing component adaptor.

In (38) they studied multiparty session types, whose problem was first introduced in (7) to type a multipoint communication. In a multipoint communication, one side, for example the one that perform a request, waits for other n participants that issue an accept. However, their calculus called DCMS (distributed calculus with multipoint session types) is asynchronous and allows starting a conversation without waiting all the parties. During the conversation the side who performed a request is allowed to communicate with the other n sides while each one of the n sides is allowed to communicate only with the requesting side. For instance in the proposed ATM scenario (7), the ATM can communicate both with the bank and with the customer. The ATM is called the master endpoint while both the bank and the customer are slave endpoints. Each party in the conversation is identified by a location and session types are annotated with location too. For example, $?^{cl}(int).!^{bk}(int)$ is a fragment of the ATM type relative to a conversation with both the client (pointed with cl) and the bank (pointed with bk). From the type of a master endpoint one can obtain the type of each slave endpoint by the so-called projection operation. The duality is a slight complex and is obtained equating the projection of a slave with the dual type of the slave endpoints. We think the main problem in reusing our results in DCMS would be related to give a solution of to set of constraints containing projections whose definition is rather involved. Subsequently in (38) they generalize the problem to n parties communicating with each other by means of a global type. A global type is a high level description of the communication that reports the global choreography of the process for instance $A \to B : \kappa(int).G$ is the global type that describes a communication from A to B using the session channel κ to send an integer and then it behaves as prescribed by G. Having the global type specification simplifies the definition of projection in such a manner that constraints

involving projection can be reduced to duality constraints. In fact, in (29) they argued that the type inference of multiparty sessions types can be reduced to the type inference of local dyadic session types.

In (26) they studied the progress property for session types and the type system uses a similar rule to our (SWEAK) in order to add ended assumptions, however the subtyping relation is not mentioned at all. We think that the mechanism they used for checking the acyclicity of opened session can be implemented on top of HVK-X without changing in the type system in a similar manner to what they do in (4) where there are in fact two different type systems: one for session safety and another one for progress.

There are also a lot of calculi specifically designed for service oriented computing such as (16; 17; 54) (just to cite few). As we discussed in Chapter 2 we choose CST since its mechanism for the instantiation of session types naturally arise specializing those in (33; 76).

The two proposals in (20; 51) are derived from SCC: the conversation calculus and the stream based service centered calculus (SSCC). The conversation calculus extends SCC with directional communications towards the current session, the parent session and the dual session while in SCC only outputs are allowed towards the first two sessions. This is achieved by orienting the subject of each communication instead of using the pipe and the return constructs. SSCC uses a stream to return values by means of the `feed` primitive. A stream can be seen to as a container of values and feeded values can be read accessing the stream by its name. The type system proposed in SSCC resembles the type system of CST but a stream (differently from the return values of CST) can contain only one type of values and the calculus allows only for a sequence of inputs and outputs without choices.

There are a lot of practical works on implementation of session types (23; 30; 39) and how to use session types in a standard programming language. Also in (27) they propose a type inference algorithm for session types without the presence of choices and delegation; we guess that it is straightforward to adapt the present theory for their object oriented language.

Finally in (21; 52) they studied the problem of checking the communication safety (or the compliance) treating session types as a specific term of a process algebra: a contract. With contracts one can face the problem reusing classical theory (24). In (53) they spelled out the connection between contracts and session types. In (11) they studied service composition (not only one client and one service) by means of contract and the

subcontract relation that allows replacing a group of compliant contracts with one of their subcontract obtaining a new compliant composition.

6.4 Final remarks on CST progress

In this chapter we have proved the progress property for CST which is a direct consequence of the subject reduction with the additional cost of computing the transitive closure of the session nesting relation. The transitive closure can be efficiently computed by means of the Bellman-Ford algorithm which is simpler due to the fact that session names can be used directly as the identifier of the respective edge in the graph. Also the same algorithm can be used to compute the reflexive closure needed for the progress property of HVK-X (26).

Chapter 7

Conclusion

The contribute of this thesis in centered around the session types framework first introduced in (37). Session types are already a well studied paradigm (6; 26; 28; 33; 37; 76) but here we have studied how is it possible to relieve the programmer from the burden of annotating programs with types. We started our study from the subtyping relation $\leq$ introduced by Gay and Hole in (33). Actually this subtyping relation is a pre-order since the antisymmetry is lost due the fact that the same infinite type can have several finite representations e.g.: $\mu\alpha.!(int).\alpha$ and $!(int).\mu\alpha.!(int).\alpha$ are the equivalent representations of the type corresponding to an infinite sequence of outputs of integer values. Starting from this subtyping pre-order we have considered the derived notion of equivalence relation taking $\lesseqgtr = \leq \cap \leq^-$.

As it is common when one must deal with type reconstruction we have studied the notion of intersection $T_1 \wedge T_2$ algebraically derived from the subtyping relation. In fact, we have defined the intersection of two session types as the greatest lower bound (modulo $\lesseqgtr$). We have provided a co-inductive characterization of the intersection and we have developed an algorithm capable to compute the intersection of two session types and to return one of its finite representations.

Since we are studying the problem of type reconstruction we need to deal with types possibly containing free type variables. We face the problem simplifying the original proposal by removing the subtyping relation in depth, used for session delegation. With this simplification we introduce the important notion of syntactic unifier, for the subtyping relation and for the meet exists relation. This syntactic unifier is the equivalent to

the most general unifier. However, it remains the most general as long as set of free variables is equal to the set of the free communicated variables. We have also introduced a correct algorithm that returns a substitution in case of constraints with free variables.

We have proved some nice properties about session types and the respective subtyping relation. All of this is reported in Chapter 3, that can serve as a useful source for these readers interested in the more abstract results of this thesis. For example we have proved that the duality switches the order of the operators for the subtyping relation so that the join of two session types can be reduced to the meet of two session types. We also have proved some lattice-like properties, for instance we define the relation minorant that captures the (possibly infinite) set of minorants of two session types and we have proved that the existence of a lower bound implies the existence of the greatest lower bound (see Lemma 3.13).

In Chapter 4 we have studied how to embed the session types framework in a calculus directly derived from CaSPiS (9). This calculus provides an excellent setting for studying the type reconstruction of session types. We call this calculus CaSPiS for Session Types, or CST. The good thing about CST is that it allows managing only two sessions per time, the current one (used to talk with a partner) and the parent one (used to talk with the parent session which encloses the current one).

We have first introduced a non syntax directed type system which has its main source of non-determinism in the rule (TWEAK)(see Figure 16), for session types relaxation. Another problem we have faced is the inference of recursive behaviors.

Given the limited number of sessions that one can handle each time, we have studied a very naive approach to the type inference of recursion, that is if a session has a certain type T then the corresponding replicated version has type $\mu\alpha.T$. To this end we have introduced the so-callled $\bullet$-machinery that introduces a $\bullet$ symbol as a placeholder for a recursion variable. Again thanks to the $\bullet$-machinery we have proved some interesting properties about session types. For example if $T \leq U$ then substituting one or more trailing `end` with a recursion variable α in order to obtain the two types T' and U' one can conclude $\mu\alpha.T' \leq \mu\alpha.U'$. The converse is more surprising if two recursive types are in subtyping relation $\mu\alpha.T_1 \leq \mu\alpha.U_1$ we can find two equivalent representations s.t. $\mu\alpha.T_1 \lesseqgtr \mu\alpha.T_1'$ and $\mu\alpha.U_1 \lesseqgtr \mu\alpha.U_1'$ in order to replace recursion with $\bullet$ and obtain two types that are still in subtyping relation. The proof of this fact comes directly from the representation of the intersection returned

by our algorithm for meet that expands the two types in order to have two equivalent types (in the sense of $\leqslant$) with the same syntactic structure.

Next to avoid type variables relative to the recursion we have introduced a type system that collects a set of types for each session. For example when we find an if-then-else we compute the union of the two types returned by each branch. This syntax directed type system uses sets and the $\bullet$-machinery to infer the type of a recursion and an in-line subtyping checking to get rid of the rule (Tweak). The correspondence with the original type system holds only for the correctness part (since it does not support recursion for nested services) if there exists the intersection of each type contained in the set relative to the current session. We propose `INF`, an algorithm that is able to extract a set of constraints that are satisfied if and only if the process is typable in the set based type system. Our `solve` algorithm can be used to solve a set of constraints generated by `INF` and it succeeds if and only if there exists a substitution that satisfies the set of constraints. As a consequence of using this syntax directed type system with sets we have a very simple `solve` algorithm and consequently an elegant proof of its soundness and completeness. In fact using sets we avoid inserting free variables in a type, thus we can use the syntactic unifier to solve each constraint. We conclude Chapter 4 introducing a sound and complete syntax-directed type system with the corresponding `INF` algorithm. However we did not show the corresponding `solve` algorithm as the one introduced in Chapter 5 can be used instead.

In Chapter 5 we study the type inference of session types for the original calculus proposed in (76) which we call HVK-X from the initials of the original authors. We do not retain the original name due to some differences. First of all we choose full recursion in place of process definition second we allow service name extrusion (with the consequent possibility of the dynamic refinement of service accepts and service requests). As we have done before for CST, we have first introduced a syntax directed type system to get rid of the rule (Tweak). To this end we ideate a function for accessing linear environments (used to store assumptions about opened sessions) that introduces ended sessions only when it is strictly necessary. However this is not sufficient to avoid completely ended sessions since spurious ended sessions can be introduced by means of a throw instruction. We remedy to the presence of spurious sessions using a subtyping relation between linear environments that limits the domain of action of the subtyping relation in the introduction of ended sessions. The relative `INF` algorithm extracts a set of constraints which has cyclic

dependencies, arbitrary free variables and whose types are constrained by means of the duality relation. We propose our solve algorithm which is correct, but complete only partially. In particular the completeness is achieved if the analyzed process is closed and has not free names.

An interesting fact is that one can study a type system with sets for this calculus too. At a first glance it is also correct and complete without delegation and one can prove formally this fact. How is it possible, if in CST (that has no session delegation) we achieved only the correctness with the set syntax directed type system? The answer to this question is given by the encoding function: recursion in CST is encoded using session delegation in HVK-X. For this reason and also for debugging purposes it is nice to show two encodings. We prove the goodness of these encodings showing the correspondence between type systems, instead of giving an operational correspondence. The first encoding we proposed is from the π-calculus to HVK-X: we have shown that a well typed process in the simply typed π-calculus is also well typed when encoded in HVK-X and vice versa. One can think of this result as the fact that session types used just as a simply request-response interactions degenerate to the simply typed π-calculus whose type discipline only checks that each channel is used to send and receive only one type of values. The second encoding we have proposed is from CST to HVK-X and the result is very surprising since each well typed process in CST is a well typed encoded process in HVK-X and vice versa. Notwithstanding, the encoding function is weak, if we try to relate a processes with its encoding we have the same granularity of the type system. In fact, the correspondence between type systems holds even if we would add into the encoding an arbitrary process Q typed with the same standard environment but with the empty linear environment regardless if Q is executed or not. The problem is that the execution of Q can interfere with the execution of the encoded process consuming shared names and session types are not powerful enough to capture this fact.

As far as properties of the subtyping relation are concerned, we have proved in Chapter 5 that the subtyping relation is preserved by arbitrary n-holes contexts. This fact is not obvious since for example recursion is able to duplicate each hole appearing in a context.

In Chapter 6 we have shown the differences with our major sources of inspirations comparing them with our work so one can better appreciate the differences and the similarities. We have shown also how to prove a more powerful property then the session safety which is a direct consequence of the subject reduction. The progress property (here we prove

it for CST) ensures that the dependencies among opened sessions are not cyclic, and then it provides a way to specify a new process that allows the entire system to proceed. This process is composed in parallel with the original stuck process and it is either an accept or a request on a missed service. The deadlock freedom property instead checks that in fact not only the dependencies among opened sessions are not cyclic but also that the system provides all the service accepts and all the service requests it requires for a correct termination. We have concluded Chapter 6 with an overview of the related works.

7.1 Future work

We have studied session types as a framework for interactions among services. One can try to fit session types in real standards. It is easy to see that using WSDL-like or RPC patterns with session types they degenerate to the simply typed π-calculus. It is also possible to model WSDL-2 interactions which can optionally generate fault messages by means of choices. We can also find a way to directly model WSCL (3) and its activity diagrams or maybe to model local parties in a chorography written in WS-CDL. Moreover other standards can be used to secure the message exchanging such as WS-SecureConversation (5). We can spend a lot of efforts providing a mapping to these standards versus session types, but such a task would not contribute substantially to our research outcome which has a more formal flavor and can be applied in different settings. In fact, here we consolidated a theory that can be used whenever two parties communicate with a sequence of bidirectional messages exchanging. We have shown how to achieve additional flexibility using branching, delegation and recursion for an unbounded (not known a priori) conversation. Web Services are certainly the closest paradigm that actually exists but not the only one. Take for example the REST architecture. In a nutshell (see (65) for a detailed discussion about REST), entire systems are modeled in REST by means of resources. On each resource only four basic actions are possible: creating a new resource, updating a resource, removing a resource and reading the state of a resource. Every system can be modeled by a set of resources that offers these basic actions. The closest paradigm to a REST architecture is client-side Javascript with HTTP used to offer resources by means of an URL. On each URL one can make a basic action by means of the basic standard actions of the HTTP protocol (POST,GET and DELETE). One can map session types

also in this paradigm providing for example a resource that allows for the creation of new sessions. This resource first waits two parties asking for a private communication and then it releases a new session resource. Subsequently, standard HTTP actions GETs are used to read and POSTs are used to send according to the fact that each time each party provides the identifier of the session resource. REST with RUBY is a paradigm actually used more and more frequently than Web service standards, every applications we run in a browser are going to become resource oriented. Moreover they actually are services provided by means of a set of resources.

The point is that one can try to apply the session type framework to existing standards, but standards in computer science are often replaced by new ones. Hoping that we have provided a well founded theory inspired from service oriented architecture but that can be used every time two parties need to communicate using non trivial protocols. The calculi we have outlined are used to focus only on communication primitives but people have started studying how to embed it in everyday programming language like Java (27; 39). Perhaps one can say that with session types support, Java is more service oriented than before or one can simply say, we studied a paradigm to simplify the life of programmers in developing communicating applications. As witness of this fact we propose the SOAM machine (14), an abstract machine which can communicate only using sessions. We prove its expressive power providing three encodings for three different calculi, which in particular comprise the encoding of ORC (22), closer to the REST architecture than the other calculi available in literature.

Appendix

In this appendix we show some running examples of TypSes see Chapter 5 for technical details.

The ATM example

We have encoded in HVK-X the Example 4.1 from (37). The syntax of TypSes is slight different, in order to have a clear readability of the source code.

```
new a in (rec X.accept a(k).k?(id).k|>
              deposit:request b(h).k?(amt).h<|deposit.h!(id,amt).X
              ||withdraw:request b(h).k?(amt).h<|withdraw.h!(id,amt).
                              h|>success:k<|dispense.k!(amt).X
                                 || failure:k<|overdraft.X
              ||balance:request b(h).h<|balance.h?(amt).k!(amt).X |
request a(k).k!(12345).k<|withdraw.k!(58).k|>dispense:k?(amt)
                                            ||overdraft:0 )
```

The ATM accepts requests on `a` and uses `b` to communicate with the bank. Initially the ATM requires the `id` of the user and then it allows either `deposit` or `withdraw` or `balance` requests. Once one of those options is chosen by the user a connection is established with the bank on *b*. The bank actually performs the required task and then responds to the ATM which provides the output to the user. For example `overdraft` can be viewed as the exception handling code, e.g. to display an error message to the user. Notice that the definition of `b` is missing so we expect to use <u>localsolve</u>.

Running TypSes we obtain the following constraints:

```
\scriptsize
[alpha11]=s(alpha1)
```

```
alpha11<=?( s(alpha3)).&{deposit:?( s(alpha9)).end ,
                         withdraw:?( s(alpha6)).alpha7 ,
                         balance:!( s(alpha4)).end , }

[alpha10]=s(alpha2)

\oplus{deposit:!( s(alpha3) s(alpha9)).end , } d<= alpha10

[alpha8]=s(alpha2)

\oplus{withdraw:!( s(alpha3) s(alpha6)).&{success:end ,
                                          failure:end , } , } d<= alpha8
alpha7<=\oplus{dispense:!( s(alpha6)).end , }

alpha7<=\oplus{overdraft:end , }

[alpha5]=s(alpha2)

\oplus{balance:?( s(alpha4)).end , } d<= alpha5

[alpha13]=s(alpha1)

!( int).\oplus{withdraw:!( int).&{dispense:?( s(alpha12)).end ,
                                  overdraft:end , } , } d<= alpha13
```

where <= is the subtyping relation and `d<=` is the subtyping of the form $\overline{T} \leq U$ with the first member overlined and variables of the form `s(alpha)` are type variables for sorts. After some iterations of `solve` we obtain the following output:

```
\oplus{balance:?( s(alpha4)).end , } d<= alpha5

\oplus{withdraw:!( int int).&{success:end ,
                              failure:end , } , } d<= alpha5

\oplus{deposit:!( int s(alpha9)).end , } d<= alpha5

!( int).\oplus{withdraw:!( int).&{dispense:?( s(alpha12)).end ,
                                  overdraft:end , } , } d<=
?( int).&{deposit:?( s(alpha9)).end ,
          withdraw:?( int).\oplus{dispense:!( int).end , } ,
          balance:!( s(alpha4)).end , }

------
\oplus{balance:?( s(alpha4)).end , } d<= alpha5

\oplus{withdraw:!( int int).&{success:end ,
                              failure:end , } , } d<= alpha5

\oplus{deposit:!( int s(alpha9)).end , } d<= alpha5

------

The process is typable!
Elapsed Time: 0.015sec
```

where `------` separates each resolution step of the algorithm. In the penultimate line TypSes checks the compatibility of the two specifications of service `a` relative to the ATM and in the last step the algorithm uses <u>localsolve</u> to locally decide the type of `b`.

The FTP example

We have encoded in HVK-X the Example 4.3 from (37) with two threads.

```
new b in (rec FTPD.accept pid(s).request b(k).throw k(s).FTPD |
rec FTPTHREAD.accept b(k).catch k(s).s?(userid,passwd).
      request nis(j).j<|checkuser.j!(userid,passwd).
                         j|>invalid:s<|sorry.O
                            ||valid:s<|welcome.rec ACTIONS.
                             s|>get:s?(file).ACTIONS
                               ||put:s?(file).ACTIONS
                               ||bye:FTPTHREAD |
rec FTPTHREAD.accept b(k).catch k(s).s?(userid,passwd).
      request nis(j).j<|checkuser.j!(userid,passwd).
                         j|>invalid:s<|sorry.O
                            ||valid:s<|welcome.rec ACTIONS.
                             s|>get:s?(file).ACTIONS
                               ||put:s?(file).ACTIONS
                               ||bye:FTPTHREAD |
| request pid(s).s!(123,123).s|>sorry:O||welcome:s<|bye.O
```

This example uses session delegation in order to distribute the workload among different `FTPTHREAD` available. Here (this is not the case of (37)) we can have different specifications of each thread `b` as long as these specifications are compatible with the client. Running TypSes we obtain:

```
The process is typable!
Elapsed Time: 0.156sec
```

In order to get a type error one can for instance modify the client omitting the specification of the branch labeled with `sorry` then:

```
Error protocols
  \oplus{welcome:&{get:?( int).&{bye:end , } , } , } and
  \oplus{sorry:end , }
are incompatible
```

Nested CST recursion

Consider the following source code of the CST process $\texttt{rec}\ X.a.\langle 5\rangle.\texttt{return}\ 5.X \mid \overline{a}.(x).(x)$:

```
new b in new a in (rec X.a?.5.return 5.X | a!.(x).(x) )
```

Running TypSes with the SCC option we obtain:

```
rec X.(accept aX(k2).catch k2 (k3).
   accept a(k4).k4!( 5).k3!( 5).
     request aX(k5).throw k5 (k4).0|X)
|request a(k1).k1?( x).k1?( x).0)
The process is typable!
Elapsed Time: 0.sec
```

The encoded process uses the replicated service `aX` and delegation in order to encode process definition.

CST Proxy

We report the code of a proxy server capable to invoke a new definition of the service `a` for each client request

```
(b?.rec X.(y).(a!.y.return 5>(x)>X)) /*proxy*/
|(a?.(x)) /*service*/
|(b!.rec X.5.X)) /*client*/
```

and then as expected

```
 ((accept b(k6).new aX in
   Rec X.(accept aX(k7).catch k7 (k8).k8?( y).newS k9 in
   (request a(k11).k11!( y).k9^+!(5).0|k9^-?( x).
    request aX(k10).throw k10 (k8).0)|X)|accept a(k5).k5?( x).
    request b(k1).new aX in
    Rec X.(accept aX(k2).catch k2 (k3).k3!( 5).
    request aX(k4).throw k4 (k3).0|X))
The process is typable!
Elapsed Time: 0.sec
```

where we encoded the pipe using a new fresh session `k9` declared by means of the operator `newS`. `k9^+`, `k9^-` stay for the positive and the negative polarity of `k9`. Notice also that the encoded process is alpha renamed in order to have all names different.

π-calculus factorial

We report the code of the factorial written in π-calculus. We use the feature of TypSes that allows importing external functions. These functions have a simply functional type and their presence can be an easy addition also in the theory.

```
import sub:int,int->int;
import mul:int,int->int;

new fatt in (rec X.(fatt?(x,r).
   if x=1 then r!(0)
   else
     new r1 in
        (fatt!(sub(x,1),r1).r1?(res).r!(mul(res,x)))|X)
 | new r in (fatt!(5,r).r?(x)) )
```

We import (with the `import` construct) two functions in order to subtract (`sub`) and to multiply (`mul`) two integers. We use the second channel `r` to reply with the result of the factorial of `x`. Running TypSes with the simply typed π-calculus option we obtain:

```
 new fatt1 in (Rec X6.(accept fatt1(pi7).pi7?( x8 r9).
   if (x=1) then request r9(pi15).pi15!( 0).0
   else new r110 in request fatt1(pi11).pi11!( sub( x8 1) r110).
     accept r110(pi12).pi12?( res13).request r9(pi14).pi14!(mul( res13 x8))
     |X6)|
new r2 in request fatt1(pi3).pi3!( 5 r2).accept r2(pi4).pi4?( x5).0)
The process is typable!
Elapsed Time: 0.015sec
```

The only possible typing errors are due to the tuple length and sort mismatching. For example if we invoke the `fact` without the reply channel

```
import sub:int,int->int;
import mul:int,int->int;

new fatt in (rec X.(fatt?(x,r).
   if x=1 then r!(0)
   else
     new r1 in
       (fatt!(sub(x,1),r1).r1?(res).r!(mul(res,x)))|X)
 | (fatt!(5) )
```

and then

```
 Error protocols
    ?( int).end
 with
    ?( int [alpha7]).end
 are incompatible
```

or if we use a channel to send values with different types

```
import sub:int,int->int;
import mul:int,int->int;

new fatt in (rec X.(fatt?(x,r).
   if x=1 then r!(0)
   else
     new r1 in
      (fatt!(sub(x,1),r1).r1?(res).r!(mul(res,x)))|X)
 | new r in new r2 in (fatt!(r2,r).r?(x)) )
```

and then

```
Unification error cannot unify
  [alpha12]
with
  int
```

Bibliography

[1] L. Acciai and M. Boreale. A type system for client progress in a service-oriented calculus. In *Proc. of Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *LNCS*, pages 642–658. Springer, 2008. 179

[2] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. of POPL*, pages 104–118. ACM, 1991. 4, 23, 58

[3] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, et al. *Web Services Conversation Language (Wscl) 1.0*, Mar. 2002. `http://www.w3.org/TR/2002/NOTE-wscl10-20020314`. 199

[4] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *Proc. CONCUR*, LNCS. Springer, 2008. 178, 193

[5] K. Bhargavan, R. Corin, C. Fournet, and A. D. Gordon. Secure sessions for web services. *ACM Trans. Inf. Syst. Secur.*, 10(2):8, 2007. 199

[6] E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence assertions for process synchronization in concurrent communications. *J. Funct. Program.*, 15(2):219–247, 2005. 120, 132, 191, 192, 195

[7] E. Bonelli and A. B. Compagnoni. Multipoint session types for a distributed calculus. In *Proc. of TGC*, pages 240–256, 2007. 192

[8] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos,

and G. Zavattaro. SCC: A service centered calculus. In *Proc. of WS-FM*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006. 20, 21, 60, 61, 179

[9] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *Proc. of FMOODS*, pages 19–38, 2008. 12, 60, 61, 179, 196

[10] M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Inf. Comput.*, 120(2):279–303, 1995. 17

[11] M. Bravetti and G. Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundam. Inform.*, 89(4):451–478, 2008. 193

[12] A. Brogi, C. Canal, and E. Pimentel. Behavioural types and component adaptation. In *In Proc. of AMAST*, volume 3116 of *LNCS*, pages 42–56, 2004. 192

[13] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984. 14

[14] R. Bruni, R. De Nicola, M. Loreti, and L. G. Mezzina. Provably correct implementations of services. In *Proc. of TGC*, 2008. 200

[15] R. Bruni and L. G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *Proc. of AMAST*, volume 5140 of *LNCS*, pages 100–115. Springer, 2008. 13, 70, 169

[16] M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007. 193

[17] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Sock: a calculus for service oriented computing. In *Proc. of ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006. 193

[18] L. Caires and L. Cardelli. A spatial logic for concurrency (part ii). In *Proc. of CONCUR*, pages 209–225, London, UK, 2002. Springer-Verlag. 17

[19] L. Caires and L. Cardelli. A spatial logic for concurrency (part i). *Inf. Comput.*, 186(2):194–235, 2003. 17

[20] L. Caires, H. Viera, and J. Seco. The conversation calculus: a model of service oriented computation. In *Proc. of ESOP'08 Programming Languages and Systems*, LNCS, pages 269–283. Springer, 2008. 193

[21] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *Proc. of POPL*, pages 261–272. ACM, 2008. 193

[22] W. Cook, D. Kitchin, and J. Misra. A language for task orchestration and its semantic properties. In *Proc. of CONCUR*, volume 4137 of *LNCS*, pages 477–491. Springer, 2006. 200

[23] R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure implementations for typed session abstractions. In *Proc. of CSF*, pages 170–186, Washington, DC, USA, 2007. IEEE Computer Society. 193

[24] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984. 193

[25] R. Demangeon, D. Hirschkoff, N. Kobayashi, and D. Sangiorgi. On the complexity of termination inference for processes. In *Proc. of TGC*, LNCS, pages 140–155, 2007. 11

[26] M. Dezani-Ciancaglini, U. de' Liguoro, and N. Yoshida. On Progress for Structured Communications. In *Proc. of TGC*, volume 4912 of *LNCS*, pages 257–275. Springer, 2008. 72, 115, 168, 178, 193, 194, 195

[27] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *Proc. of ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006. 193, 200

[28] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *Proc. of ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006. 195

[29] N. Y. Dimitris Mostrous and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *Proc. of ESOP*, number 5502 in LNCS. Springer, 2009. To appear. 193

[30] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006. 193

[31] V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed: (functional pearl). In *Proc. of ICFP*, pages 221–231. ACM, 2000. 4, 23, 25, 58

[32] P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. Bass: boxed ambients with safe sessions. In *Proc. of PPDP*, pages 61–72. ACM, 2006. 191

[33] S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inform.*, 42(2):191–225, 2005. 12, 20, 22, 23, 26, 29, 58, 179, 190, 193, 195

[34] A. D. Gordon. A tutorial on co-induction and functional programming. In *Proc. of Glasgow functional programming workshop*, pages 78–95. Springer Workshops in Computing, 1994. 191

[35] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. In *Proc. of CONCUR*, volume 5201 of *LNCS*, pages 492–507, 2008. 153

[36] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proc. of ICALP*, pages 299–309, London, UK, 1980. Springer-Verlag. 17

[37] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. 3, 12, 20, 23, 61, 82, 116, 120, 168, 179, 195, 201, 203

[38] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, pages 273–284. ACM, 2008. 192

[39] R. Hu, N. Yoshida, and K. Honda. Session-based distributed programming in java. In *Proc. of ECOOP*, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag. 193, 200

[40] S. Hym and M. Hennessy. Adding recursion to dpi. *Theor. Comput. Sci.*, 373(3):182–212, 2007. 191

[41] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004. 11, 18

[42] D. Kikuchi and N. Kobayashi. Type-based verification of correspondence assertions for communication protocols. In *Proc. of APLAS*, pages 191–205, 2007. 191

[43] N. Kobayashi. Typical: Type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`. 18

[44] N. Kobayashi. Type systems for concurrent programs. In *Proc. of 10th Anniversary Colloquium of UNU/IIST*, pages 439–453, 2002. 20

[45] N. Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4):291–347, 2005. 11, 18, 191

[46] N. Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247, 2006. 191

[47] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999. 9, 187

[48] N. Kobayashi and D. Sangiorgi. A hybrid type system for lock-freedom of mobile processes. In *Proc. of CAV*, pages 80–93, Berlin, Heidelberg, 2008. Springer-Verlag. 11

[49] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the p-calculus. *Logical Methods in Computer Science*, 2(3), 2006. 11, 18, 191

[50] D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. In *Proc. of POPL*, pages 419–428. ACM, 1993. 4, 58

[51] I. Lanese, V. Vasconcelos, F. Martins, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM*, pages 305–314. IEEE Computer Society Press, 2007. 61, 193

[52] C. Laneve and L. Padovani. The must preorder revisited. In *Proc. of CONCUR*, volume 4703 of *LNCS*, pages 212–225. Springer, 2007. 193

[53] C. Laneve and L. Padovani. The pairing of contracts and session types. In *Proc. of Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *LNCS*, pages 681–700. Springer, 2008. 193

[54] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007. 193

[55] L. G. Mezzina. Typses: a tool for type checking session types. Tool available at `http://www.di.unipi.it/~mezzina/`. 10

[56] L. G. Mezzina. How to infer finite session types in a calculus of services and sessions. In *Proc. of COORDINATION*, volume 5052 of *LNCS*, pages 216–231. Springer, 2008. 13

[57] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982. 14

[58] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989. 14

[59] R. Milner. The polyadic pi-calculus: A tutorial. *Logic and Algebra of Specification*, 1993. 115

[60] R. Milner. *Communicating and Mobile Systems the Pi-Calculus*. Cambridge University Press, June 1999. 16, 157

[61] R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992. 14, 17

[62] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. of ICALP*, LNCS, pages 685–695, London, UK, 1992. Springer-Verlag. 116, 153

[63] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Proc. of LICS*, pages 376–385. IEEE Computer Society, 1993. 3, 27, 83

[64] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. 5, 9

[65] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007. 199

[66] J. A. Robinson. Logic and logic programming. *Commun. ACM*, 35(3):40–65, 1992. 47

[67] D. Sangiorgi. Pi-I: A symmetric calculus based on internal mobility. In *Proc. of TAPSOFT*, pages 172–186, 1995. 17, 183

[68] D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Inf.*, 33(1):69–97, 1996. 17

[69] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *Proc. of Foclasa*, volume 68 of *ENTCS*, pages 439–456. Elsevier, 2003. 29

[70] V. Vasconcelos and K. Honda. Principal typing-schemes in a polyadic -calculus, 1992. 18, 190

[71] V. T. Vasconcelos. Recursive types in a calculus of objects. *Transactions of Information Processing Society of Japan*, 35(9):1828–1836, Sept. 1994. 190

[72] V. T. Vasconcelos. Unification of kinded infinite trees. *Information Processing Letters*, 55(6):323 – 328, 1995. 191

[73] V. T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Proc. of ISOTAS*, LNCS, pages 460–474. Springer-Verlag, 1993. 190, 191

[74] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informatic*, 1987. 191

[75] T. Woo and S. Lam. A semantic model for authentication protocols. *Research in Security and Privacy, 1993 IEEE Computer Society Symposium on*, pages 178–194, May 1993. 191

[76] N. Yoshida and V. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Elect. Notes in Th. Comput. Sci.*, 171(4):73–93, 2007. 115, 116, 118, 121, 124, 125, 131, 157, 158, 166, 168, 181, 186, 193, 195, 197