XPi: a Typed Process Calculus for XML Messaging*

Lucia Acciai Michele Boreale

Dipartimento di Sistemi e Informatica, Università di Firenze

{lacciai;boreale}@dsi.unifi.it

Abstract

We present XPi, a core calculus for XML messaging. XPi features asynchronous communication, pattern matching, name and code mobility, integration of static and dynamic typing. In XPi, a type system disciplines XML message handling at the level of channels, patterns, and processes. A run-time safety theorem ensures that in well-typed systems no service will ever receive documents it cannot understand, and that the offered services will be consistent with the declared channel capacities. An inference system is introduced, which is proved to be in full agreement with type checking. A notion of barbed equivalence is defined that takes into account information about service interfaces. Flexibility and expressiveness of this calculus are illustrated by a number of examples, some concerning description and discovery of web services.

1 Introduction

The explosive growth of the Web has led to the widespread use of *de facto* standards for naming schemes (URI, URL), communication protocols (SOAP, HTTP, TCP/IP) and message formats (XML). These three components are at the basis of the Web Services technology (WS, [35]), which underlies important application scenarios, like business-to-business applications [7]. The resulting architectural and programming paradigm, sometimes referred to as *Service Oriented Computing*, is centered around XML-message passing. Major reasons for the emergence of message-passing are its conceptual simplicity, its neutrality with respect to back-ends and platforms of services [8] and, of course, the widespread availability of effective message-oriented middleware [33, 19].

It is generally recognized that some of the proposed languages and standards for WS draw their inspiration from the π-calculus [27] , which conveys the message-passing paradigm in a distilled form (see also [24]). However, until recently, there was a significant gap between theory (formal models and analysis techniques) and practice (programming). One could find standards like WSDL [34], apt to describe service interfaces but saying very little about behaviour, or, at the other extreme, languages like BPEL4WS [4], oriented to detailed descriptions of services, but hardly amenable to formal analysis.

The situation has changed in the last couple of years, with a number of proposals based on process calculi, that lay the basis for formal specification and analysis of WS-compliant applications at diverse levels of abstractions. These proposals and their relationships to our work are discussed in the final section.

*This paper is an extended and revised version of [2]. The work reported in this paper was carried out while the first author was with the Laboratoire d'Informatique Fondamentale de Marseille, Université de Provence, as a PhD student. Work partially supported by the EU within the FET-GC2 initiative, project SENSORIA.

In this paper, we place ourselves at a somewhat basic level to give a concise semantic account of XML messaging and of the related typing issues. To this purpose, we present XPi, a process language based on the asynchronous π-calculus. Prominent features of XPi are: patterns generalizing ordinary inputs, ML-like pattern matching, and integration of static and dynamic typing. Our objective is to study issues raised by these features in connection with name and code mobility. A more precise account of our work and contributions follows.

For the sake of simplicity, syntax and reduction semantics of XPi are first introduced in an untyped setting (Section 2). In XPi, resource addresses on the net are represented as *names*, which can be generally understood as channels at which services are listening. *Messages* passed around are XML documents, represented as tagged/nested lists, in the vein of XDuce [22, 23]. Services and their clients are *processes*, that may send messages to channels, or query channels to retrieve messages obeying given patterns. Messages may contain names, which are passed around with only the *output capability* [30]. Practically, this means that a client receiving a service address cannot use this address to re-define the service. This assumption is perfectly sensible, simplifies typing issues, and does not affect expressive power (see e.g. [9, 25]). Messages may also contain mobile code in the form of *abstractions*, roughly, functions that take some argument and yield a process as a result. More precisely, abstractions can consume messages through pattern matching, thus supplying actual parameters to the contained code and starting its execution. This mechanism allows for considerable expressiveness. For example, we show that it permits a clean encoding of encryption primitives, hence of the spi-calculus [1], into XPi.

Types (Section 3) discipline processing of messages at the level of channels, patterns, and processes. At the time of its creation, each channel is given a *capacity*, i.e. a type specifying the format of messages that can travel on that channel. *Subtyping* arises from the presence of star types (arbitrary length lists) and union types, and by lifting at the level of messages a subtyping relation existing on basic values. The presence of a top type $\mathbf{T}$ enhances flexibility, allowing for such types as "all documents with an external tag f, containing a tag g and something else", written $\mathsf{T} = f[g[\mathbf{T}], \mathbf{T}]$. Subtyping is contravariant on channels: this is natural if one thinks of services, roughly, as functions receiving their arguments through channels. Contravariance calls for a bottom type $\mathbf{\bot}$, which allows one to express such sets of values as "all channels that can transport documents of some type $\mathsf{S} < \mathsf{T}$", written $ch(f[g[\mathbf{\bot}], \mathbf{\bot}])$. Abstractions that can safely consume messages of type T are given type $(\mathsf{T})\mathsf{Abs}$. Interplay between pattern matching, types, and capacities raises a few interesting issues concerning *type safety* (Section 4). Stated in terms of services accessible at given channels, our run-time safety theorem ensures that in well-typed systems, first, no service will ever receive documents it cannot understand, and second, that the offered service will comply with the statically declared capacities. The first property simply means that no process will ever output messages violating channel capacities. The second property means that no service will hang due to an input pattern (hence a type of incoming messages) that is not consistent with the channel's capacity – a form of "pattern consistency". Note that this property holds despite the fact that input patterns can partially be defined at run-time. Type checking is entirely static, in the sense that no run-time type check is required. A simple type-inference algorithm can be derived from type-checking (Section 5).

Our type system is partially inspired by XSD [20], but it is less rich than, say, the language of [12]. In particular, we have preferred to omit recursive types. While certainly useful in a full blown language, recursion would raise technicalities that hinder issues concerning name and code mobility. Also, our pattern language is quite basic, partly for similar reasons of simplicity, partly because more sophisticated patterns can be easily encoded.

The calculus described so far enforces a strictly static typing discipline. We also consider an extension of this calculus with *dynamic abstractions* (Section 6), which are useful when little or nothing is known about the actual types of incoming messages. Run-time type checks ensure that substitutions arising from pattern matching respect the types statically assigned to variables. Run time safety carries over. We argue that dynamic abstractions, combined with code mobility and subtyping, can provide linguistic support to such tasks as publishing and querying services:

indeed, we show that, relying on these features, dedicated primitives for publishing and discovering services can be easily encoded into XPi (Section 7).

A *behavioural equivalence* based on barbed bisimulation [31] is introduced (Section 8). This equivalence takes into account both type information and the presence of an input interface. The underlying idea is that systems come equipped with an interface, i.e. a set of input channels at which services are offered; on these channels external observers do not have the input capability.

There have been a number of proposals for integrating XML manipulation primitives into statically typed languages. We conclude (Section 9) with some discussion on recent related work in this field, and with a few directions for future extensions. Appendices A, B, D and E report the most lengthy or technical proofs, while Appendix F contains a somewhat more concrete example of service composition in XPi.

2 Untyped XPi

This section presents syntax and reduction semantics of untyped XPi, and a few derived constructs.

2.1 Syntax

We assume a countable set of *variables* $\mathcal{V}$, ranged over by *x, y, z, ...*, a set of *tags* $\mathcal{T}$, ranged over by *f,g,...*, and a set of *basic values* $\mathcal{BV}$ ranged over by *v, w, ...*. We leave $\mathcal{BV}$ unspecified (it might contain such values as integers, strings, or Java objects), but assume that $\mathcal{BV}$ contains a countable set of *names* $\mathcal{N}$, ranged over by *a, b, c, ...*. $\mathcal{N}$ is partitioned into a family of countable sets called *sorts* $\mathcal{S}, \mathcal{S}', \ldots$. We let u range over $\mathcal{N} \cup \mathcal{V}$ and let $\tilde{x}, \ldots$ denote a tuple of variables.

Definition 1 (messages, patterns and processes) *The set $\mathcal{M}$ of* XPi messages $M, N, \ldots$, *the set $\mathcal{Q}$ of* XPi patterns $Q, Q', \ldots$ *and the set $\mathcal{P}$ of* XPi processes $P, R, \ldots$ *are defined by the syntax in Table 1. In $Q_{\tilde{x}}$, we impose the following linearity condition: $\tilde{x}$ is a tuple of distinct names and each $x_i \in \tilde{x}$ occurs at most once in Q.*

In the style of XDuce [22, 23] and ℂDuce [5] XML documents are represented in XPi as tagged lists that can be arbitrarily nested; these are the messages being exchanged among processes. A message can be either a basic value, a variable, a tagged message, a list of messages, or an abstraction. The latter take the form $(Q_{\tilde{x}})P$, where variables $\tilde{x}$ represent formal parameters, to be replaced by actual parameters at run-time. A pattern is simply an abstraction-free message. For the sake of simplicity, we have ignored tag-variables that could be easily accommodated. Also, note that patterns do not allow for direct decomposition of documents into sublists (akin to the pattern `p, p'` in XDuce). The latter can be easily encoded though, as we show later in this section.

Process syntax is a variation on the π-calculus. In particular, asynchronous (non blocking) output on a channel u is written $\overline{u}\langle M \rangle$, and u is said to occur in *output subject position*. Nondeterministic guarded summation $\sum_{i \in I} a_i.A_i$ waits for any message matching A_i's pattern at channel a_i, for some $i \in I$ (I finite), consumes this message and continues as prescribed by A_i; names a_i are said to occur in *input subject position*. Note that the syntax forbids variables in input subject position, hence a received name cannot be used as an input channel; in other words, names are passed around with the output capability only (the motivation of this choice is discussed in Remark 2). Parallel composition $P_1|P_2$ represents concurrent execution of P_1 and P_2. Process P else R behaves like P, if P can do some internal reduction, otherwise reduces to R. This operator will be useful for coding up, e.g., if-then-else, without the burden of dealing with explicit negation on pattern. Replication $!P$ represents the parallel composition of arbitrarily many copies of P. Restriction $(\nu a)P$ creates a fresh name a, whose initial scope is P.

Message	$M ::=$	v	*Value*
		$\mid x$	*Var*
		$\mid f(M)$	*Tag*
		$\mid LM$	*List*
		$\mid A$	*Abstraction*
List of messages	$LM ::=$	$[]$	*Empty list*
		$\mid x$	*Var*
		$\mid M \cdot LM$	*Concatenation*
Abstraction	$A ::=$	$(Q_{\tilde{x}})P$	*Pattern and Continuation*
		$\mid x$	*Var*
Pattern	$Q ::=$	v	*Value*
		$\mid x$	*Var*
		$\mid f(Q)$	*Tag*
		$\mid LQ$	*List*
List of patterns	$LQ ::=$	$[]$	*Empty list*
		$\mid x$	*Var*
		$\mid Q \cdot LQ$	*Concatenation*
Process	$P ::=$	$\overline{u}\langle M \rangle$	*Output*
		$\mid \sum_{i \in I} a_i.A_i$	*Guarded Summation*
		$\mid P \text{ else } R$	*Else*
		$\mid P_1 \mid P_2$	*Parallel*
		$\mid !P$	*Replication*
		$\mid (\nu a)P$	*Restriction*

Table 1: Syntax of XPi messages, patterns and processes.

Binding conventions We stipulate that in every abstraction $(Q_{\tilde{x}})P$ the variables in $\tilde{x}$ bind with scope P, and that in each restriction $(\nu a)P$ name a binds with scope P. Accordingly, notions of alpha-equivalence ($=_\alpha$), free and bound names ($\mathsf{fn}(\cdot)$ and $\mathsf{bn}(\cdot)$), free and bound variables ($\mathsf{fv}(\cdot)$ and $\mathsf{bv}(\cdot)$) arise as expected for both messages and processes. We assume that $=_\alpha$ is sort-respecting, in the sense that a bound name can be α-renamed only to a name of the same sort. Whenever needed, we shall implicitly assume all binding occurrences of names (resp. variables) are distinct and disjoint from free names (resp. variables). We let $\mathcal{M}_{cl}$ be the set of closed messages and $\mathcal{P}_{cl}$ be the set of closed processes.

Notations The following abbreviations for messages and patterns are used: $[M_1, M_2, \ldots, M_{k-1}, M_k]$ stands for $M_1 \cdot (M_2 \cdot (\ldots (M_{k-1} \cdot (M_k \cdot [])) \ldots))$, while $f[M_1, \ldots, , M_k]$ stands for $f([M_1, \ldots, M_k])$. The

following abbreviations for processes are used: $\mathbf{0}$, $a_1.A_1$ and $a_1.A_1 + a_2.A_2 + \cdots + a_n.A_n$ stand for $\sum_{\{i \in I\}} a_i.A_i$ when $|I| = 0$, $|I| = 1$, and $|I| = n$, respectively; $(\nu a_1, \ldots, a_n)P = (\nu \widetilde{a})P$ stands for $(\nu a_1) \ldots (\nu a_n)P$. We sometimes save on subscripts by marking binding occurrences of variables in abstractions by a "?" symbol, or by replacing a binding occurrence of a variable by a don't care symbol, "_", if that variable does not occur in the continuation process. E.g. $([f[?x], g[\_]])P$ stands for $([f[x], g[y]]_{\{x,y\}})P$ where $y \notin \mathrm{fv}(P)$.

Our list representation of XML ignores algebraic properties of concatenation (such as associativity, see [23]). We simply take for granted some translation from actual XML documents to our syntax. The following example illustrates informally what this translation might look like.

Example 1 An XML document encoding an address book (on the left) and its representation in XPi (on the right)[1]:

```
<addrbook>                          addrbook[
 <person>                             person[
     <name>John Smith</name>              name("John Smith"),
     <tel>12345</tel>                     tel(12345),
     <emailaddrs>                         emailaddrs[
       <email>john@smith</email>             email("john@smith"),
       <email>smith@john</email>             email("smith@john")
     </emailaddrs>                        ]
 </person>                             ],
 <person>                             person[
     <name>Eric Brown</name>              name("Eric Brown"),
     <tel>678910</tel>                    tel(678910),
     <emailaddrs></emailaddrs>            emailaddrs[]
 </person>                             ]
</addrbook>                         ]
```

Note that in XPi a sequence of tagged documents such as `<tag1>M</tag1><tag2>N</tag2>`$\cdots$ is rendered as a list `[tag1(M), tag2(N),`$\cdots$`]`. A pattern, which extracts name and telephone number of the first person of the address book above, is: $Q_{xy} =$ `addrbook[ person[name(?x),tel(?y),_],_]`.

2.2 Reduction Semantics

A *reduction relation* describes system evolution via internal communications. Following [26], XPi reduction semantics is based on *structural congruence* $\equiv$, defined as the least congruence on processes satisfying the laws in Table 2. As it is usually the case, structural congruence permits certain rearrangements of parallel composition, replication, and restriction. Structural congruence extends to abstractions, hence to messages, in the expected manner. The reduction semantics also relies on a standard matching predicate, that matches a (linear) pattern against a closed message and yields a substitution.

Definition 2 (substitutions and matching) Substitutions $\sigma, \sigma', \ldots$ *are finite partial maps from the set* $\mathcal{V}$ *of variables to the set* $\mathcal{M}_{cl}$ *of closed messages. We denote by* ε *the empty substitution. For any term* t, $t\sigma$ *denotes the result of applying* σ *onto* t *(with alpha-renaming of bound names and variables if needed.) Let* M *be a closed message and* Q *be a linear pattern:* $\mathsf{match}(M, Q, \sigma)$ *holds true if and only if* $\mathrm{dom}(\sigma) = \mathrm{fv}(Q)$ *and* $Q\sigma = M$*; in this case, we also say that* M matches Q.

Definition 3 (reduction) *The reduction relation,* $\rightarrow \subseteq \mathcal{P}_{cl} \times \mathcal{P}_{cl}$*, is the least binary relation on closed processes satisfying the rules in Table 3.*

[1] We shall prefer the typewriter font whenever useful to improve on readability.

$$
\begin{array}{rclrcl}
P =_\alpha R \Rightarrow P & \equiv & R & P|R & \equiv & R|P \\
(P|R_1)|R_2 & \equiv & P|(R_1|R_2) & P|\mathbf{0} & \equiv & P \\
!P & \equiv & P|!P & (\nu a)(P|R) & \equiv & P|(\nu a)R \quad \text{if} \quad a \notin \mathrm{fn}(P) \\
(\nu a)\mathbf{0} & \equiv & \mathbf{0} & (\nu a)(\nu b)P & \equiv & (\nu b)(\nu a)P
\end{array}
$$

Table 2: Structural congruence.

$$
(\textsc{com}) \quad \frac{j \in I \quad a_j = a, \quad A_j = (Q_{\tilde{x}})P, \quad \mathrm{match}(M, Q, \sigma)}{\overline{a}\langle M \rangle \mid \sum_{i \in I} a_i.A_i \rightarrow P\sigma}
$$

$$
(\textsc{struct}) \quad \frac{P \equiv P', \quad P' \rightarrow R', \quad R' \equiv R}{P \rightarrow R} \qquad (\textsc{ctx}) \quad \frac{P \rightarrow P'}{(\nu \tilde{a})(P|R) \rightarrow (\nu \tilde{a})(P'|R)}
$$

$$
(\textsc{else}_1) \quad \frac{P \rightarrow P'}{P \text{ else } R \rightarrow P'} \qquad (\textsc{else}_2) \quad \frac{P \not\rightarrow}{P \text{ else } R \rightarrow R}
$$

Table 3: Reduction semantics.

A few words on the semantics of the else operator are in order. P else R behaves like P only if P can perform some internal actions, otherwise reduces to R. This semantics allows for the coding of if-then-else and similar constructs (see the Case defined below), without the need of introducing a (burdensome) explicit negation in patterns. Note that we forbid interaction of either branches of the else with the environment: e.g., in $\overline{a}v \mid \big(a.(Q)P \text{ else } R\big)$, we do not allow the left component of the else to consume the output on channel a. Indeed, by allowing that we would grant processes with the ability of atomically detecting presence/absence of messages on channels: this ability is unrealistic in a distributed setting. Moreover, it would have an impact on the observational semantics of the calculus difficult to assess.

Example 2 Consider the message M and the pattern Q_{xy} defined in Example 1, according to (COM):

$$
\begin{array}{c}
\overline{a}\langle M \rangle \mid a.(Q_{xy})(\overline{b}\langle [\mathtt{name}(x), \mathtt{tel}(y)] \rangle \mid P) \\
\rightarrow \\
\overline{b}\langle [\mathtt{name}(\text{“}\mathtt{JohnSmith}\text{”}), \mathtt{tel}(12345)] \rangle \mid (P[\text{“}\mathtt{JohnSmith}\text{”}/x, 12345/y]) \ .
\end{array}
$$

2.3 Derived constructs and examples

XPi allows for straightforward definition of a few powerful constructs, that will be used in later examples. In the following, we shall freely use recursive definitions of processes, that can be coded up using replication [26].

Application. A functional-like application for abstractions, $A \bullet M$, can be defined as $(\nu c)(\overline{c}\langle M \rangle | c.A)$, for any $c \notin \mathrm{fn}(M,A)$.

Case. A pattern matching construct relying on a *first match* policy, written

$$
\text{Case } M \text{ of } \quad (Q_1)_{\tilde{x}_1} \Rightarrow P_1, \ (Q_2)_{\tilde{x}_2} \Rightarrow P_2, \cdots, (Q_k)_{\tilde{x}_k} \Rightarrow P_k
$$

evolves into P_1 if M matches Q_1 (with substitutions involved), otherwise evolves into P_2 if M matches Q_2, and so on; if there is no match, the process is stuck. This construct can be defined in XPi as follows (assuming precedence of $\bullet$ on else and right-associativity for else):

$$(Q_1)_{\tilde{x}_1} P_1 \bullet M \quad \text{else} \quad (Q_2)_{\tilde{x}_2} P_2 \bullet M \quad \text{else} \quad \cdots \quad \text{else} \quad (Q_k)_{\tilde{x}_k} P_k \bullet M\,.$$

Example 3 Consider the message M defined in Example 1. Suppose that we want to extract and send along b the name of all persons that have at least one email address, and along c the name of all persons that do not have an email. Assume M is available on channel a. A process that performs this task is: $a.(\texttt{addrbook}[?x])R(x)$, where $R(x)$ is:

$$\begin{aligned} R(x) = \text{Case } x \text{ of} \quad & \texttt{person}[\texttt{name}(?y), \_, \texttt{emailaddrs}[\texttt{email}(\_), \_]] \cdot ?w \quad \Rightarrow \overline{b}\langle y\rangle \,|\, R(w) \\ & \texttt{person}[\texttt{name}(?z), \_] \cdot ?j \quad \Rightarrow \overline{c}\langle z\rangle \,|\, R(j). \end{aligned}$$

Decomposition and list processing. A process that attempts to *decompose* a message M into two sublists that satisfy the patterns $Q_{\tilde{x}}$ and $Q'_{\tilde{y}}$ and proceeds like P (with substitutions for $\tilde{x}$ and $\tilde{y}$ involved), if possible, otherwise is stuck, written:

$$M \text{ as } Q_{\tilde{x}}, Q'_{\tilde{y}} \Rightarrow P$$

can be defined as the recursive process $R(M)$, where:

$$\begin{aligned} R(x) = \text{Case } [] \text{ of} \quad & Q_{\tilde{x}} \Rightarrow \text{Case } x \text{ of} \quad Q'_{\tilde{y}} \Rightarrow P \\ & \qquad\qquad\qquad\quad \_ \Rightarrow R'([[],x]) \\ & \_ \Rightarrow R'([[],x]) \\ R'([l,x]) = \text{Case } x \text{ of } ?y \cdot ?w \Rightarrow (\,& \text{Case } l@y \text{ of } Q_{\tilde{x}} \Rightarrow (\,\text{Case } w \text{ of} \quad Q'_{\tilde{y}} \Rightarrow P, \\ & \qquad\qquad\qquad\qquad\quad \_ \Rightarrow R'([l@y,\, w])\,), \\ & \qquad\qquad\quad \_ \Rightarrow R'([l@y, w])\,). \end{aligned}$$

Here we have used a list-append function @, which can be easily defined via a call to a suitable recursive process.

Example 4 Consider

$$M = [\texttt{int}(1), \texttt{int}(2), \texttt{int}(3), \texttt{char}(\text{“a”}), \texttt{char}(\text{“b”}), \texttt{char}(\text{“c”})]$$

And the patterns: $Q_x = ?x$ and $Q'_{yw} = \texttt{char}(?y) \cdot ?w$. Then

$$\begin{aligned} & \overline{a}\langle M\rangle \;\mid\; a.(?z)\, z \text{ as } Q_x,\, Q'_{yw} \;\Rightarrow\; \overline{b}\langle x\rangle \,|\, \overline{c}\langle \texttt{char}(y) \cdot w\rangle \\ & \qquad\qquad \longrightarrow^* \\ & \overline{b}\langle[\texttt{int}(1), \texttt{int}(2), \texttt{int}(3)]\rangle \,|\, \overline{c}\langle[\texttt{char}(\text{“a”}), \texttt{char}(\text{“b”}), \texttt{char}(\text{“c”})]\rangle \end{aligned}$$

A process that, from a list LM, generates another list containing all messages of the original list satisfying a certain (closed) pattern Q, assigns this list to a variable y and proceeds like P:

$$\text{let } y = \text{map } Q, LM \text{ in } P$$

can be defined as the process $R([[],LM])$, where the following recursive definition is assumed:

$$\begin{aligned} R([l,x]) = \quad & \text{Case } x \text{ of} \quad ?z \cdot ?w \quad \Rightarrow (\,\text{Case } z \text{ of} \quad Q \quad \Rightarrow R([z \cdot l, w]), \\ & \qquad\qquad\qquad\qquad\qquad\qquad\quad \_ \quad \Rightarrow R([l,w])\,), \\ & \qquad\quad \_ \quad \Rightarrow P[l/y]. \end{aligned}$$

Example 5 Consider the message M of Example 1, available at a. Here is a process that consumes M, creates a list of all persons that have at least one email address and sends this list along b:

$$a.(\texttt{addrbook}[?x])(\text{let } y = \text{map } \texttt{person}[\_,\_,\texttt{emailaddrs}[\texttt{email}(\_),\_]],x \text{ in } \overline{b}\langle y\rangle)$$

Most common list manipulation constructs can be easily coded up in this style. We shall not pursue this direction any further.

Example 6 (a web service) Consider a web service WS that offers two different operations:

- an audio streaming service, offered at channel *stream*;
- a player download service, offered at channel *download*.

Clients that request the first kind of service must specify a streaming channel and its bandwidth ("`high`" or "`low`"), so that WS can stream one of two files ($\texttt{v}_{\texttt{low}}$ and $\texttt{v}_{\texttt{high}}$), as appropriate. Clients that request to download must specify a channel at which the player will be received. A client can run the downloaded player locally, supplying it appropriate parameters (a local streaming channel and its bandwidth). We represent streaming on a channel simply as an output action along that channel:

$$\begin{aligned} WS \stackrel{\triangle}{=} !\Big(\quad & \mathit{stream}.(\texttt{req\_stream}[\texttt{bandwidth}(\text{“}\texttt{low}\text{”}),\texttt{channel}(?x)])\overline{x}\langle \texttt{v}_{\texttt{low}}\rangle \\ & + \mathit{stream}.(\texttt{req\_stream}[\texttt{bandwidth}(\text{“}\texttt{high}\text{”}),\texttt{channel}(?y)])\overline{y}\langle \texttt{v}_{\texttt{high}}\rangle \\ & + \mathit{download}.(\texttt{req\_down}(?z))\overline{z}\langle \mathit{Player}\rangle \quad \Big). \end{aligned}$$

Player is an abstraction:

$$\mathit{Player} \stackrel{\triangle}{=} (\texttt{req\_stream}[\texttt{bandwidth}(?y'),\texttt{channel}(?z')])\Big(\text{Case } y' \text{ of } \text{“}\texttt{low}\text{”} \Rightarrow \overline{z'}\langle \texttt{v}_{\texttt{low}}\rangle \\ \text{“}\texttt{high}\text{”} \Rightarrow \overline{z'}\langle \texttt{v}_{\texttt{high}}\rangle \Big).$$

Note that the first two summands of WS are equivalent to *stream.Player*. However, the extended form written above makes a static optimization of channels possible (see Example 10). A client that asks for low bandwidth streaming, listens at s and then proceeds like C is:

$$C_1 \stackrel{\triangle}{=} (\nu s)(\overline{\mathit{stream}}\langle \texttt{req\_stream}[\texttt{bandwidth}(\text{“}\texttt{low}\text{”}),\texttt{channel}(s)]\rangle \,|\, s.(?v)C).$$

Another client that asks for download, then runs the player locally, listening at a local high bandwidth channel s is C_2 defined as:

$$\begin{aligned} C_2 \stackrel{\triangle}{=} (\nu d, s)\Big(& \overline{\mathit{download}}\langle \texttt{req\_down}(d)\rangle \\ & \mid d.(?x_p)(x_p \bullet \texttt{req\_stream}[\texttt{bandwidth}(\text{“}\texttt{high}\text{”}),\texttt{channel}(s)] \\ & \mid s.(?v)C)\Big). \end{aligned}$$

2.4 Expressiveness: encoding of encryption and decryption

Cryptographic primitives are sometimes used in distributed applications to guarantee secrecy and authentication of transmitted data. As a testbed for expressiveness of XPi, we show how to encode shared-key encryption and decryption primitives à la spi-calculus [1] into XPi. We first introduce XPi^{cr}, a cryptographic extension of XPi that subsumes shared-key spi-calculus, and then show how to encode XPi^{cr} into XPi. Message syntax is extended with the following clause, that represents encryption of M using N as a key:

$$M \quad ::= \quad \cdots \mid \{M\}_N \qquad \textit{(encryption)}$$

where N does not contain neither abstractions nor encryptions. Process syntax is extended with a case operator, that attempts decryption of M using N as a key and if successful binds the result to a variable x :

$$P \quad ::= \quad \cdots \mid \mathrm{case}\, M \,\mathrm{of}\, \{x\}_N \,\mathrm{in}\, P \qquad \textit{(decryption)}$$

where N does not contain neither abstractions nor encryptions, M is a variable or a message of the form $\{M'\}_{N'}$ and x binds in P. Patterns remain unchanged, in particular they may not contain encryptions nor abstractions. The additional reduction rule is:

$$(\textsc{Dec}) \quad \mathrm{case}\, \{M\}_N \,\mathrm{of}\, \{x\}_N \,\mathrm{in}\, P \to P[M/x].$$

Next, two translation functions, one for messages ($[\![\cdot]\!]$) and one for processes ($(\!|\cdot|\!)$), are defined from $\mathrm{XPi}^{\mathrm{cr}}$ to XPi. The translations of messages follow a familiar continuation-passing style. The relevant clauses of the definition, by structural induction, are as follows (on the others the functions just go through the structure of terms):

$$\begin{array}{rcl} [\![u]\!] & = & u \\ [\![\{M\}_N]\!] & = & ([N, ?x])\,\bar{x}\langle [\![M]\!] \rangle \\ \\ (\!|\bar{u}\langle M\rangle|\!) & = & \bar{u}\langle [\![M]\!] \rangle \\ (\!|\mathrm{case}\, M \,\mathrm{of}\, \{x\}_N \,\mathrm{in}\, P|\!) & = & (\nu\, r)\,([\![M]\!] \bullet [N, r] \mid r.(?x)\,(\!|P|\!)). \end{array}$$

Before proving the correctness of the encoding, we need to introduce some preliminary definitions. Following [31], let us define the *observation predicate (barb)* $P \downarrow_a$, which detects the possibility for P of immediately interacting along port a. Being in an asynchronous setting, we restrict our attention to output ports (see e.g. [3]). Thus, in XPi, $P \downarrow_a$ holds true if P has an output action $\bar{a}\langle M\rangle$, for some M, which is not in the scope of another prefix, or of (νa) or of an else operator; $P \Downarrow_a$ means that for some P', $P \to^* P'$ and $P' \downarrow_a$. Below, $P \hat{\to} P'$ stands for either $P \to P'$ or $P = P'$. In the following we define a barbed equivalence and a barbed expansion preorder.

Definition 4 (barbed bisimulation) *A symmetric binary relation on closed processes is a* barbed bisimulation *if* $(P,R) \in \mathcal{R}$ *implies:*

- *whenever* $P \to P'$ *then there is* R' *such that* $R \to^* R'$ *and* $(P', R') \in \mathcal{R}$*;*
- *whenever* $P \downarrow_a$ *then* $R \Downarrow_a$*.*

Two processes P *and* R *are* barbed bisimilar*, written* $P \dot{\approx} R$*, if* $(P,R) \in \mathcal{R}$ *for some barbed bisimulation* $\mathcal{R}$*.*

Following [10], we obtain barbed equivalence by closing barbed bisimulation under static contexts (in π-calculus, one gets ordinary early bisimulation this way.)

Definition 5 (barbed equivalence) *Two processes* P_1 *and* P_2 *are* barbed equivalent*, written* $P_1 \approx P_2$*, if for each* $\widetilde{h}$ *and each* R *it holds that* $(\nu \widetilde{h})(P_1|R) \dot{\approx} (\nu \widetilde{h})(P_2|R)$*.*

In a similar vein, we define the *barbed expansion preoder*:

Definition 6 (barbed expansion preorder) $\dot{\lesssim}$ *is the largest preorder such that* $P \dot{\lesssim} R$ *implies:*

- *whenever* $P \to P'$ *then there is* R' *such that* $R \to^* R'$ *and* $P' \dot{\lesssim} R'$*;*
- *whenever* $R \to R'$ *then there is* P' *such that* $P \hat{\to} P'$ *and* $P' \dot{\lesssim} R'$*;*
- *whenever* $P \downarrow_a$ *it holds that* $R \Downarrow_a$ *and whenever* $R \downarrow_a$ *then* $P \downarrow_a$*.*

We say that a process R expands P, written $P \lesssim R$, if for each $\tilde{h}$ and for each P' it holds that $(\nu\tilde{h})(P|P') \dot{\lesssim} (\nu\tilde{h})(R|P')$.

Finally, for reasoning on the encoding, we introduce *barbed $\langle\!|\cdot|\!\rangle$-equivalence*, which is obtained by closing barbed bisimulation under contexts that respect the encoding, that is, that are encodings of XPi^{cr} contexts (cfr. e.g. [9]).

Definition 7 (barbed $\langle\!|\cdot|\!\rangle$-equivalence) *Two processes P_1 and P_2 are* barbed $\langle\!|\cdot|\!\rangle$-equivalent*, written $P_1 \approx_{\langle\!|\cdot|\!\rangle} P_2$, if for each $\tilde{h}$ and each R it holds that $(\nu\tilde{h})(P_1|\langle\!|R|\!\rangle) \dot{\approx} (\nu\tilde{h})(P_2|\langle\!|R|\!\rangle)$.*

The encoding defined above is correct, in the sense that it preserves reductions and barbs in both directions, as stated by the following results (the proofs are reported in Appendix A). This implies that secrecy is preserved when moving from XPi^{cr} to XPi, provided in the latter only contexts that respect the encoding are taken into account.

Proposition 1 *Let P be a closed process in XPi^{cr}.*

1. *if $P \to P'$ then $\langle\!|P|\!\rangle \to^* \langle\!|P'|\!\rangle$;*
2. *if $\langle\!|P|\!\rangle \to P'$ then $\exists P'' \in \text{XPi}^{\text{cr}}$ s.t. $P \to P''$ and $\langle\!|P''|\!\rangle \lesssim P'$;*
3. *$P \downarrow a$ implies $\langle\!|P|\!\rangle \Downarrow a$ and $\langle\!|P|\!\rangle \downarrow a$ implies $P \Downarrow a$.*

Theorem 1 *Let P be a closed process in XPi^{cr}. $P \dot{\approx} \langle\!|P|\!\rangle$.*

Corollary 1 *Let P_1 and P_2 be closed processes in XPi^{cr}. $P_1 \approx P_2$ if and only if $\langle\!|P_1|\!\rangle \approx_{\langle\!|\cdot|\!\rangle} \langle\!|P_2|\!\rangle$.*

3 A type system

In this section, we define a type system that disciplines messaging at the level of channels, patterns and processes in XPi. The system guarantees that well-typed processes respect channels capacities at runtime. In other words, services are guaranteed to receive only requests they can understand, and conversely, services offered at a given channel will be consistent with the type declared for that channel. XPi's type system draws its inspiration from, but is less rich than, XML-Schema [20]. Our system permits to specify types for basic values (such as string or int) and provides tuple types (fixed-length lists) and star types (arbitrary-length lists); moreover, it provides abstraction types for code mobility. For the sake of simplicity, we have omitted attributes and recursive types.

3.1 Message types and subtyping

We assume an unspecified set $\mathcal{BT}$ of *basic types* bt, bt',... that might include int, string, boolean, or even Java classes. We assume that $\mathcal{BT}$ contains a countable set of *sort names* in one-to-one correspondence with the sorts $\mathcal{S}, \mathcal{S}', \ldots$ of $\mathcal{N}$; by slight abuse of notation, we denote sort names by the corresponding sorts.

Definition 8 (types) *The set $\mathcal{T}$ of* types*, ranged over by T, S, ..., is defined by the syntax in Table 4.*

Note the presence of the union type T+T', that is the type of all messages of type T or T', and of the star type $*$T, that is the type of all lists of elements of type T. (T)Abs is the type of all abstractions that can consume messages of type T. Finally, note the presence of $\mathbf{T}$ and $\mathbf{\bot}$ types. $\mathbf{T}$ is simply the type of all messages. On the contrary, no message has type $\mathbf{\bot}$, but this type is extremely useful for the purpose of defining channel types, as we shall see below.

Types	$\mathsf{T} ::=$	bt	*Basic type* $(\mathsf{bt} \in \mathcal{BT})$
		$\mid \mathbf{T}$	*Top*
		$\mid \mathbb{1}$	*Bottom*
		$\mid f(\mathsf{T})$	*Tag* $(f \in \mathcal{F})$
		$\mid \mathsf{LT}$	*List*
		$\mid \mathsf{T}+\mathsf{T}$	*Union*
		$\mid (\mathsf{T})\mathsf{Abs}$	*Abstraction*
List types	$\mathsf{LT} ::=$	$[]$	*Empty*
		$\mid *\mathsf{T}$	*Star*
		$\mid \mathsf{T}\cdot\mathsf{LT}$	*Concatenation*

Table 4: Syntax of types.

Notation The following abbreviations for types are used: $[\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_{k-1}, \mathsf{T}_k]$ stands for $\mathsf{T}_1 \cdot (\mathsf{T}_2 \cdot (\ldots (\mathsf{T}_{k-1} \cdot (\mathsf{T}_k \cdot [])) \ldots))$, while $f[\mathsf{T}_1, \ldots, \mathsf{T}_k]$ stands for $f([\mathsf{T}_1, \ldots, \mathsf{T}_k])$.

Example 7 A type for address books, on the left (see message M in Example 1), and a type for all SOAP messages, consisting of an *optional* header and a body, enclosed in an envelope, on the right:

```
addrbook[ *person[ name(string),                    envelope[ [] + header(T),
                   tel(int),                                  body(T)
                   emailaddrs(*email(string))]]              ].
```

Next, we associate types with channels, or more precisely with sorts. This is done by introducing a "capacity" function.

Definition 9 *A* capacity *function is a surjective map from the set of sorts to the set of types.*

In the sequel, we fix a generic capacity function. We shall denote by $ch(\mathsf{T})$ a generic sort that is mapped to T. Note that, by surjectivity of the capacity function, for each type T there is a sort $ch(\mathsf{T})$. In particular, $ch(\mathbf{T})$ is the sort of channels that can transport anything. In practice, determining capacity T of a given channel a, i.e. that a belongs to $ch(\mathsf{T})$, might be implemented with a variety of mechanisms, such as attaching to a an explicit reference to T's definition. We abstract away from these details.

List and star types and the presence of $\mathbf{T}$ and $\mathbb{1}$ naturally induce a subtyping relation. For example, a service capable of processing messages of type $\mathsf{T} = f(*\mathsf{int})$ must be capable of processing messages of type $\mathsf{T}' = f[\mathsf{int}, \mathsf{int}]$, i.e. T' is a subtype of T. Subtyping also serves to lift a generic subtyping preorder on basic types, $\prec$, to all types.

Definition 10 (subtyping) *The* subtyping relation $< \subseteq \mathcal{T} \times \mathcal{T}$ *is the least reflexive and transitive relation closed under the rules of Table 5.*

Note that we disallow subtyping on abstractions. The reason for this limitation will be discussed shortly after presenting the type checking system (see Remark 1). Also note that subtyping is contravariant on sorts capacities (rule (Sub-Sort)): this is natural if one thinks of a name of capacity T as, roughly, a function that can take arguments of type T. As a consequence of contravariance, for any T, we have $ch(\mathsf{T}) < ch(\mathbb{1})$, that is, $ch(\mathbb{1})$ is the type of all channels.

	(Sub-Sort) $\dfrac{\mathsf{T} < \mathsf{T}'}{ch(\mathsf{T}') < ch(\mathsf{T})}$		
(Sub-Top)	$\dfrac{}{\mathsf{T} < \mathbf{T}}$	(Sub-Bottom)	$\dfrac{}{\mathbf{\bot} < \mathsf{T}}$
(Sub-Basic)	$\dfrac{\mathsf{bt1} \prec \mathsf{bt2}}{\mathsf{bt1} < \mathsf{bt2}}$	(Sub-Tag)	$\dfrac{\mathsf{T}' < \mathsf{T}}{f(\mathsf{T}') < f(\mathsf{T})}$
(Sub-Star$_1$)	$\dfrac{}{[] < *\mathsf{T}}$	(Sub-Star$_2$)	$\dfrac{\mathsf{T}' < \mathsf{T}, \quad \mathsf{LT} < *\mathsf{T}}{\mathsf{T}' \cdot \mathsf{LT} < *\mathsf{T}}$
(Sub-Star$_3$)	$\dfrac{\mathsf{T}' < \mathsf{T}}{*\mathsf{T}' < *\mathsf{T}}$	(Sub-List)	$\dfrac{\mathsf{T}_1 < \mathsf{T}'_1, \quad \mathsf{LT} < \mathsf{LT}'}{\mathsf{T}_1 \cdot \mathsf{LT} < \mathsf{T}'_1 \cdot \mathsf{LT}'}$
(Sub-Union$_1$)	$\dfrac{\mathsf{T} < \mathsf{T}' \text{ or } \mathsf{T} < \mathsf{T}''}{\mathsf{T} < \mathsf{T}' + \mathsf{T}''}$	(Sub-Union$_2$)	$\dfrac{\mathsf{T}' < \mathsf{T}, \quad \mathsf{T}'' < \mathsf{T}}{\mathsf{T}' + \mathsf{T}'' < \mathsf{T}}$

Table 5: Rules for subtyping.

3.2 Type checking

A *basic typing* relation $v : \mathsf{bt}$ on basic values and basic types is presupposed, which is required to respect subtyping, i.e. whenever $\mathsf{bt} \prec \mathsf{bt}'$ and $v : \mathsf{bt}$ then $v : \mathsf{bt}'$. We further require that for each bt there is at least one $v : \mathsf{bt}$, and that for each v the set of bt's s.t. $v : \mathsf{bt}$ has a minimal element. On names, the basic typing relation is the following:

$$a : \mathcal{S} \text{ iff } a \in \mathcal{S}' \text{ for some } \mathcal{S}' < \mathcal{S} \ .$$

Contexts $\Gamma, \Gamma', \ldots$ are finite partial maps from variables $\mathcal{V}$ to types $\mathcal{T}$, sometimes denoted as sets of variable bindings $\{x_i : \mathsf{T}_i\}_{i \in I}$ (x_i's distinct). We denote the empty context by $\emptyset$. Let $\widetilde{x}$ be a set of variables; we denote by $\Gamma_{-\widetilde{x}}$ the context obtained from Γ by removing the bindings for the variables in $\tilde{x}$, and by $\Gamma_{|\tilde{x}}$ the context obtained by restricting Γ to the bindings for the variables in $\tilde{x}$. The subtyping relation is extended to contexts by letting $\Gamma_1 < \Gamma_2$ iff $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$ and $\forall x \in \mathsf{dom}(\Gamma_1)$ it holds that $\Gamma_1(x) < \Gamma_2(x)$. Union of contexts Γ_1 and Γ_2 having disjoint domains is written as $\Gamma_1 \cup \Gamma_2$ or as Γ_1, Γ_2 if no ambiguity arises. Sum of contexts Γ_1 and Γ_2 is written as $\Gamma_1 + \Gamma_2$ and is defined as $(\Gamma_1 + \Gamma_2)(x) = \Gamma_1(x) + \Gamma_2(x)$ if $x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)$, and $(\Gamma_1 + \Gamma_2)(x) = \Gamma_i(x)$ if $x \in \mathsf{dom}(\Gamma_i)$, for $i = 1, 2$, otherwise.

Type checking relies on a type-pattern matching predicate, $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$, whose role is twofold: (1) it extracts from T the types expected for the variables in Q after matching against messages of type T, yielding the context Γ, (2) it checks that Q is consistent with type T, i.e. that the type of Q is of a subtype of T under Γ.

Definition 11 (type-pattern match) *The predicate* $\mathsf{tpm}(T, Q, \Gamma)$ *is defined by the rules in Table 6.*

It is worth noticing that the condition $Q \neq x$ in rule (tpm-Top) is there just to enforce the use of (tpm-Var) in case $Q = x$ and $\mathsf{T} = \mathbf{T}$, so as to preserve the syntax-driven nature of the system. Rule (tpm-Union) deals with union types. Note in particular that if the pattern matches both components of the union, the resulting context will be given by the sum of the contexts produced by both matchings.

As expected, type checking works on an *annotated syntax* for processes and patterns, where each $Q_{\widetilde{x}}$ is decorated by a context Γ for its binding variables $\widetilde{x}$, written $Q_{\tilde{x}} : \Gamma$, with $\tilde{x} = \mathsf{dom}(\Gamma)$, or simply $Q : \Gamma$, where it is understood that the binding variables of Q are $\mathsf{dom}(\Gamma)$. For notational simplicity, we shall use such abbreviations as $a.(f[?x : \mathsf{T}, ?y : \mathsf{T}'])P$ for $a.(f[x,y] : \{x : \mathsf{T}, y : \mathsf{T}'\})P$, and

$$(\textsc{tpm-Top}) \quad \frac{Q \neq x}{\mathsf{tpm}(\mathbf{T}, Q, \Gamma)},\ \forall x \in \mathrm{fv}(Q):\ \Gamma(x) = \mathbf{T}$$

$$(\textsc{tpm-Empty}) \quad \frac{}{\mathsf{tpm}([],[],\emptyset)} \qquad (\textsc{tpm-Var}) \quad \frac{}{\mathsf{tpm}(\mathsf{T}, x, \{x:\mathsf{T}\})}$$

$$(\textsc{tpm-Value}) \quad \frac{v:\mathsf{bt}}{\mathsf{tpm}(\mathsf{bt}, v, \emptyset)} \qquad (\textsc{tpm-Tag}) \quad \frac{\mathsf{tpm}(\mathsf{T}, Q, \Gamma)}{\mathsf{tpm}(f(\mathsf{T}), f(Q), \Gamma)}$$

$$(\textsc{tpm-Star}_1) \quad \frac{}{\mathsf{tpm}(*\mathsf{T}, [], \emptyset)} \qquad (\textsc{tpm-Star}_2) \quad \frac{\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1),\ \mathsf{tpm}(*\mathsf{T}, LQ, \Gamma_2)}{\mathsf{tpm}(*\mathsf{T}, Q \cdot LQ, \Gamma_1 \cup \Gamma_2)}$$

$$(\textsc{tpm-List}) \quad \frac{\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1),\ \mathsf{tpm}(\mathsf{LT}, LQ, \Gamma_2)}{\mathsf{tpm}(\mathsf{T} \cdot \mathsf{LT}, Q \cdot LQ, \Gamma_1 \cup \Gamma_2)}$$

$$(\textsc{tpm-Union}) \quad \frac{\mathsf{tpm}(\mathsf{T}_0, Q, \Gamma_0) \quad \text{or} \quad \mathsf{tpm}(\mathsf{T}_1, Q, \Gamma_1)}{\mathsf{tpm}(\mathsf{T}_0 + \mathsf{T}_1, Q, \Gamma)}, \quad \text{where:}$$

$$\Gamma = \begin{cases} \Gamma_0 + \Gamma_1 & \text{if } \mathsf{tpm}(\mathsf{T}_0, Q, \Gamma_0) \text{ and } \mathsf{tpm}(\mathsf{T}_1, Q, \Gamma_1) \\ \Gamma_i & \text{if } \mathsf{tpm}(\mathsf{T}_i, Q, \Gamma_i) \text{ and for no } \Gamma' \ \mathsf{tpm}(\mathsf{T}_{i+1 \bmod 2}, Q, \Gamma'),\ i = 0, 1 \end{cases}$$

Table 6: Matching types and patterns.

$$(\textsc{tm-Empty}) \quad \frac{}{\Gamma \vdash [] : []} \qquad (\textsc{tm-Top}) \quad \frac{}{\Gamma \vdash M : \mathbf{T}}$$

$$(\textsc{tm-Value}) \quad \frac{v:\mathsf{bt}}{\Gamma \vdash v : \mathsf{bt}} \qquad (\textsc{tm-Var}) \quad \frac{\Gamma(x) < \mathsf{T}}{\Gamma \vdash x : \mathsf{T}}$$

$$(\textsc{tm-Tag}) \quad \frac{\Gamma \vdash M : \mathsf{T}}{\Gamma \vdash f(M) : f(\mathsf{T})} \qquad (\textsc{tm-List}) \quad \frac{\Gamma \vdash M : \mathsf{T}, \quad \Gamma \vdash LM : \mathsf{LT}}{\Gamma \vdash (M \cdot LM) \ : \ (\mathsf{T} \cdot \mathsf{LT})}$$

$$(\textsc{tm-Star}_1) \quad \frac{}{\Gamma \vdash [] : *\mathsf{T}} \qquad (\textsc{tm-Star}_2) \quad \frac{\Gamma \vdash M : \mathsf{T}, \quad \Gamma \vdash LM : *\mathsf{T}}{\Gamma \vdash (M \cdot LM) \ : \ *\mathsf{T}}$$

$$(\textsc{tm-Union}) \quad \frac{\Gamma \vdash M : \mathsf{T} \quad \text{or} \quad \Gamma \vdash M : \mathsf{T}'}{\Gamma \vdash M : \mathsf{T} + \mathsf{T}'}$$

$$(\textsc{tm-Abs}) \quad \frac{\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1), \quad (\Gamma_1)_{|\tilde{x}} < \Gamma_Q, \quad (\Gamma_1)_{|\tilde{y}} > \Gamma_{|\tilde{y}}, \quad \Gamma, \Gamma_Q \vdash P : ok}{\Gamma \vdash (Q : \Gamma_Q)P : (\mathsf{T})\mathsf{Abs}}$$

where $\tilde{x} = \mathsf{dom}(\Gamma_Q)$, $\tilde{y} = \mathrm{fv}(Q) \setminus \tilde{x}$ and $(\Gamma_1)_{|\tilde{y}}$ is abstraction-free

Table 7: Type system for messages.

assume that don't care variables "_" are always annotated with $\mathbf{T}$. Reduction semantics carries over to annotated closed processes formally unchanged.

In what follows, we shall use the following additional notation and terminology. We say that a type T is *abstraction-free* if T contains no subterms of the form $(\mathsf{T}')\mathsf{Abs}$. A context Γ is abstraction-free if for each $x \in \mathsf{dom}(\Gamma)$, $\Gamma(x)$ is abstraction-free. We use $\Gamma \vdash u \in ch(\mathsf{T})$ as an abbreviation for: either $u = a \in ch(\mathsf{T})$ or $u = x \in \mathcal{V}$ and $\Gamma(x) = ch(\mathsf{T})$.

The type checking system, defined on open terms, consists of two sets of inference rules, one for messages and one for processes, displayed in Table 7 and 8, respectively. These two systems are mutually dependent, since abstractions may contain processes, and processes may contain abstractions. Note that the system is entirely syntax driven, i.e. the process P (resp. the pair (M, T)) determines the rule that should be applied to check $\Gamma \vdash P : ok$ (resp. $\Gamma \vdash M : \mathsf{T}$).

The most interesting of these rules is (TM-Abs). Informally, $\Gamma \vdash A : (\mathsf{T})\mathsf{Abs}$ ensures that under Γ the following is true: (1) abstraction $A = (Q_{\tilde{x}} : \Gamma_Q)P$ behaves safely upon consuming messages

$$\text{(T-In)}\quad \frac{a \in ch(\mathsf{T}),\quad \Gamma \vdash A : (\mathsf{T})\mathsf{Abs}}{\Gamma \vdash a.A : ok}$$

$$\text{(T-Out)}\quad \frac{\Gamma \vdash u \in ch(\mathsf{T}),\quad \Gamma \vdash M : \mathsf{T}}{\Gamma \vdash \overline{u}\langle M\rangle : ok} \qquad \text{(T-Sum)}\quad \frac{\forall i \in I,\quad \Gamma \vdash a_i.A_i : ok \quad |I| \neq 1}{\Gamma \vdash \sum_{i\in I} a_i.A_i : ok}$$

$$\text{(T-Rep)}\quad \frac{\Gamma \vdash P : ok}{\Gamma \vdash !P : ok} \qquad \text{(T-Par)}\quad \frac{\Gamma \vdash P : ok,\quad \Gamma \vdash R : ok}{\Gamma \vdash (P|R) : ok}$$

$$\text{(T-Res)}\quad \frac{\Gamma \vdash P : ok}{\Gamma \vdash (\nu a)P : ok} \qquad \text{(T-Else)}\quad \frac{\Gamma \vdash P : ok,\quad \Gamma \vdash R : ok}{\Gamma \vdash P \text{ else } R : ok}$$

Table 8: Type system for processes.

of type T (because the type at which the actual parameters will be received is a subtype of the type declared for formal parameters, $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$, and because of $\Gamma, \Gamma_Q \vdash P : ok$); (2) the pattern Q is consistent with type T, i.e. essentially the run-time type of Q is a subtype of T (because of type-pattern match and of $\Gamma_{|\tilde{y}} < (\Gamma_1)_{|\tilde{y}}$). This guarantees existence of a message of type T that matches the pattern. Moreover, no ill-formed pattern will arise from Q (abstraction-freeness). Examples 8 and 9 further illustrate the premises of this rule.

Rule (T-In) checks that an abstraction A residing at channel $a \in ch(\mathsf{T})$ can safely consume messages of type T, and that there do exist messages of type T that match the pattern of A. Conversely (T-Out) checks that messages sent at u are of type T. Input and summation (rule (T-Sum)) are dealt with separately only for notational convenience. Finally, it is worth to notice that, by definition of $a : \mathcal{S}$, rule (TM-Value) entails subsumption on channels (i.e. $\Gamma \vdash a : \mathcal{S}$ and $\mathcal{S} < \mathcal{S}'$ implies $\Gamma \vdash a : \mathcal{S}'$.) The remaining rules should be self-explanatory.

In the sequel, for closed annotated processes P, we shall write $P : ok$ for $\emptyset \vdash P : ok$, and say that P is well-typed. Similarly for $M : \mathsf{T}$, for annotated closed M.

Example 8 (condition $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$ in TM-Abs) Consider the following process:

$$P = a.(?x : \mathsf{int})\overline{b}\langle x\rangle \,|\, \overline{a}\langle[1,2,3]\rangle \ .$$

Suppose $a \in ch(*\mathsf{int})$ and $b \in ch(\mathsf{int})$. If condition $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$ is omitted then process P above results well-typed, but the subject reduction property would be violated because at run-time we would have $\overline{b}\langle[1,2,3]\rangle$, which is ill-typed according to (T-Out).

Example 9 (condition $(\Gamma_1)_{|\tilde{y}} > \Gamma_{|\tilde{y}}$ in TM-Abs) Assume $a \in ch(*\mathsf{int})$ and $b \in ch(f[\mathsf{int}, *\mathsf{int}])$. Then $P : ok$, where:

$$P = a.(?y : *\mathsf{int})b.(f[?x : \mathsf{int}, y])\overline{a}\langle x \cdot y\rangle \mid \overline{a}\langle[4,5]\rangle \mid \overline{a}\langle[4,5,6]\rangle.$$

Suppose condition $(\Gamma_1)_{|\tilde{y}} > \Gamma_{|\tilde{y}}$ is omitted. The sort associated to b could be changed into $ch(f[\mathsf{int}, [\mathsf{int}, \mathsf{int}]])$ and process P would still result well-typed, but the subject reduction property would be violated because at run-time we could have $b.(f[?x : \mathsf{int}, [4,5,6]])\overline{a}\langle x \cdot [4,5,6]\rangle \mid \overline{a}\langle[4,5]\rangle$, which is ill-typed according to (T-In), because $f[?x : \mathsf{int}, [4,5,6]]$ is not consistent with the capacity associated to b, that is $f[\mathsf{int}, [\mathsf{int},\mathsf{int}]]$.

To illustrate the use of $ch(\mathbf{1})$, and contravariance on sort names, consider a "link process" ([9]) that constantly receives any *name* on a and sends it along b. This can be written as $!a.(?x : ch(\mathbf{1}))\overline{b}\langle x\rangle$. This process is well-typed provided $a \in ch(ch(\mathsf{T}))$, for some T, and that $b \in ch(ch(\mathbf{1}))$.

Remark 1 (on abstractions and subtyping) To see why we disallow subtyping on abstractions, consider the types $\mathsf{T} = [f(\mathsf{int}), f(\mathsf{int})]$ and $*f(\mathsf{int}) = \mathsf{T}'$. Clearly $\mathsf{T} < \mathsf{T}'$. Assume we had defined subtyping *covariant* on abstractions, so that $(\mathsf{T})\mathsf{Abs} < (\mathsf{T'})\mathsf{Abs}$. Now, clearly $A = (?x :$

$\mathsf{T})\mathbf{0} : (\mathsf{T})\mathsf{Abs}$, but *not* $A : (\mathsf{T}')\mathsf{Abs}$ (the condition $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$ of (TM-ABS) fails). In other words, the basic principle of subtyping (that a supertype types more terms than a subtype) would be violated.

On the other hand, assume we had defined subtyping *contravariant* on abstractions, so that $(\mathsf{T}')\mathsf{Abs} < (\mathsf{T})\mathsf{Abs}$. Consider $A' = (Q : \Gamma_Q)\mathbf{0}$, where $Q : \Gamma_Q = [f(?x : \mathsf{int}), f(?y : \mathsf{int}), f(?z : \mathsf{int})]$; clearly $A' : (\mathsf{T}')\mathsf{Abs}$, but *not* $A' : (\mathsf{T})\mathsf{Abs}$ (simply because there is no type-pattern match between T and Q, hence Q won't be T-consistent.) This would violate again the basic principle of subtyping and the safety property.

Remark 2 (on input locality) To illustrate input locality, assume we allowed a process to use a received name as input subject, like in P below

$$P = a.(?x : ch(\mathsf{S}))x.(?y : \mathsf{S})\overline{c}\langle y\rangle \mid \overline{a}\langle b\rangle \mid \overline{b}\langle[1,2,3]\rangle \ .$$

Assume $a : ch(ch(\mathsf{S}))$, $c : ch(\mathsf{S})$ and $b : ch(\mathsf{T})$ for $\mathsf{S} = [\mathsf{int}]$ and $\mathsf{T} = *\mathsf{int}$. By (SUB-STAR$_2$), $\mathsf{S} < \mathsf{T}$ and by (SUB-SORT) $ch(\mathsf{T}) < ch(\mathsf{S})$. Process P is well-typed by rules (T-PAR), (T-IN), (T-OUT) and (TM-ABS). But here a reduction violates the subject reduction property. In fact, $P \to P' \to P''$ with $P' = b.(?y : \mathsf{S})\overline{c}\langle y\rangle \mid \overline{b}\langle[1,2,3]\rangle$ and $P'' = \overline{c}\langle[1,2,3]\rangle$ and process P'' is not well-typed because $c : ch(\mathsf{S})$ but not $[1,2,3] : \mathsf{S}$. (Note that P' is not well-typed either, because $\mathsf{tpm}(\mathsf{T}, ?y, \{y : \mathsf{T}\})$ and not $\mathsf{S} > \mathsf{T}$ as required by (TM-ABS)).

3.3 Typing rules for Application and Case

The rules below can be easily derived from the translation of derived constructs application and case to the base syntax. In the following, we let $\mathsf{T}_{M,\Gamma}$ denote the *exact type* of M under Γ, obtained from M by replacing each x by $\Gamma(x)$, each name $a \in ch(\mathsf{T})$ by $ch(\mathsf{T})$, each other v by the least type bt s.t. $v : \mathsf{bt}$, and, recursively, each abstraction subterm $(Q : \Gamma_Q)P$ by $(\mathsf{T}_{Q,\,\Gamma\cup\Gamma_Q})\mathsf{Abs}$. The rule for application is:

$$\text{(T-APPL)} \quad \frac{\Gamma \vdash A : (\mathsf{T}_{M,\Gamma})\mathsf{Abs}}{\Gamma \vdash A \bullet M : ok}.$$

that is easily proven sound recalling that $A \bullet M = (\nu c)(c.A|\overline{c}\langle M\rangle)$ (c fresh), and assuming that c is chosen s.t. $c \in ch(\mathsf{T}_{M,\Gamma})$.

Concerning Case, first note that the typed version of this construct contemplates annotated patterns, thus:

$$\begin{aligned} \text{Case } M \text{ of} \quad & Q_1 : \Gamma_{Q_1} \Rightarrow P_1, \\ & Q_2 : \Gamma_{Q_2} \Rightarrow P_2, \\ & \quad\vdots \\ & Q_k : \Gamma_{Q_k} \Rightarrow P_k \,. \end{aligned}$$

Then, relying on the rule for application, the typing rule for case can be written as:

$$\text{(T-CASE)} \quad \frac{\forall i = 1,\ldots,k : \ \Gamma \vdash (Q_i : \Gamma_{Q_i})P_i \bullet M : ok}{\Gamma \vdash \text{Case } M \text{ of } Q_1 : \Gamma_{Q_1} \Rightarrow P_1, \ldots, Q_k : \Gamma_{Q_k} \Rightarrow P_k : ok}.$$

Example 10 (a web service, continued) Consider the service defined in Example 6. Assume a basic type stream of all files, such that $\mathtt{v_{low}}$, $\mathtt{v_{high}} : \mathsf{stream}$, and a basic type $\mathsf{low-stream}$ of low quality files, s.t. $\mathtt{v_{low}} : \mathsf{low-stream}$, but *not* $\mathtt{v_{high}} : \mathsf{low-stream}$. Assume $\mathsf{low-stream} < \mathsf{stream}$; note that this implies that $ch(\mathsf{stream}) < ch(\mathsf{low\text{-}stream})$, i.e. if a channel can be used for streaming generic files, it can also be used for streaming low-quality files, which fits intuition. Let T be $\mathtt{req\_stream}[\mathtt{bandwidth}(\mathsf{string}), \mathtt{channel}(ch(\mathsf{stream}))]$ and fix the following capacities for channels *stream* and *download*: $stream \in ch(\mathsf{T})$ and $download \in ch(\mathtt{req\_down}(ch((\mathsf{T})\mathsf{Abs})))$. An annotated version of *WS*, which permits in principle a static optimization of channels (assuming allocation of low-quality channels is less expensive than generic channels'):

$$
\begin{aligned}
WS = !\Big(& \mathit{stream}.(\texttt{req\_stream}[\texttt{bandwidth}(\text{“}\texttt{low}\text{”}), \texttt{channel}(?x : \mathit{ch}(\mathsf{low} - \mathsf{stream}))])\overline{x}\langle \mathtt{v}_{\mathtt{low}} \rangle \\
& + \mathit{stream}.(\texttt{req\_stream}[\texttt{bandwidth}(\text{“}\texttt{high}\text{”}), \texttt{channel}(?y : \mathit{ch}(\mathsf{stream}))])\overline{y}\langle \mathtt{v}_{\mathtt{high}} \rangle \\
& + \mathit{download}.(\texttt{req\_down}[?z : \mathit{ch}((\mathsf{T})\mathsf{Abs})])\overline{z}\langle \mathit{Player} \rangle \Big)
\end{aligned}
$$

where *Player* is the obvious annotated version of the player of Example 6. It is easy to check that $\mathit{Player} : (\mathsf{T})\mathsf{Abs}$ and that $WS : \mathit{ok}$.

4 Run-time safety

The safety property of our interest can be defined in terms of channel capacities, message types, and consistency. First, a formal definition of pattern consistency.

Definition 12 (T-consistency) *A type T is* consistent *if $\mathbb{L}$ does not occur in T. A pattern Q is* T-consistent *if there is a message $M : \mathsf{T}$ that matches Q.*

Note that all sort names, including $\mathit{ch}(\mathbb{L})$, are consistent types by definition. A safe process is one whose output and input actions are in agreement with channel capacities, as stated by the definition below. It is worth noticing that condition (2) guarantees accessibility of services. Of course, for input actions it makes sense to require consistency (condition (2)) only if the input channel capacity is consistent.

Definition 13 (safety) *Let P be an annotated closed process. P is* safe *if and only if for each name $a \in \mathit{ch}(T)$:*

1. *whenever $P \equiv (\nu\, \tilde{h})(\overline{a}\langle M \rangle \,|\, R)$ then $M : \mathsf{T}$;*
2. *suppose T is consistent. Whenever $P \equiv (\nu\, \tilde{h})(S \,|\, R)$, where S is a guarded summation, $a.A$ a summand of S and Q is A's pattern, then Q is T-consistent.*

A first, expected result about the type system is type safety which relies on the following lemma (omitted proofs are reported in Appendix B).

Lemma 1 *Suppose T is consistent. If* $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$ *for some Γ then Q is T-consistent.*

Theorem 2 (type safety) *Let P be an annotated closed process and suppose $P : \mathit{ok}$, then P is safe.*

Subject reduction relies on the following lemmas. The first lemma states that typing does respect the subtyping relation:

Lemma 2 (subtyping) *If $T' < T$ then for any M such that $\Gamma \vdash M : \mathsf{T}'$ we have $\Gamma \vdash M : \mathsf{T}$.*

The following lemma ensures, roughly, that type-pattern match agrees with message-pattern match. In particular, if a closed message of type T matches a pattern Q, then the values taken on by Q's variables after matching will be of the type predicted by tpm.

Lemma 3 (matching) *Let $M : \mathsf{T}$ be a closed message. If* $\mathsf{match}(M, Q, \sigma)$ *and* $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$ *then* $\forall x \in \mathsf{dom}(\sigma) : \sigma(x) : \Gamma(x)$.

The next lemmas ensure that typing is preserved by substitutions and structural congruence.

Lemma 4 (substitution) *(a) If $\Gamma, x : \mathsf{T} \vdash P : \mathit{ok}$ and $\Gamma \vdash M : \mathsf{T}$ then $\Gamma \vdash P[M/x] : \mathit{ok}$; (b) if $\Gamma, x : T \vdash N : \mathsf{S}$ and $\Gamma \vdash M : \mathsf{T}$ then $\Gamma \vdash N[M/x] : S$.*

Lemma 5 (structural congruence) *Let P and Q be annotated closed processes. If $P : ok$ and $P \equiv Q$ then $Q : ok$.*

Theorem 3 (subject reduction) *Let P be an annotated closed process. If $P : ok$ and $P \to P'$ then $P' : ok$.*

PROOF: By induction on the derivation of $P \to P'$. We distinguish the last reduction rule applied:

(COM) $\overline{a}\langle M\rangle | \sum_{i\in I} a_i.A_i \to P\sigma$ where, for some $j \in I$:

- $a = a_j$;
- $A_j = (Q_{\tilde{x}} : \Gamma_Q)P$;
- $\mathsf{match}(M, Q, \sigma)$.

We have to prove that $P\sigma : ok$. From $\overline{a}\langle M\rangle | \sum_{i\in I} a_i.A_i : ok$ and the premises of the rule (T-PAR), we deduce that $\overline{a}\langle M\rangle : ok$. By the latter and the premises of the rule (T-OUT), for some T:

- $a \in ch(\mathsf{T})$;
- $M : \mathsf{T}$.

Hence, from $\sum_{i\in I} a_i.A_i : ok$ and the premises of rules (T-SUM) and (T-IN), we deduce that $A_j : (\mathsf{T})\mathsf{Abs}$.

From this, and the premises of the rule(TM-ABS), we infer:

- $\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1)$, $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$;
- $\Gamma_Q \vdash P : ok$.

By Lemma 3 (matching), $M : \mathsf{T}$, $\mathsf{match}(M, Q, \sigma)$, and $\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1)$ we have $\forall x \in \mathsf{dom}(\sigma) : \sigma(x) : \Gamma_1(x)$, hence, by Lemma 2 (subtyping), $\sigma(x) : \Gamma_Q(x)$. In conclusion, by $\Gamma_Q \vdash P : ok$ and Lemma 4 (substitution) we have $P\sigma : ok$.

(STRUCT) by $P \to Q$ and the premises of the rule, we get $P \equiv P'$, $P' \to Q'$ and $Q' \equiv Q$. $P' : ok$ and Lemma 5 imply $Q' : ok$, and $Q' \equiv Q$ implies $Q : ok$;

(CTX) by $(\nu\tilde{a})(P|R) \to (\nu\tilde{a})(P'|R)$ and the premises of the rule, we get $P \to P'$. By the premises of the rule (T-RES) and $(\nu\tilde{a})(P|R) : ok$, we get $P|R : ok$, and by (T-PAR) $P : ok$ and $R : ok$. By induction $P \to P'$ and $P : ok$ implies $P' : ok$, that is $(\nu\tilde{a})(P'|R) : ok$ by (T-PAR) and (T-RES);

(ELSE$_1$) by P else $Q \to P'$ and the premises of the rule, we get $P \to P'$. By the premises of the rule (T-ELSE) and P else $Q : ok$, we get $P : ok$ and $Q : ok$. By induction $P \to P'$ and $P : ok$ imply $P' : ok$;

(ELSE$_2$) P else $Q \to Q$; by the premises of the rule (T-ELSE) and P else $Q : ok$, we get $P : ok$ and $Q : ok$.

□

As a consequence of subject reduction and type safety we get run-time safety.

Corollary 2 (run-time safety) *Let P be an annotated closed process. If $P : ok$ and $P \to^* P'$ then P' is safe.*

PROOF: By Theorem 2 (Type Safety) and 3 (Subject Reduction). □

$$(\text{TI}_M\text{-Empty})\quad \frac{}{\mathsf{ti}_\mathsf{M}([],[],\Gamma,\emptyset)} \qquad (\text{TI}_M\text{-Top})\quad \frac{}{\mathsf{ti}_\mathsf{M}(\mathsf{T},M,\Gamma,\Gamma')},\ \forall x\in \mathsf{bv}(M):\ \Gamma'(x)=\mathbb{L}$$

$$(\text{TI}_M\text{-Value})\quad \frac{v:\mathsf{bt}}{\mathsf{ti}_\mathsf{M}(\mathsf{bt},v,\Gamma,\emptyset)} \qquad (\text{TI}_M\text{-Var})\quad \frac{\Gamma(x)<\mathsf{T}}{\mathsf{ti}_\mathsf{M}(\mathsf{T},x,\Gamma,\emptyset)}$$

$$(\text{TI}_M\text{-Tag})\quad \frac{\mathsf{ti}_\mathsf{M}(\mathsf{T},M,\Gamma,\Gamma')}{\mathsf{ti}_\mathsf{M}(f(\mathsf{T}),f(M),\Gamma,\Gamma')} \qquad (\text{TI}_M\text{-List})\quad \frac{\mathsf{ti}_\mathsf{M}(\mathsf{T},M,\Gamma,\Gamma_1),\quad \mathsf{ti}_\mathsf{M}(\mathsf{LT},LM,\Gamma,\Gamma_2)}{\mathsf{ti}_\mathsf{M}(\mathsf{T}\cdot\mathsf{LT},M\cdot LM,\Gamma,\Gamma_1\cup\Gamma_2)}$$

$$(\text{TI}_M\text{-Star}_1)\quad \frac{}{\mathsf{ti}_\mathsf{M}(*\mathsf{T},[],\Gamma,\emptyset)} \qquad (\text{TI}_M\text{-Star}_2)\quad \frac{\mathsf{ti}_\mathsf{M}(\mathsf{T},M,\Gamma,\Gamma_1),\quad \mathsf{ti}_\mathsf{M}(*\mathsf{T},LM,\Gamma,\Gamma_2)}{\mathsf{ti}_\mathsf{M}(*\mathsf{T},M\cdot LM,\Gamma,\Gamma_1\cup\Gamma_2)}$$

$$(\text{TI}_M\text{-Union})\quad \frac{\mathsf{ti}_\mathsf{M}(\mathsf{T}_0,M,\Gamma,\Gamma_0)\quad\text{or}\quad \mathsf{ti}_\mathsf{M}(\mathsf{T}_1,M,\Gamma,\Gamma_1)}{\mathsf{ti}_\mathsf{M}(\mathsf{T}_0+\mathsf{T}_1,M,\Gamma,\Gamma')},\quad \text{where:}$$

$$\Gamma'=\begin{cases}\Gamma_0+\Gamma_1 & \text{if } \mathsf{tpm}(\mathsf{T}_0,Q,\Gamma_0) \text{ and } \mathsf{tpm}(\mathsf{T}_1,Q,\Gamma_1)\\ \Gamma_i & \text{if } \mathsf{tpm}(\mathsf{T}_i,Q,\Gamma_i) \text{ and for no } \Gamma'' \ \mathsf{tpm}(\mathsf{T}_{i+1\bmod 2},Q,\Gamma''),\ i=0,1\end{cases}$$

$$(\text{TI}_M\text{-Abs})\quad \frac{\mathsf{tpm}(\mathsf{T},Q,\Gamma_1),\quad (\Gamma_1)_{|\tilde{y}}>(\Gamma)_{|\tilde{y}},\quad \mathsf{ti}_\mathsf{P}(P,\Gamma\cup(\Gamma_1)_{|\tilde{x}},\Gamma_2)}{\mathsf{ti}_\mathsf{M}((\mathsf{T})\mathsf{Abs},\ (Q_{\tilde{x}})P,\ \Gamma,\ (\Gamma_1)_{|\tilde{x}}\cup\Gamma_2)}$$

where $\tilde{y}=\mathsf{fv}(Q)\setminus\tilde{x}$ and $(\Gamma_1)_{|\tilde{y}}$ abstraction-free

Table 9: Inference for messages.

$$(\text{TI}_P\text{-In})\quad \frac{a\in ch(\mathsf{T}),\quad \mathsf{ti}_\mathsf{M}((\mathsf{T})\mathsf{Abs},A,\Gamma,\Gamma_1)}{\mathsf{ti}_\mathsf{P}(a.A,\Gamma,\Gamma_1)}$$

$$(\text{TI}_P\text{-Out})\quad \frac{\Gamma\vdash u\in ch(\mathsf{T}),\quad \mathsf{ti}_\mathsf{M}(\mathsf{T},M,\Gamma,\Gamma')}{\mathsf{ti}_\mathsf{P}(\overline{u}\langle M\rangle,\Gamma,\Gamma')} \qquad (\text{TI}_P\text{-Sum})\quad \frac{\forall i\in I:\ \mathsf{ti}_\mathsf{P}(u_i.A_i,\Gamma,\Gamma_i),\quad |I|\neq 1}{\mathsf{ti}_\mathsf{P}(\sum_{i\in I}u_i.A_i,\Gamma,\bigcup_{i\in I}\Gamma_i)}$$

$$(\text{TI}_P\text{-Rep})\quad \frac{\mathsf{ti}_\mathsf{P}(P,\Gamma,\Gamma')}{\mathsf{ti}_\mathsf{P}(!P,\Gamma,\Gamma')} \qquad (\text{TI}_P\text{-Par})\quad \frac{\mathsf{ti}_\mathsf{P}(P_1,\Gamma,\Gamma_1),\quad \mathsf{ti}_\mathsf{P}(P_2,\Gamma,\Gamma_2)}{\mathsf{ti}_\mathsf{P}(P_1|P_2,\Gamma,\Gamma_1\cup\Gamma_2)}$$

$$(\text{TI}_P\text{-Res})\quad \frac{\mathsf{ti}_\mathsf{P}(P,\Gamma,\Gamma')}{\mathsf{ti}_\mathsf{P}((\nu a)P,\Gamma,\Gamma')} \qquad (\text{TI}_P\text{-Else})\quad \frac{\mathsf{ti}_\mathsf{P}(P,\Gamma,\Gamma_1),\quad \mathsf{ti}_\mathsf{P}(R,\Gamma,\Gamma_2)}{\mathsf{ti}_\mathsf{P}(P\ \mathsf{else}\ R,\Gamma,\Gamma_1\cup\Gamma_2)}$$

Table 10: Inference system.

5 Inferring process annotations

Once channel capacities have been fixed, a suitable type for each pattern variable occurring in an abstraction can be extracted from those capacities. We present here a simple inference system intended to relieve programmers from explicit type annotation. Note, however, that there are cases where a programmer might prefer to use explicit type annotations (see Example 11 below).

Below, we presuppose a *non-annotated* syntax of processes and messages. The inference system is defined by two sets of mutually dependent rules, for messages and processes, presented in Table 9 and 10, respectively. The message inference system is defined as a predicate $\mathsf{ti}_\mathsf{M}(\mathsf{T},M,\Gamma,\Gamma')$: this yields a context Γ' for the bound variables[2] in M, such that M annotated with Γ' is of type T under Γ. The process inference system is defined as a predicate $\mathsf{ti}_\mathsf{P}(P,\Gamma,\Gamma')$: this yields a context Γ' for the bound variables in P, such that P annotated with Γ' is well-typed under Γ. The rules follow closely those of type checking, are syntax-driven and should be self-explanatory.

Let us discuss the relationship between inference and type checking. We use the following

[2]Note that we do *not* identify processes or messages up to α-equivalence, which would make $\mathsf{bv}(\cdot)$ not well-defined.

additional notation. Given a non annotated P, such that $\mathsf{bv}(P) \cap \mathsf{fv}(P) = \emptyset$, and a context Γ s.t. $\mathsf{bv}(P) \subseteq \Gamma$, we let P_Γ be the annotated process resulting by annotating each binding occurrence of any $x \in \mathsf{bv}(P)$ with the type $\Gamma(x)$; similarly for M_Γ. Note in particular that if $M = (Q_{\tilde{x}})P$ then $M_\Gamma = (Q_{\tilde{x}} : \Gamma_{|\tilde{x}})P_{\Gamma_{|\mathsf{bv}(P)}}$. The proofs of the following results are reported in Appendix C.

Theorem 4 (correctness) *Suppose* $\mathsf{fv}(P) \subseteq \mathsf{dom}(\Gamma_0)$ *and* $\mathsf{fv}(M) \subseteq \mathsf{dom}(\Gamma_0)$. *If* $\mathsf{ti}_\mathrm{P}(P, \Gamma_0, \Gamma)$ *then* $\Gamma_0 \vdash P_\Gamma : ok$ *and if* $\mathsf{ti}_\mathrm{M}(\mathsf{T}, M, \Gamma_0, \Gamma)$ *then* $\Gamma_0 \vdash M_\Gamma : T$.

Theorem 5 (completeness) *Suppose* $\mathsf{bv}(P) \subseteq \mathsf{dom}(\Gamma')$ *and* $\mathsf{bv}(M) \subseteq \mathsf{dom}(\Gamma')$. *Then: (a) If* $\Gamma_0 \vdash P_{\Gamma'} : ok$ *then there is* Γ *s.t.* $\mathsf{ti}_\mathrm{P}(P, \Gamma_0, \Gamma)$ *and* $\Gamma'_{|\mathsf{bv}(P)} > \Gamma$, *and (b) If* $\Gamma_0 \vdash M_{\Gamma'} : \mathsf{T}$ *then there is* Γ *s.t.* $\mathsf{ti}_\mathrm{M}(\mathsf{T}, M, \Gamma_0, \Gamma)$ *and* $\Gamma'_{|\mathsf{bv}(M)} > \Gamma$.

Inference for derived constructs. The following rules for Application and Case can easily be proven admissible, i.e. if the premises are provable so is the conclusion, assuming, for application, that a bound name $c \in ch(\mathsf{T})$ is chosen in the translation to the base syntax:

$$(\text{TI}p\text{-APPL}) \quad \frac{\mathsf{ti}_\mathrm{M}(\mathsf{T}, M, \Gamma, \Gamma_1), \quad \mathsf{ti}_\mathrm{M}((\mathsf{T})\mathsf{Abs}, A, \Gamma, \Gamma_2)}{\mathsf{ti}_\mathrm{P}(A \bullet M, \Gamma, \Gamma_1 \cup \Gamma_2)}.$$

$$(\text{TI}p\text{-CASE}) \quad \frac{\forall i \in \{1, \ldots, k\} : \quad \mathsf{ti}_\mathrm{P}(((Q_i)_{\tilde{x}_i})P_i \bullet M, \Gamma, \Gamma_i)}{\mathsf{ti}_\mathrm{P}(\mathrm{Case}\, M \,\mathrm{of}\, (Q_1)_{\tilde{x}_1} \Rightarrow P_1, \ldots, (Q_k)_{\tilde{x}_k} \Rightarrow P_k,\ \Gamma,\ \bigcup_{i=1,\ldots,k} \Gamma_i)}.$$

Example 11 Consider the process WS defined in Example 6 and the sorting assumptions defined in Example 10. If we apply the inference algorithm to WS, we obtain the following context:

$$\Gamma = \{x : ch(\mathsf{stream}), y : ch(\mathsf{stream}), z : ch((\mathsf{T})\mathsf{Abs}), y' : \mathsf{string}, z' : ch(\mathsf{stream})\}.$$

i.e., it holds $\mathsf{ti}_\mathrm{P}(WS, \emptyset, \Gamma)$. In particular we have that $\Gamma(x) = ch(\mathsf{stream})$, which is a subtype of the type assigned to x in Example 10 ($ch(\mathsf{low-stream}) > ch(\mathsf{stream})$). It may be argued that allocation of a channel variable at a subtype is more expensive than allocation of a channel variable at a supertype. This shows that explicit type annotation can sometimes be preferable to inference.

6 Dynamic abstractions

Although satisfactory in many situations, a static typing scenario does not seem appropriate in those cases where little is known in advance on actual types of data that will be received from the network.

Example 12 (a directory of services) Suppose one has to program an online directory of (references to) services. Upon request of a service of type T, for *any* T, the directory should lookup its catalog and respond by sending a channel of type $ch(\mathsf{T})$ along a reply channel. If the reply channel is fixed statically, it must be given capacity $ch(\mathbb{1})$, that is, any channel. Then, a client that receives a name at this channel must have some mechanism to cast at runtime this generic type to the subtype $ch(\mathsf{T})$, which means going beyond static typing. If the reply channel is provided by clients the situation does not get any better. E.g. consider the following service (here we use some syntactic sugar for the sake of readability):

$$!\mathit{request}.(\mathtt{req}[?t : \mathsf{Td}, ?x_{rep} : ch(\mathsf{Tr})])\,\mathsf{let}\, y = \mathit{lookup}(t)\,\mathsf{in}\,\overline{x}_{rep}\langle y\rangle \tag{1}$$

where lookup is a function from some type Td of type-descriptors to the type of *all* channels, $ch(\mathbb{1})$. It is not clear what capacity Tr the return channel variable x_{rep} should be assigned. The only choice that makes the above process well typed is to set $\mathsf{Tr} = ch(\mathbb{1})$, that is, x_{rep} can transport any channel. But then, a client's call to this service like $\overline{\mathit{request}}\langle\mathtt{req}[v_{td}, r]\rangle$, where r

has capacity $ch(\mathsf{T})$, is not well typed (because $r \in ch(ch(\mathsf{T}))$ and $ch(ch(\mathsf{T}))$ is not a subtype of $ch(\mathsf{Tr}) = ch(ch(\mathbf{1}))$).

Even ignoring the static vs. dynamic issue, the schemas sketched above would imply some form of encoding of type and subtyping into XML, which is undesirable if one wishes to reason at an abstract level. As we shall see below, dynamic abstractions can solve these difficulties.

The scenario illustrated in the above example motivates the extension of the calculus presented in the preceding sections with a form of dynamic abstraction. The main difference from ordinary abstractions is that type checking for pattern variables is moved to run-time. This is reflected into an additional communication rule, that explicitly invokes type checking. We describe below the necessary extensions to syntax and semantics. We extend the syntactic category of Abstractions thus:

$$A ::= \quad \cdots \mid (\!|Q_{\tilde{x}} : \Gamma|\!)P \quad \textit{Dynamic abstraction}$$

with $\tilde{x} = \mathsf{dom}(\Gamma)$. We let D range over dynamic abstractions and A over all abstractions. We add a new reduction rule:

$$(\text{COM-D}) \quad \frac{j \in I,\ a_j = a, \quad A_j = (\!|Q_{\tilde{x}} : \Gamma|\!)P, \quad \mathsf{match}(M, Q, \sigma), \quad \forall y \in \mathsf{dom}(\sigma) :\ \sigma(y) : \Gamma(y)}{\overline{a}\langle M\rangle \mid \sum_{i \in I} a_i.A_i \to P\sigma}.$$

We finally add a new type checking rule. For this, we need the following additional notation. Given Γ_1 and Γ_2, we write $\Gamma_1 \lesseqgtr \Gamma_2$ if $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$ and $\forall x \in \mathsf{dom}(\Gamma_1)$ there is a consistent type T s.t. $\mathsf{T} < \Gamma_1(x)$ and $\mathsf{T} < \Gamma_2(x)$.

$$(\text{TM-ABS-D}) \ \frac{\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1), \quad (\Gamma_1)_{|\tilde{x}} \lesseqgtr \Gamma_Q, \quad (\Gamma_1)_{|\tilde{y}} > \Gamma_{|\tilde{y}}, \quad \Gamma, \Gamma_Q \vdash P : ok}{\Gamma \vdash (\!|Q_{\tilde{x}} : \Gamma_Q|\!)P : (\mathsf{T})\mathsf{Abs}}$$

where $\tilde{y} = \mathsf{fv}(Q) \setminus \tilde{x}$ and $(\Gamma_1)_{|\tilde{y}}$ is abstraction free. The existence of a common consistent subtype for Γ_Q and $(\Gamma_1)_{|\tilde{x}}$ ensures a form of dynamic consistency for Q, detailed below.

We discuss now the extension of run-time safety. The safety property needs to be extended to inputs formed with dynamic abstractions. A stronger form of pattern consistency is needed.

Definition 14 (dynamic T-consistency) *An annotated pattern $Q : \Gamma$ ($\mathsf{fv}(Q) = \mathsf{dom}(\Gamma)$) is* dynamically T-consistent *if there is a message $M : \mathsf{T}$ s.t. $\mathsf{match}(Q, M, \sigma)$ and $\forall x \in \mathsf{dom}(\sigma)$ we have $\sigma(x) : \Gamma(x)$.*

Definition 15 (dynamic safety) *Let P be an annotated closed process. P is* dynamically safe *if for each name $a \in ch(\mathsf{T})$ conditions 1 and 2 of Definition 13 hold, and moreover the following condition is true: Suppose T is consistent. Whenever $P \equiv (\nu\, \tilde{h})(S \mid R)$, where S is a guarded summation, $a.D$ is a summand of S and $Q : \Gamma$ is D's annotated pattern, then $Q : \Gamma$ is dynamically T-consistent.*

It is straightforward to prove the extensions of Theorem 2 and Corollary 2 to the dynamic case, i.e.: every closed annotated well-typed P is dynamically safe, and dynamic safety is preserved by reductions (See Appendix D).

Corollary 3 (run-time dynamic safety) *Let P be an annotated closed process. If $P : ok$ and $P \to^* P'$ then P' is dynamically safe.*

Example 13 (a directory of services, continued) Consider again the directory of services. Clients can either request a (reference to a) service of a given type, by sending a message to channel *discovery*, or request the directory to update its catalog with a new service, using the channel *publish*. Each request to *discovery* should contain some type information, which would allow the directory to select a (reference to a) service of that type, taking subtyping into account. Types cannot be passed around explicitly. However one can pass a dynamic abstraction that will

do the selection on behalf of the client and return the result back to the client at a private channel. The catalog is maintained on a channel cat local to the directory. Thus the directory process can be defined as follows, where $\prod_{i\in I}!\overline{cat}\langle c_i\rangle$ stands for $!\overline{cat}\langle c_1\rangle \,|\, \cdots \,|\, !\overline{cat}\langle c_n\rangle$ (for $I = 1,\ldots,n$) and the following capacities are assumed: $discovery \in ch((ch(\mathbf{1}))\mathsf{Abs})$, $publish, cat \in ch(ch(\mathbf{1}))$.

$$\begin{array}{rcll} Directory & \triangleq & (\nu\, cat)(\prod_{i\in I}!\overline{cat}\langle c_i\rangle \,|\; & !\, publish.(?y : ch(\mathbf{1}))!\overline{cat}\langle y\rangle \\ & & \;|\; & !\, discovery.(?x : (ch(\mathbf{1}))\mathsf{Abs})\, cat.x\,) \end{array}$$

Note that $(ch(\mathbf{1}))\mathsf{Abs}$ is the type of all abstractions that can consume some channel. A client that wants to publish a new service S that accepts messages of some type T at a new channel $a \in ch(\mathsf{T})$ is:

$$C_1 \triangleq (\nu\, a)(\,\overline{publish}\langle a\rangle \,|\, S\,).$$

A client that wants to retrieve a reference to a service of type T, or any subtype of it, is:

$$C_2 \triangleq (\nu\, r)(\,\overline{discovery}\langle (\!|?z : ch(\mathsf{T})|\!)\overline{r}\langle z\rangle\rangle \,|\, r.(?y : ch(\mathsf{T}))C'\,)\ .$$

Note that we have preferred not to define C_2 as $(\nu\, r)(\,\overline{discovery}\langle (\!|?y : ch(\mathsf{T})|\!)C'\rangle)$ so to avoid to charge the server with non-local computations. In fact C' may have to use resources which are local to C_2 and access to these resources from the server location could be expensive.

Suppose $r \in ch(ch(\mathsf{T}))$. Assuming S and C' are well typed (the latter under $\{y : ch(\mathsf{T})\}$), it is easily checked that the global system

$$P \triangleq Directory \,|\, C_1 \,|\, C_2$$

is well typed too.

7 Publishing and discovering services

In this section, we further elaborate on the theme of publishing and discovering that we used as a motivating example for dynamic abstractions in the preceding section. We first define an extension of XPi, that we name XPi^{E}, with primitives for publishing and discovering. Then we show that XPi^{E} can be encoded into XPi.

In XPi^{E} an UDDI directory of services available at channel d is written $d\langle S\rangle$, where $S \subseteq_{fin} \mathcal{N}$ is the finite set of published services. The primitive $\overline{d}^{(p)}\langle c\rangle$ allows to publish the service c on d, while $\overline{d}^{(q)}\langle \mathsf{T}, a\rangle$ allows to query d for services of type $\mathsf{T} \in \mathcal{T}$. Channel a is used by the directory as reply channel. In what follows, we presuppose a distinct sort $\mathcal{D} \subseteq \mathcal{N}$, ranged over by $d, d', \ldots$, of directory identifiers. The set of XPi^{E} processes is defined by extending the syntax of XPi with the clauses below (recall that $u \in \mathcal{N} \cup \mathcal{V}$):

$$P ::= \cdots \,\big|\, d\langle S\rangle \,\big|\, \overline{u}^{(p)}\langle c\rangle \,\big|\, \overline{u}^{(q)}\langle \mathsf{T}, u\rangle.$$

where we require T to be of the form $ch(\mathsf{T}')$ for some T'. It is worth noticing that a name $d \in \mathcal{D}$ received in input cannot be used to define a new direcory. The reduction semantics of the new operators is given by the following rules:

$$(\text{PUB})\ d\langle S\rangle \,|\, \overline{d}^{(p)}\langle c\rangle \to d\langle S \cup \{c\}\rangle$$

$$(\text{QUERY-T})\ \frac{\exists c \in S \quad c : \mathsf{T}}{d\langle S\rangle \,|\, \overline{d}^{(q)}\langle \mathsf{T}, a\rangle \to d\langle S\rangle \,|\, \prod\limits_{\{c\in S \,|\, c:\mathsf{T}\}} \overline{a}\langle c\rangle} \qquad (\text{QUERY-F})\ \frac{\nexists c \in S : \quad c : \mathsf{T}}{d\langle S\rangle \,|\, \overline{d}^{(q)}\langle \mathsf{T}, a\rangle \to d\langle S\rangle \,|\, \overline{a}\langle \mathtt{ff}\rangle}$$

where `ff` stands for the boolean value "false". When a client queries a directory for services complying with a certain type, the directory either replies `ff`, if no service of that type is available

(rule (QUERY-F)), or displays all possibilities to the client (rule (QUERY-T)). The client may then decide to use one or more of the offered services (e.g., choosing the one with the most precise type, or trying them all for estimating their performances, etc.).

For the sake of simplicity, in the following we shall freely use recursive definitions in both XPi^{E} and XPi, which can be easily coded up by using replication [26].

Example 14 A client C queries a directory d for a service that is ready to receive a streamed file (to, e.g., in turn stream it to a set of subscribed users). Assume stream is the type of streamed files and $\mathsf{low-stream}$ is the type low-rate streams, and $\mathsf{low-stream} < \mathsf{stream}$. Hence $ch(\mathsf{stream}) < ch(\mathsf{low-stream})$. As already discussed in Example 10, it can be the case that using a channel of the subtype $ch(\mathsf{stream})$ requires more resources than using a channel of the supertype $ch(\mathsf{low-stream})$. A client C with scarce resources, waiting to stream a low-quality file f, might even decide to stick to $ch(\mathsf{low-stream})$ services. The client C defined below, after querying the directory, discards all services of type $ch(\mathsf{stream})$ (first branch of C''s else). If no service of type $ch(\mathsf{low-stream})$ is available, eventually C blocks. The whole system is $Sys \triangleq C \,|\, D$, with:

$$
\begin{aligned}
C &\triangleq (\nu r)\big(\overline{d}^{(q)}\langle ch(\mathsf{low-stream}), r\rangle \,\big|\, r.(\!|?x:\mathsf{bool}|\!)\overline{abort} \,\big|\, C'\big)\\
C' &\triangleq r.(\!|?x:ch(\mathsf{low-stream})|\!)(\nu t)\Big(\,(\bar{t}\langle x\rangle \,|\, t.(\!|?y:ch(\mathsf{stream})|\!)C')\\
&\qquad\qquad\qquad\qquad \mathsf{else}\ (\bar{t}\langle x\rangle \,|\, t.(\!|?y:ch(\mathsf{low-stream})|\!)\bar{y}\langle f\rangle)\Big)\\
D &\triangleq d\langle S\rangle
\end{aligned}
$$

where S is the set of offered services.

It would be easy to modify this example to describe a more realistic scenario, where clients do not simply block if $ch(\mathsf{low-stream})$ services are not available. Indeed, the directory could be modified to also provide the clients with the number n of different services being offered. Then a client could search among these n services to e.g. find the one with the greatest or cheapest type.

It is easy to extend XPi's type system to cope with the new primitives. First, we assume the capacity function maps $\mathcal{D}$ to type $\mathbf{L}$, that is, names of sort $\mathcal{D}$ cannot transport anything, hence cannot be used as channels. Recall that we write $\Gamma \vdash u \in \mathcal{D}$ if either $u = d \in \mathcal{D}$ or $u = x$ and $\Gamma(x) = \mathcal{D}$. The system in Table 8 is extended by adding the following rules:

$$
\text{(T-DIR)}\ \Gamma \vdash d\langle S\rangle : ok \qquad \text{(T-PUB)}\ \frac{\Gamma \vdash u \in \mathcal{D}}{\Gamma \vdash \overline{u}^{(p)}\langle c\rangle : ok}
$$

$$
\text{(T-QUERY)}\quad \frac{\Gamma \vdash u \in \mathcal{D} \quad \Gamma \vdash u' \in ch(\mathsf{T}+\mathsf{bool})}{\Gamma \vdash \overline{u}^{(q)}\langle \mathsf{T}, u'\rangle : ok}
$$

Run-time safety with these new typing rules carries over. In what follows, we use barbed equivalence (Definition 5) for reasoning on XPi^{E} processes. We observe not only outputs, but also publish and query actions, hence we extend the definition of barbs as expected, in particular it holds that $\overline{d}^{(p)}\langle c\rangle \downarrow_d$ and $\overline{d}^{(q)}\langle \mathsf{T}, a\rangle \downarrow_d$.

We define now a translation function, $[\![\cdot]\!]^E$, from XPi^{E} to XPi. In what follows, we use the following abbreviation: $\mathsf{chan} \triangleq ch(\mathbf{L})$. The relevant clauses of the definition are the following:

$$
\begin{aligned}
[\![d\langle S\rangle]\!]^E &= D(S) \text{ where}\\
&\qquad D(S) \triangleq\ d.(\mathtt{publish}(?x:\mathsf{chan}))D(S\cup\{x\})\\
&\qquad\qquad + d.(\mathtt{query}(?y:(\mathsf{chan}+\mathsf{bool})\mathsf{Abs}))\\
&\qquad\qquad\qquad \big(\,(\nu t)\big((\textstyle\prod_{c\in S}\bar{t}\langle c\rangle \,|\, !t.y)\ \mathsf{else}\ (\bar{t}\langle \mathtt{ff}\rangle \,|\, t.y)\big)\,|\,D(S)\,\big)\\
[\![\overline{u}^{(p)}\langle c\rangle]\!]^E &= \overline{u}\langle \mathtt{publish}(c)\rangle\\
[\![\overline{u}^{(q)}\langle \mathsf{T}, a\rangle]\!]^E &= \overline{u}\langle \mathtt{query}((\!|?z:\mathsf{T}+\mathsf{bool}|\!)\bar{a}\langle z\rangle)\rangle
\end{aligned}
$$

in the other cases the function just goes through the structure of terms.

Concerning types of the translated processes (in XPi), we make the following assignments

$$d,u : ch\big(\texttt{publish}(\mathsf{chan}) + \texttt{query}((\mathsf{chan}+\mathsf{bool})\mathsf{Abs})\big) \;\text{ and }\; t : ch(\mathsf{chan}+\mathsf{bool}) \tag{2}$$

while type associations for the other names and variables remain unchanged.

Example 15 In this example we show the encoding of the system *Sys* defined in Example 14. $[\![Sys]\!]^E = D(S) \,|\, [\![C]\!]^E$ where $D(S)$ is given by the encoding above and

$$[\![C]\!]^E = \overline{d}\langle \texttt{query}((\!|?x : ch(\mathsf{low}-\mathsf{stream})+\mathsf{bool}|\!)\overline{r}\langle x\rangle)\rangle \,\big|\, r.(\!|?x : \mathsf{bool}|\!)\overline{abort} \,\big|\, C' \;.$$

In the following we denote by $\approx_{[\![\cdot]\!]^E}$ the barbed equivalence obtained by closing barbed bisimulation (Definition 4) under contexts that are translations of $\mathrm{XPi}^{\mathrm{E}}$ contexts (cfr. Definition 7). The corollary below guarantees the correctness of the encoding. Note that we close barbed equivalence by considering only contexts that are encodings of $\mathrm{XPi}^{\mathrm{E}}$ contexts: the intuitive content of this fact is that programs written in $\mathrm{XPi}^{\mathrm{E}}$ can be faithfully translated into XPi. The proofs of the following results are reported in Appendix E.

Lemma 6 *Suppose $P \in \mathcal{P}^E$.*

1. *$P \to P'$ implies that $\exists R$ such that $[\![P]\!]^E \to R$ and $[\![P']\!]^E \lesssim R$;*
2. *$[\![P]\!]^E \to R$ implies that $\exists P' \in \mathrm{XPi}^E$ such that $P \to P'$ and $[\![P']\!]^E \lesssim R$;*
3. *$P \downarrow_a$ if and only if $[\![P]\!]^E \downarrow_a$.*

Theorem 6 *Suppose $P \in \mathcal{P}^E$. $P \dot{\approx} [\![P]\!]^E$.*

Corollary 4 *Suppose $P, P_1, P_2 \in \mathcal{P}^E$.*

1. *P well-typed implies $[\![P]\!]^E$ well-typed under the type assumptions in (2);*
2. *$P_1 \approx P_2$ if and only if $[\![P_1]\!]^E \approx_{[\![\cdot]\!]^E} [\![P_2]\!]^E$.*

8 *I*-Barbed equivalence

In [31], Milner and Sangiorgi propose *barbed bisimulation* as a tool for uniformly defining bisimulation-based equivalences. Barbed equivalence is useful for its "portability" when studying a new calculus or a refinement of an existing one, as we are doing here.

Barbed bisimulation is a very coarse relation. According to a common pattern, one closes barbed bisimulation under all contexts, thus getting barbed congruence. Here, we find it useful to depart from this pattern so as to capture an *input locality* property for the observed processes (in the same vein as [28]). Approximately, one may think of each observed process P as equipped with an "interface" I, a set of input channels at which services are offered. Input channels in I should remain confined to P, in other words, only external observers that do not own the input capability on channels in I should be considered when closing barbed equivalence by contexts. Moreover, one wants to consider only well-typed processes and observers. These considerations motivate a form of barbed equivalence presented in the sequel.

The definition relies on the reduction relation of the calculus and on an barbs, $P \downarrow_a$, which have been already defined in Section 2.4. We define here a version of barbed bisimulation that respects an input interface I. This means that output at names in I are not observed, because the observer has not the corresponding input capability.

Definition 16 (I-barbed bisimulation) *Let $I \subseteq \mathcal{N}$. A symmetric binary relation on annotated closed processes is a* I-barbed bisimulation *if $(P,R) \in \mathcal{R}$ implies:*

- *whenever* $P \to P'$ *then there is* R' *such that* $R \to^* R'$ *and* $(P', R') \in \mathcal{R}$*;*
- *whenever* $P \downarrow_{\overline{a}}$ *and* $a \notin I$ *then* $R \Downarrow_{\overline{a}}$*.*

Two processes P *and* R *are* I-barbed bisimilar*, written* $P \dot{\approx}^I R$*, if* $(P,R) \in \mathcal{R}$ *for some* I*-barbed bisimulation* $\mathcal{R}$*.*

Note that one gets ordinary barbed bisimulation (Definition 4) by setting $I = \emptyset$. The next step is closing I-barbed bisimulation under appropriate contexts, while respecting input locality for names in I. In the sequel, let us denote by $\mathsf{isubj}(P)$ the set of names that occur free in P in input subject position; similarly for $\mathsf{isubj}(M)$.

Definition 17 (I-barbed equivalence) *Let* $I \subseteq \mathcal{N}$*. Two well-typed processes* P_1 *and* P_2 *are* I-barbed equivalent*, written* $P_1 \approx^I P_2$*, if for each* $\widetilde{h}$ *and each well-typed* R *s.t.* $\mathsf{isubj}(R) \cap I = \emptyset$*, it holds that* $(\nu\widetilde{h})(P_1|R) \dot{\approx}^I (\nu\widetilde{h})(P_2|R)$*.*

Ordinary barbed equivalence (Definition 5) is obtained by setting $I = \emptyset$. Note that I-barbed equivalence is not a congruence (not even ordinary barbed equivalence is), but it is preserved by restriction, and by parallel composition with those well-typed R s.t. $\mathsf{isubj}(R) \cap I = \emptyset$.

Example 16 This example illustrates the effect of considering only well-typed contexts. Suppose $I = \emptyset$ and $a \in ch(f[\mathsf{int}])$ and consider

$$P = a.(?x : f(*\mathsf{int})) \text{ Case } x \text{ of } \quad \begin{array}{ll} f[] & \Rightarrow P_1 \\ \_ & \Rightarrow P_2\,. \end{array}$$

Note that, according to (TM-ABS), well-typedness of the Case continuation in the process P above is evaluated under the assumption $x : f(*\mathsf{int})$.

Clearly, $P \approx^I a.(?x : f(*\mathsf{int}))P_2$, because no well-typed context ever sends $f[]$ along a, hence the first branch of the Case is never triggered. Note that this equality does not hold for untyped barbed equivalence.

Example 17 (a web service, continued) Consider the web service WS and the clients C_1 and C_2 defined in Example 6, and let $I = \{stream, download\}$. The following equality states that, not surprisingly, requesting WS a streaming service is functionally equivalent to requesting download and then running the player locally, regardless of the capacity of the employed channels (high or low):

$$WS|C_1 \approx^I WS|C_2\,.$$

The above equality does not hold for ordinary ($I = \emptyset$) barbed equivalence, because, e.g. C_1 has an output barb on $stream$, which C_2 does not.

Note that, albeit defined over all closed processes, $\approx^I$ only makes sense for those processes that do not export input capability of names in I. This may happen by "packaging" input channels in abstractions that are passed around, as in $P = \overline{a}\langle([])b.([])\mathbf{0}\rangle|P'$ and $I = \{b\}$: $P \mid a.(?x)(x \bullet []) \to^* P' \,|b.([])\mathbf{0}$.

9 Conclusions and related work

We have presented XPi, a core calculus for XML messaging, featuring asynchronous communications, pattern matching, name and code mobility, static and dynamic typing. We have proved results on run-time safety, and presented a notion of barbed equivalence that is useful to validate interesting equations. Flexibility of the language has been demonstrated by a number of examples, mainly concerning description and discovery of services.

As for further work, it would be interesting to devise tractable characterizations of I-barbed equivalence, in terms of a labelled bisimulation. One major obstacle towards this result is the presence of pattern matching, which makes existing techniques (se e.g. [29]) not directly applicable.

A number of proposals aim at integrating XML processing primitives in the context of traditional, statically typed sequential languages or logics. The ones most closely related to our work are XDuce [23] and $\mathbb{C}$Duce, [5], two (functional) languages for XML document processing. XPi's list-like representation of documents draws its inspiration from them. TQL [12] is both a logic and a query language for XML, based on a spatial logic for the Ambient calculus [13]. All these languages support query primitives more sophisticated than XPi's patterns, but issues raised by communication and code/name mobility, which are our main focus, are of course absent.

The presence of abstractions makes XPi somehow related to higher-order π-calculus [29], an extension of the π-calculus where processes can be passed around. In fact, XPi might also be viewed as a typed version of higher-order π with structured messages and pattern matching.

Early works aiming at integration of XML and process calculi are [21] and [6]. Xdπ [21] is a calculus for describing interaction between data and processes across distributed locations; it is focused on process migration rather than on communication and pattern matching. A type system for Xdπ with security types ensuring secrecy of data (by means of access and movements rights) is provided in [18]. Iota [6] is a concurrent XML scripting language for home-area networking. It relies on syntactic subtyping, like XPi, but is characterized by a different approach to XML typing. In particular, Iota's type system just ensures well-formedness of XML documents, rather than the stronger validity, which we consider here.

Roughly contemporary to ours, and with similar goals, are [11] and [17]. The language πDuce [11] features asynchronous communication and name mobility. Similarly to XDuce's, πDuce's pattern matching embodies built-in type checks, which may be expensive at run-time. The language in [17] is basically a π-calculus enriched with a rich form of "semantic" subtyping and pattern matching. Code mobility is not addressed. Pattern matching, similarly to πDuce's, performs type checks on messages. By contrast, in XPi static type checks and plain pattern matching suffice, as types of pattern variables are checked statically against channel capacities. We confine dynamic type checking to dynamic abstractions, which can be used whenever no refined typing information on incoming messages is available (e.g. at channels of capacity $\mathbf{T}$). The type systems in [17] and [11] also guarantee a form of absence of deadlock, which however presupposes that basic values do not appear in patterns. In XPi, we thought it was important to allow basic values in patterns for expressiveness reasons (e.g., they are crucial in the encoding of the spi-calculus presented in Section 2).

Finally, we mention some recent proposals oriented to the definition of contract languages for Web Services focusing both on documents' schemas [14] and on contract behaviour [15, 16]. Essentially, the schema language in [14] allows to describe XML documents containing references to remote operations. The contract language in [16] is an evolution of those in [15]. The main focus of both works is on compliance of clients and servers to exposed contracts. Specifically, they put forward techniques to extrapolate the behaviours of services and clients in terms of CCS processes and then check their agreement to the given contract by means of a compliance relation.

References

[1] M. Abadi, and A.D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1-70, Academic Press, 1999.

[2] L. Acciai, and M. Boreale. XPi: a Typed Process Calculus for XML Messaging. In *Proceedings of FMOODS'05*, *LNCS* 3535:47–66. Springer, 2005.

[3] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulation for the Asynchronous π-calculus, 1997. In *Proceedings of CONCUR '96*, *LNCS* 1119:147–162, Springer Verlag. A revised version has appeared in *Theoretical Computer Science* 195(2):291–324, 1998.

[4] T. Andrews, F. Curbera, and S. Thatte. Business Process Execution Language for Web Services, v1.1, 2003. `http://ifr.sap.com/bpel4ws`.

[5] V. Benzaken, G. Castagna, and A. Frisch. Cduce: An XML-Centric General-Purpose Language. In *Proceedings of the ACM International Conference on Functional Programming*, pp. 51–63, 2003.

[6] G.M. Bierman and P. Sewell. Iota: A concurrent XML scripting language with applications to Home Area Networking. Technical Report 577, University of Cambridge Computer Laboratory, 2003.

[7] Biztalk Server Home. `http://www.microsoft.com/biztalk/`.

[8] S. Bjorg, and L.G. Meredith. Contracts and Types. *Communication of the ACM*, 46(10):41–47, 2003.

[9] M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *Theoretical Computer Science* 195(2):205–226, 1998.

[10] M. Boreale, and D. Sangiorgi. Bisimulation in Name-Passing Calculi without Matching. In *Proceedings of LICS '98*, pp. 165–175, 1998.

[11] A. Brown, C. Laneve, and L.G. Meredith. πDuce: A process calculus with native XML datatypes. EPEW/WS-FM 2005, *LNCS* 3670:18–34. Springer, 2005.

[12] L. Cardelli, and G. Ghelli. TQL: A Query Language for Semistructured Data Based on the Ambient Logic. *Mathematical Structures in Computer Science* 14(3):285–327, 2004.

[13] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science* 240(1):177-213, 2000.

[14] S. Carpineti and C. Laneve. A basic contract language for Web Services. In *Proceedings of ESOP '06*, *LNCS* 3924:197–213. Springer, 2006.

[15] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani A formal account of contracts for Web Services. In *Proceedings of WS-FM '06*, *LNCS* 4184:148–162. Springer, 2006.

[16] G. Castagna, N. Gesbert, and L. Padovani. A Theory of Contracts for Web Services. In *Proceedings of POPL'08*, pp. 261–272, 2008.

[17] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *Proc. of LICS'05*, pp. 92–101, 2005.

[18] M. Dezani-Ciancaglini, S. Ghilezan and J. Pantovic, Security types for Dynamic Web Data. In *Proceeding of TGC'06*, *LNCS* 4661:263–280, Springer, 2007.

[19] W. Emmerich, M. Aoyama and J. Sventek. The impact of research on middleware technology. *SIGSOFT Software Engineering Notes* 32(1):21–46, 2007.

[20] D.C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, 2001. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502`.

[21] P. Gardner, and S. Maffeis. Modelling Dynamic Web Data. In *Proceedings of DBPL 2003*, *LNCS* 2921:130–146. Springer, 2003. A revised version has appeared in *Theoretical Computer Science*, 342(1):104-131, 2005.

[22] H. Hosoya, and B. Pierce. Regular Expression Pattern Matching for XML. *Journal of Functional Programming* 13(6):961–1004, 2003.

[23] H. Hosoya, and B. Pierce. Xduce: A Statically Typed XML Processing Language. In *Proceedings of ACM Transaction on Internet Technology* 3(2):117–148, 2003.

[24] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming* 70(1):96–118, 2007.

[25] M. Merro. Locality and polyadicity in asynchronous name-passing calculi. In *Proceedings of FoSSaCS 2000*, *LNCS* 1784:238–251. Springer, 2000.

[26] R. Milner. The Polyadic π-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, LFCS, Dept. of Computer Science, Edinburgh University, 1991.

[27] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, part I and II. *Information and Computation* 100:1–41 and 42–78, 1992.

[28] D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science* 221(1-2): 457–493, 1999.

[29] D. Sangiorgi. Bisimulation in higher-order calculi. *Information and Computation* 131(2), 1996.

[30] B. Pierce, and D. Sangiorgi. Typing and Subtyping for Mobile Process. *Mathematical Structures in Computer Science* 6(5): 409-453, 1996.

[31] D. Sangiorgi, and R. Milner. Barbed bisimulation. In *Proceedings of ICALP'92*, *LNCS* 623:685–695. Springer, 1992.

[32] D. Sangiorgi, and R. Milner. Techniques of "weak bisimulation up to". In *Proceedings of CONCUR'92*, *LNCS* 630. Springer, 1992.

[33] S. Steinke. Middleware Meets the Network. *LAN: The Network Solutions Magazine 10*, 13, December 1995.

[34] W3C. Web Services Description Language 2.0. W3C Recommendation, 2007. `http://www.w3.org/TR/wsdl20/`.

[35] Web services activity web site, 2002. `http://www.w3.org/2002/ws`.

A Proofs of Section 2

Recall from [32] that a *weak barbed bisimulation up to expansion* is a relation satisfying the condition given in Definition 4, *but* with the clause "$P' \approx R'$" replaced by the weaker "$P' \gtrsim \dot{\approx} \lesssim R'$". From the results in [32], it follows that if $\mathcal{R}$ is a weak barbed bisimulation up to to expansion then $\mathcal{R} \subseteq \dot{\approx}$.

Proposition A.1 (Proposition 1) *Let P be a closed process in* $\mathrm{XPi}^{\mathrm{cr}}$.

1. *if $P \to P'$ then $(\!|P|\!) \to^* (\!|P'|\!)$;*
2. *if $(\!|P|\!) \to P'$ then $\exists P'' \in \mathrm{XPi}^{\mathrm{cr}}$ s.t. $P \to P''$ and $(\!|P''|\!) \lesssim P'$;*
3. *$P \downarrow a$ implies $(\!|P|\!) \Downarrow a$ and $(\!|P|\!) \downarrow a$ implies $P \Downarrow a$.*

Proof:

1. The proof is straightforward by induction on the derivation of $P \to P'$. We consider the last reduction rule applied; the most interesting case is rule (Dec), in the other cases $(\!|P|\!)$ reduces directly into $(\!|P'|\!)$.

   case $\{M\}_N$ of $\{x\}_N$ in $P \to P[M/x]$ and

$$\begin{aligned}
(\!|\text{case } \{M\}_N \text{ of } \{x\}_N \text{ in } P|\!) &= (\nu r)\big([\![\{M\}_N]\!] \bullet [N,r] \mid r.(?x)\,(\!|P|\!)\big) && \text{by def. of } (\!|\cdot|\!)\\
&= (\nu r)\big(([N,?x])\overline{x}\langle[\![M]\!]\rangle \bullet [N,r] \mid r.(?x)\,(\!|P|\!)\big) && \text{by def. of } [\![\cdot]\!]\\
&= (\nu r)\big((\nu c)(c.([N,?x])\overline{x}\langle[\![M]\!]\rangle \mid \overline{c}\langle[N,r]\rangle) \mid r.(?x)\,(\!|P|\!)\big) && \text{by def. of } \bullet\\
&\to (\nu r)\big(\overline{r}\langle[\![M]\!]\rangle \mid r.(?x)\,(\!|P|\!)\big) && \text{by (com)}\\
&\to (\!|P|\!)[{}^{[\![M]\!]}/_x] && \text{by (com)}\\
&= (\!|P[{}^M/_x]|\!) && \text{by def. of } (\!|\cdot|\!).
\end{aligned}$$

2. The proof is straightforward by induction on $(\!|P|\!) \to P'$. We proceed by distinguishing the possible cases for the structure of P. The most interesting case is when $P = \text{case } \{M\}_N \text{ of } \{x\}_{N'} \text{ in } R$. There is only one possibility for $(\!|P|\!) \to P'$, that is

$$\begin{aligned}
(\!|P|\!) &= (\nu r)\big((\nu c)(c.([N,?x])\overline{x}\langle[\![M]\!]\rangle \mid \overline{c}\langle[N',r]\rangle) \mid r.(?x)\,(\!|R|\!)\big)\\
&\to (\nu r)\big((\nu c)(\overline{r}\langle[\![M]\!]\rangle \mid \mathbf{0}) \mid r.(?x)\,(\!|R|\!)\big)\\
&= P'.
\end{aligned}$$

   Here is a communication on c has been inferred by rules (com) and then (ctx). By the premise of (com), we have $\mathsf{match}([N',r],[N,x],\sigma)$, hence $N' = N$, $\sigma(x) = r$. By $N = N'$ and rule (Dec), we have case $\{M\}_N$ of $\{x\}_{N'}$ in $R \to R[{}^M/_x] = P''$. It is a matter of routine to prove, by exhibiting a suitable expansion relation, that $(\!|P''|\!) \lesssim P'$.

3. The proof is straightforward by induction on the structure of P and by (1) and (2).

□

Theorem A.1 (Theorem 1) *Let P be a closed process in* $\mathrm{XPi}^{\mathrm{cr}}$. *$P \dot{\approx} (\!|P|\!)$.*

PROOF: We show that the relation

$$\mathcal{R} = \left\{ \langle P, \langle\!| P |\!\rangle \rangle \,\middle|\, P \in \mathrm{XPi}^{\mathrm{cr}} \right\}$$

is a weak barbed bisimulation up to $\lesssim$.

First of all, By Proposition A.1 (3), $P \downarrow a$ implies $\langle\!| P |\!\rangle \Downarrow a$ and $\langle\!| P |\!\rangle \downarrow a$ implies $P \Downarrow a$.

If $P \to P'$, by Proposition A.1 (1), $\langle\!| P |\!\rangle \to^* \langle\!| P' |\!\rangle$ and $\langle P', \langle\!| P' |\!\rangle \rangle \in \mathcal{R}$.

If $\langle\!| P |\!\rangle \to P'$, by Proposition A.1 (2), $P \to P''$ with $\langle\!| P'' |\!\rangle \lesssim P'$. $\langle P'', \langle\!| P'' |\!\rangle \rangle \in \mathcal{R}$; hence $P'' \mathcal{R} \lesssim P'$.
That is $\mathcal{R} \subseteq \dot{\approx}$. □

Corollary A.1 (Corollary 1) *Let P_1 and P_2 be closed processes in* $\mathrm{XPi}^{\mathrm{cr}}$. *$P_1 \approx P_2$ if and only if $\langle\!| P_1 |\!\rangle \approx_{\langle\!| \cdot |\!\rangle} \langle\!| P_2 |\!\rangle$.*

PROOF:

(⇒): Take any R and $\tilde{b}$, we have to prove that $(\nu\tilde{b})(\langle\!| P_1 |\!\rangle | \langle\!| R |\!\rangle) \dot{\approx} (\nu\tilde{b})(\langle\!| P_2 |\!\rangle | \langle\!| R |\!\rangle)$.

By definition of $\approx$, $P_1 \approx P_2$ implies that $(\nu\tilde{b})(P_1|R) \dot{\approx} (\nu\tilde{b})(P_2|R)$. By Theorem A.1, $(\nu\tilde{b})(P_1|R) \dot{\approx} \langle\!| (\nu\tilde{b})(P_1|R) |\!\rangle$ and $(\nu\tilde{b})(P_2|R) \dot{\approx} \langle\!| (\nu\tilde{b})(P_2|R) |\!\rangle$. By transitivity of $\dot{\approx}$, $\langle\!| (\nu\tilde{b})(P_1|R) |\!\rangle \dot{\approx} \langle\!| (\nu\tilde{b})(P_2|R) |\!\rangle$, that is $(\nu\tilde{b})(\langle\!| P_1 |\!\rangle | \langle\!| R |\!\rangle) \dot{\approx} (\nu\tilde{b})(\langle\!| P_2 |\!\rangle | \langle\!| R |\!\rangle$
Hence, by Definition 7 ($\approx_{\langle\!| \cdot |\!\rangle}$), $\langle\!| P_1 |\!\rangle \approx_{\langle\!| \cdot |\!\rangle} \langle\!| P_2 |\!\rangle$.

(⇐): the proof proceeds similarly.

□

B Proofs of Section 4

Lemma B.1 (Lemma 1) *Suppose* T *is consistent. If* $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$ *then* Q *is* T*-consistent.*

PROOF: The proof is straightforward by induction on the derivation of $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$. The following base cases are the most interesting:

(TPM-TOP): Consider the message M obtained by replacing every variable in Q with a value; by (TM-TOP) $M : \mathbf{T}$ and M matches Q, thus Q is $\mathbf{T}$-consistent.

(TPM-VAR): by $\mathsf{tpm}(\mathsf{T}, x, \{x : \mathsf{T}\})$ and the premises of the rule, we get $Q = x$ and Q is T-consistent because we can prove that for every consistent type T there is a message $M : \mathsf{T}$ (the proof is immediate assuming that for each basic type bt there is at least one $v : \mathsf{bt}$).

□

Theorem B.1 (Theorem 2) *Let P be an annotated closed process and suppose $P : ok$, then P is safe.*

PROOF: The proof is straightforward by induction on the derivation of $P : ok$, by distinguishing the last typing rule applied. Case (T-IN) relies on Lemma B.1. □

Lemma B.2 (Lemma 2) *If $T' < T$ then for any M such that $\Gamma \vdash M : T'$ we have $\Gamma \vdash M : T$.*

PROOF: We distinguish two cases:

$M = x$: by $\Gamma \vdash M = x : \mathsf{T}'$ and the premises of the rule(TM-VAR), $\Gamma(x) < \mathsf{T}'$; but $\mathsf{T}' < \mathsf{T}$, therefore $\Gamma(x) < \mathsf{T}$ and by rule (TM-VAR) we obtain $\Gamma \vdash M = x : \mathsf{T}$.

$M \neq x$: in this case the proof is straightforward by induction on the sum of the depths of the derivations of $\mathsf{T}' < \mathsf{T}$ and $\Gamma \vdash M : \mathsf{T}'$. We distinguish the last subtyping rule applied; the most interesting cases are the following:

(SUB-SORT): by $ch(\mathsf{T}') < ch(\mathsf{T})$ and the premises of the rule, we get $\mathsf{T} < \mathsf{T}'$. By $\Gamma \vdash M : ch(\mathsf{T}')$ and the premises of the rule (TM-VALUE), we get $M = a : \mathcal{S} = ch(\mathsf{T}')$ and by definition $\exists \mathcal{S}' < \mathcal{S} : \quad a \in \mathcal{S}'$. By transitivity $\mathcal{S}' < ch(\mathsf{T})$ and by definition $a : ch(\mathsf{T})$, that is, by rule (TM-VALUE), $\Gamma \vdash M : ch(\mathsf{T})$.

(SUB-TOP): $\mathsf{T} < \mathbf{T}$ and by rule (TM-TOP) $\Gamma \vdash M : \mathbf{T}$.

(SUB-BOTTOM): $\mathbf{\bot} < \mathsf{T}$. $\nexists M : \mathbf{\bot}$.

(SUB-BASIC): by $\mathsf{bt1} < \mathsf{bt2}$ and the premises of the rule, we get $\mathsf{bt1} \prec \mathsf{bt2}$. By the premises of the rule (TM-VALUE) and $\Gamma \vdash M : \mathsf{bt1}$, we get $M = v : \mathsf{bt1}$ and by $\mathsf{bt1} \prec \mathsf{bt2}$ we obtain $v : \mathsf{bt2}$. By rule (TM-VALUE) we have $\Gamma \vdash M = v : \mathsf{bt2}$.

□

Lemma B.3 *Let M be a closed message. $M : \mathsf{T}$ and $\mathsf{match}(M, Q, \sigma)$ imply $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$.*

PROOF: The proof is straightforward by induction on typing rules for messages, distinguishing the cases $Q = x$ and $Q \neq x$. □

Lemma B.4 (Lemma 3) *Let $M : \mathsf{T}$ be a closed message. If $\mathsf{match}(M, Q, \sigma)$ and $\mathsf{tpm}(T, Q, \Gamma)$ then $\forall x \in \mathsf{dom}(\sigma) : \sigma(x) : \Gamma(x)$.*

PROOF: The proof proceeds by induction on the derivation of $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$. We distinguish the last rule applied:

(TPM-EMPTY) (TPM-VALUE) (TPM-STAR$_1$): in these cases $\Gamma = \emptyset$ and $\sigma = \varepsilon$.

(TPM-TOP): by $\mathsf{tpm}(\mathbf{T}, Q, \Gamma)$ and the premises of the rule, we have $\forall x \in \mathsf{fv}(Q) : \Gamma(x) : \mathbf{T}$; and $\forall x \in \mathsf{dom}(\sigma) : \sigma(x) : \mathbf{T} = \Gamma(x)$.

(TPM-VAR): by $\mathsf{tpm}(\mathsf{T}, x, \Gamma)$ and the premises of the rule, we have $\Gamma = \{x : \mathsf{T}\}$. Moreover, $M : \mathsf{T}$ and $\mathsf{match}(M, x, \sigma)$ with $\mathsf{dom}(\sigma) = \{x\}$ and $\sigma(x) = M : \mathsf{T} = \Gamma(x)$.

(TPM-TAG): by $\mathsf{tpm}(f(\mathsf{T}), f(Q), \Gamma)$ and the premises of the rule, we have $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$; by the premises of the rule (TM-TAG) and $f(M) : f(\mathsf{T})$, we get $M : \mathsf{T}$ and $\mathsf{match}(f(M), f(Q), \sigma)$ implies $\mathsf{match}(M, Q, \sigma)$. Therefore, by induction we have $\forall x \in \mathsf{dom}(\sigma) : \sigma(x) : \Gamma(x)$.

(TPM-STAR$_2$): by $\mathsf{tpm}(*\mathsf{T}, Q \cdot LQ, \Gamma_1 \cup \Gamma_2)$ and the premises of the rule, we get $\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1)$ and $\mathsf{tpm}(*\mathsf{T}, LQ, \Gamma_2)$. By the premises of the rule (TM-STAR$_2$) and $M \cdot LM : *\mathsf{T}$ we get $M : \mathsf{T}$ and $LM : *\mathsf{T}$. $\mathsf{match}(M \cdot LM, Q \cdot LQ, \sigma_1 \cup \sigma_2)$ implies $\mathsf{match}(M, Q, \sigma_1)$ and $\mathsf{match}(LM, LQ, \sigma_2)$. By induction we have $(i = 1, 2)$ $\forall x \in \mathsf{dom}(\sigma_i) : \sigma_i(x) : \Gamma_i(x)$, thus $\forall x \in \mathsf{dom}(\sigma_1 \cup \sigma_2) : \sigma_1, \sigma_2(x) : \Gamma_1, \Gamma_2(x)$ (recall that $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset$ because of the linearity of patterns).

(TPM-LIST): This case is similar to the previous one.

(TPM-UNION): by $\mathsf{tpm}(\mathsf{T}' + \mathsf{T}'', Q, \Gamma)$ and the premises of the rule, we distinguish three cases:

- there exist Γ_0 and Γ_1 such that $\mathsf{tpm}(\mathsf{T}', Q, \Gamma_0)$, $\mathsf{tpm}(\mathsf{T}'', Q, \Gamma_1)$ and $\Gamma = \Gamma_0 + \Gamma_1$. By $M : \mathsf{T}' + \mathsf{T}''$ and the premises of the rule (TM-UNION), we get $M : \mathsf{T}'$ and/or $M : \mathsf{T}''$, by induction $\mathsf{match}(M, Q, \sigma)$ implies $\forall x \in \mathsf{dom}(\sigma) : \sigma(x) : \Gamma_0(x)$ and/or $\sigma(x) : \Gamma_1(x)$. From $\sigma(x) : \Gamma_0(x)$ and/or $\sigma(x) : \Gamma_1(x)$ and rule (TM-UNION) we obtain $\sigma(x) : \Gamma_0 + \Gamma_1(x) = \Gamma(x)$;
- there exists Γ_0 and for all Γ_1 we have $\mathsf{tpm}(\mathsf{T}', Q, \Gamma_0)$, not $\mathsf{tpm}(\mathsf{T}'', Q, \Gamma_1)$ and $\Gamma = \Gamma_0$. By $M : \mathsf{T}' + \mathsf{T}''$ and the premises of the rule (TM-UNION), we get $M : \mathsf{T}'$ and/or $M : \mathsf{T}''$. $M : \mathsf{T}'$ because if $M : \mathsf{T}''$ by $\mathsf{match}(M, Q, \sigma)$ and Lemma B.3 we have $\mathsf{tpm}(\mathsf{T}'', Q, \Gamma_1)$ and this is not the case; thus by induction $\mathsf{match}(M, Q, \sigma)$ implies $\forall x \in \mathsf{dom}(\sigma) : \sigma(x) : \Gamma_0(x) = \Gamma(x)$;

- there exists Γ_1 and for all Γ_0 we have $\mathsf{tpm}(\mathsf{T}'', Q, \Gamma_1)$, not $\mathsf{tpm}(\mathsf{T}', Q, \Gamma_0)$ and $\Gamma = \Gamma_1$. By $M : \mathsf{T}' + \mathsf{T}''$ and the premises of the rule (TM-UNION), we get $M : \mathsf{T}'$ and/or $M : \mathsf{T}''$. This case is similar to the previous.

□

Lemma B.5 *Suppose M closed and abstraction free. If $M : \mathsf{T}$ then $\mathsf{tpm}(\mathsf{T}, M, \emptyset)$.*

PROOF: The proof is straightforward by induction on the derivation of $M : \mathsf{T}$. □

Lemma B.6 *Let M be closed and abstraction free. If $\mathsf{tpm}(T, Q, \Gamma)$ and $M : \Gamma(x)$ then $\mathsf{tpm}(T, Q[M/x], \Gamma_{-\{x\}})$.*

PROOF: The proof is straightforward by induction on the derivation of $\mathsf{tpm}(\mathsf{T}, Q, \Gamma)$, the base case (TPM-VAR) relies on Lemma B.5. □

Lemma B.7 (Lemma 4) *(a) If $\Gamma, x : T \vdash P : ok$ and $\Gamma \vdash M : T$ then $\Gamma \vdash P[M/x] : ok$; (b) if $\Gamma, x : T \vdash N : S$ and $\Gamma \vdash M : T$ then $\Gamma \vdash N[M/x] : S$.*

PROOF: By induction on the depth of the derivation. We consider the last rule which we apply for deducing $\Gamma, x : \mathsf{T} \vdash P : ok$ or $\Gamma, x : \mathsf{T} \vdash N : \mathsf{S}$. The most interesting cases are the following:

(T-IN): by $\Gamma, x : \mathsf{T} \vdash a.A : ok$ and the premises of the rule, we get $a \in ch(\mathsf{T}')$ and $\Gamma, x : \mathsf{T} \vdash A : (\mathsf{T}')\mathsf{Abs}$. By induction $\Gamma \vdash A[M/x] : (\mathsf{T}')\mathsf{Abs}$, thus, by rule (T-IN), we have $\Gamma \vdash (a.A)[M/x] : ok$.

(T-OUT): by $\Gamma, x : \mathsf{T} \vdash \overline{u}\langle M' \rangle : ok$ and the premises of the rule, we get:

- $\Gamma, x : \mathsf{T} \vdash u \in ch(\mathsf{T}')$:

  $u \neq x$: then $\Gamma \vdash u[M/x] = u \in ch(\mathsf{T}')$;

  $u = x$: then $\mathsf{T} = ch(\mathsf{T}')$, $\Gamma \vdash x[M/x] = M : ch(\mathsf{T}')$ and $M = a$ for some channel $a : ch(\mathsf{T}')$. This implies that $a \in ch(\mathsf{T}'')$ with $ch(\mathsf{T}'') = ch(\mathsf{T}')$ or $ch(\mathsf{T}'') < ch(\mathsf{T}')$; that is, by the premises of the rule (SUB-SORT), $\mathsf{T}'' = \mathsf{T}'$ or $\mathsf{T}'' > \mathsf{T}'$. If $\mathsf{T}'' > \mathsf{T}'$, by $\Gamma, x : \mathsf{T} \vdash M' : \mathsf{T}'$ and by Lemma B.2 (subtyping), $\Gamma, x : \mathsf{T} \vdash M' : \mathsf{T}''$
- either $\Gamma, x : \mathsf{T} \vdash M' : \mathsf{T}'$, if $u \neq x$, or $\Gamma, x : \mathsf{T} \vdash M' : \mathsf{T}''$, with $\mathsf{T}'' > \mathsf{T}'$, if $u = x$, and by induction we deduce either $\Gamma \vdash M'[M/x] : \mathsf{T}'$ or $\Gamma \vdash M'[M/x] : \mathsf{T}''$.

Finally, by rule (T-OUT), $\Gamma \vdash (\overline{u}\langle M' \rangle)[M/x] : ok$.

(TM-VAR): $\Gamma, x : \mathsf{T} \vdash y : \mathsf{T}'$; we distinguish two cases:

$x = y$: by the premises of the rule: $\mathsf{T} < \mathsf{T}'$. By Lemma B.2 (subtyping) $\Gamma \vdash M : \mathsf{T}$ and $\mathsf{T} < \mathsf{T}'$ imply $\Gamma \vdash M : \mathsf{T}'$ and $\Gamma \vdash x[M/x] = M : \mathsf{T}'$;

$x \neq y$: in this case $\Gamma \vdash y[M/x] = y : \mathsf{T}'$;

(TM-ABS): by $\Gamma, x : \mathsf{T} \vdash (Q_{\tilde{x}} : \Gamma_Q)P : (\mathsf{S})\mathsf{Abs}$ and the premises of the rule, we get:

- $\mathsf{tpm}(\mathsf{S}, Q, \Gamma_1)$;
- $(\Gamma_1)_{|\tilde{x}} < \Gamma_Q$;
- $(\Gamma_1)_{|\tilde{y}} > (\Gamma, x : \mathsf{T})_{|\tilde{y}}$, $(\Gamma_1)_{|\tilde{y}}$ is abstraction free;
- $\Gamma, \Gamma_Q, x : \mathsf{T} \vdash P : ok$; by induction $\Gamma, \Gamma_Q \vdash P[M/x] : ok$.

We distinguish two cases:

$x \notin v(Q)$: then $Q[M/x] = Q$.

$x \in \tilde{y}$: $(\Gamma_1)_{|\tilde{y}} > (\Gamma, x : \mathsf{T})_{|\tilde{y}}$, $M : \mathsf{T}$ and Lemma 2 (subtyping) imply that $M : \Gamma_1(x)$, therefore, by Lemma B.6 (matching), we have $\mathsf{tpm}(\mathsf{S}, Q[M/x], (\Gamma_1)_{-\{x\}})$.

In conclusion, by rule (TM-ABS), $\Gamma \vdash ((Q[M/x])_{\tilde{x}} : \Gamma_Q)P[M/x] : (\mathsf{S})\mathsf{Abs}$.

□

Lemma B.8 (Lemma 5) *Let P and Q be annotated closed processes. If $P : ok$ and $P \equiv Q$ then $Q : ok$.*

PROOF: The proof is straightforward by distinguishing on structural rules. □

C Proofs of Section 5

Theorem C.1 (Theorem 4) *Suppose* $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma_0)$ *and* $\mathrm{fv}(M) \subseteq \mathrm{dom}(\Gamma_0)$. *If* $\mathrm{ti}_{\mathrm{P}}(P, \Gamma_0, \Gamma)$ *then* $\Gamma_0 \vdash P_\Gamma : ok$ *and if* $\mathrm{ti}_{\mathrm{M}}(T, M, \Gamma_0, \Gamma)$ *then* $\Gamma_0 \vdash M_\Gamma : T$.

PROOF: The proof proceeds by mutual induction on the depth of the derivation of $\mathrm{ti}_{\mathrm{P}}(P, \Gamma_0, \Gamma)$ and $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}, M, \Gamma_0, \Gamma)$ by distinguishing the last rule applied:

(TI$_P$-IN): by $\mathrm{ti}_{\mathrm{P}}(a.A, \Gamma_0, \Gamma)$ and the premises of the rule:

- $a \in ch(\mathsf{T})$;
- $\mathrm{ti}_{\mathrm{M}}((\mathsf{T})\mathsf{Abs}, A, \Gamma_0, \Gamma)$ and, by inductive hypothesis, $\Gamma_0 \vdash A_\Gamma : (\mathsf{T})\mathsf{Abs}$.

By rule (T-IN), $\Gamma_0 \vdash (a.A)_\Gamma : ok$.

(TI$_P$-OUT): by $\mathrm{ti}_{\mathrm{P}}(\overline{u}\langle M \rangle, \Gamma_0, \Gamma)$ and the premises of the rule:

- $\Gamma_0 \vdash u \in ch(\mathsf{T})$;
- $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}, M, \Gamma_0, \Gamma)$ and, by inductive hypothesis, $\Gamma_0 \vdash M_\Gamma : \mathsf{T}$.

By rule (T-OUT), $\Gamma_0 \vdash \overline{u}\langle M_\Gamma \rangle = (\overline{u}\langle M \rangle)_\Gamma : ok$.

(TI$_M$-ABS): by $\mathrm{ti}_{\mathrm{M}}((\mathsf{T})\mathsf{Abs}, (Q_{\tilde{x}})P, \Gamma_0, (\Gamma_1)_{|\tilde{x}} \cup \Gamma_2)$ and the premises of the rule:

- $\mathrm{tpm}(\mathsf{T}, Q, \Gamma_1)$ with $(\Gamma_1)_{|\tilde{y}} > (\Gamma_0)_{|\tilde{y}}$ and $(\Gamma_1)_{|\tilde{y}}$ abstraction free;
- $\mathrm{ti}_{\mathrm{P}}(P, \Gamma_0 \cup (\Gamma_1)_{|\tilde{x}}, \Gamma_2)$ and, by inductive hypothesis, $\Gamma_0, (\Gamma_1)_{|\tilde{x}} \vdash P_{\Gamma_2} : ok$.

By rule (TM-ABS), $\Gamma_0 \vdash (Q_{\tilde{x}} : (\Gamma_1)_{|\tilde{x}})P_{\Gamma_2} = ((Q_{\tilde{x}})P)_{(\Gamma_1)_{|\tilde{x}} \cup \Gamma_2} : (\mathsf{T})\mathsf{Abs}$.

□

For proving completeness, we need a subtyping and a narrowing property. The first lemma states that inference respects subtyping. The second states that a subtype can be used wherever a supertype is expected.

Lemma C.1 (subtyping for inference) *If* $\mathsf{T} < \mathsf{T}'$ *and* $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}, M, \Gamma, \Gamma_1)$ *then* $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}', M, \Gamma, \Gamma_1')$ *and* $\Gamma_1' < \Gamma_1$.

PROOF: We distinguish two cases:

$M = x$**:** by $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}, x, \Gamma, \Gamma_1)$ and and the premises of the rule (TI$_M$-VAR) we have $\Gamma_1 = \emptyset$ and $\Gamma(x) < \mathsf{T}$. By transitivity of subtyping $\Gamma(x) < \mathsf{T}'$, and, by rule (TI$_M$-VAR), $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}', x, \Gamma, \Gamma_1')$ and $\Gamma_1' = \emptyset$.

$M \neq x$**:** We proceed by induction on the sum of the depths of the derivation of $\mathsf{T} < \mathsf{T}'$ and $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}, M, \Gamma, \Gamma_1)$. We distinguish the last subtyping rule applied:

(Sub-Sort): by $ch(\mathsf{T}') < ch(\mathsf{T})$ and the premises of the rule, we get $\mathsf{T} < \mathsf{T}'$. In this case $M = a$ and, by the premises of the rule (TI_M-Value) and $\mathsf{ti}_\mathsf{M}(ch(\mathsf{T}'), a, \Gamma, \emptyset)$, we get $a : ch(\mathsf{T}')$. By definition $\exists \mathcal{S}' < ch(\mathsf{T}')$ s.t. $a \in \mathcal{S}'$; by transitivity $\mathcal{S}' < ch(\mathsf{T})$, therefore $a : ch(\mathsf{T})$. In conclusion, by rule (TI_M-Value), $\mathsf{ti}_\mathsf{M}(ch(\mathsf{T}), a, \Gamma, \emptyset)$.

(Sub-Top): $\mathsf{T} < \mathbf{T}$ and $\mathsf{ti}_\mathsf{M}(\mathsf{T}, M, \Gamma, \Gamma_1)$. By ($\text{TI}_M$-Top) and its premises, $\mathsf{ti}_\mathsf{M}(\mathbf{T}, M, \Gamma, \Gamma_1')$ and $\forall x \in \mathsf{bv}(M) : \quad \Gamma_1'(x) = \mathbf{\bot}$; hence, by rule (Sub-Bottom), $\Gamma_1'(x) < \Gamma_1(x)$.

(Sub-Bottom): $\mathbf{\bot} < \mathsf{T}$ and not $\mathsf{ti}_\mathsf{M}(\mathbf{\bot}, M, \ldots)$.

(Sub-Basic): by $\mathsf{bt1} < \mathsf{bt2}$ and the premises of the rule, we get $\mathsf{bt1} \prec \mathsf{bt2}$. By the premises of the rule (TI_M-Value) and $\mathsf{ti}_\mathsf{M}(\mathsf{bt1}, M, \Gamma, \Gamma_1)$, we get $\Gamma_1 = \emptyset$ and $M = v : \mathsf{bt1}$. By the subtyping relation on basic types, $v : \mathsf{bt2}$ and, by rule (TI_M-Value), $\mathsf{ti}_\mathsf{M}(\mathsf{bt2}, v, \Gamma, \Gamma_1')$ with $\Gamma_1' = \emptyset$.

(Sub-Tag): by $f(\mathsf{T}) < f(\mathsf{T}')$ and the premises of the rule, we get $\mathsf{T} < \mathsf{T}'$. By the premises of the rule (TI_M-Tag) and $\mathsf{ti}_\mathsf{M}(f(\mathsf{T}), f(M), \Gamma, \Gamma_1)$, we get $\mathsf{ti}_\mathsf{M}(\mathsf{T}, M, \Gamma, \Gamma_1)$ and, by inductive hypothesis, there is a Γ_1' such that $\mathsf{ti}_\mathsf{M}(\mathsf{T}', M, \Gamma, \Gamma_1')$ with $\Gamma_1' < \Gamma_1$. In conclusion, by rule (TI_M-Tag), $\mathsf{ti}_\mathsf{M}(f(\mathsf{T}'), f(M), \Gamma, \Gamma_1')$.

(Sub-Star$_1$): $[] < *\mathsf{T}$. By the premises of the rule (TI_M-Empty) and $\mathsf{ti}_\mathsf{M}([], M, \Gamma, \Gamma_1)$, we get $M = []$ and $\Gamma_1 = \emptyset$. By rule (TI_M-Star$_1$), $\mathsf{ti}_\mathsf{M}(*\mathsf{T}, [], \Gamma, \Gamma_1')$ and $\Gamma_1' = \emptyset$.

(Sub-Star$_2$): by $\mathsf{T}' \cdot \mathsf{LT} < *\mathsf{T}$ and the premises of the rule, we get $\mathsf{T}' < \mathsf{T}$ and $\mathsf{LT} < *\mathsf{T}$. By the premises of the rule (TI_M-List) and $\mathsf{ti}_\mathsf{M}(\mathsf{T}' \cdot \mathsf{LT}, M \cdot LM, \Gamma, \Gamma_1 \cup \Gamma_2)$:

- $\mathsf{ti}_\mathsf{M}(\mathsf{T}', M, \Gamma, \Gamma_1)$, and, by induction, there is a Γ_1' such that $\mathsf{ti}_\mathsf{M}(\mathsf{T}, M, \Gamma, \Gamma_1')$ with $\Gamma_1' < \Gamma_1$;
- $\mathsf{ti}_\mathsf{M}(\mathsf{LT}, LM, \Gamma, \Gamma_2)$, and, by induction, there is a Γ_2' such that $\mathsf{ti}_\mathsf{M}(*\mathsf{T}, LM, \Gamma, \Gamma_2')$ with $\Gamma_2' < \Gamma_2$.

In conclusion, by rule (TI_M-Star$_2$), $\mathsf{ti}_\mathsf{M}(*\mathsf{T}, M \cdot LM, \Gamma, \Gamma_1' \cup \Gamma_2')$ and $\Gamma_1' \cup \Gamma_2' < \Gamma_1 \cup \Gamma_2$.

(Sub-Star$_3$): by $*\mathsf{T} < *\mathsf{T}'$ and the premises of the rule, we get $\mathsf{T} < \mathsf{T}'$. We distinguish two cases:

$M = []$: by rule (TI_M-Star$_1$), $\mathsf{ti}_\mathsf{M}(*\mathsf{T}, [], \Gamma, \Gamma_1)$, $\mathsf{ti}_\mathsf{M}(*\mathsf{T}', [], \Gamma, \Gamma_1')$ and $\Gamma_1 = \Gamma_1' = \emptyset$;

$M = M \cdot LM$: by the premises of the rule (TI_M-Star$_2$) and $\mathsf{ti}_\mathsf{M}(*\mathsf{T}, M \cdot LM, \Gamma, \Gamma_1 \cup \Gamma_2)$:

- $\mathsf{ti}_\mathsf{M}(\mathsf{T}, M, \Gamma, \Gamma_1)$, and, by induction, there is a Γ_1' such that $\mathsf{ti}_\mathsf{M}(\mathsf{T}', M, \Gamma, \Gamma_1')$ with $\Gamma_1' < \Gamma_1$;
- $\mathsf{ti}_\mathsf{M}(*\mathsf{T}, LM, \Gamma, \Gamma_2)$, and, by induction, there is a Γ_2' such that $\mathsf{ti}_\mathsf{M}(*\mathsf{T}', LM, \Gamma, \Gamma_2')$ with $\Gamma_2' < \Gamma_2$.

In conclusion, by rule (TI_M-Star$_2$), $\mathsf{ti}_\mathsf{M}(*\mathsf{T}', M \cdot LM, \Gamma, \Gamma_1' \cup \Gamma_2')$ and $\Gamma_1' \cup \Gamma_2' < \Gamma_1 \cup \Gamma_2$.

(Sub-List): by $\mathsf{T} \cdot \mathsf{LT} < \mathsf{T}' \cdot \mathsf{LT}'$ and the premises of the rule, we get $\mathsf{T} < \mathsf{T}'$ and $\mathsf{LT} < \mathsf{LT}'$. By rule the premises of the rule (TI_M-List) and $\mathsf{ti}_\mathsf{M}(\mathsf{T} \cdot \mathsf{LT}, M \cdot LM, \Gamma, \Gamma_1 \cup \Gamma_2)$:

- $\mathsf{ti}_\mathsf{M}(\mathsf{T}, M, \Gamma, \Gamma_1)$, and, by induction, there is a Γ_1' such that $\mathsf{ti}_\mathsf{M}(\mathsf{T}', M, \Gamma, \Gamma_1')$ with $\Gamma_1' < \Gamma_1$;
- $\mathsf{ti}_\mathsf{M}(\mathsf{LT}, LM, \Gamma, \Gamma_2)$, and, by induction, there is a Γ_2' such that $\mathsf{ti}_\mathsf{M}(\mathsf{LT}', LM, \Gamma, \Gamma_2')$ with $\Gamma_2' < \Gamma_2$.

In conclusion, by rule (TI_M-List), $\mathsf{ti}_\mathsf{M}(\mathsf{T}' \cdot \mathsf{LT}', M \cdot LM, \Gamma, \Gamma_1' \cup \Gamma_2')$ and $\Gamma_1' \cup \Gamma_2' < \Gamma_1 \cup \Gamma_2$.

(Sub-Union$_1$): by $\mathsf{T} < \mathsf{T}' + \mathsf{T}''$ and the premises of the rule, we get $\mathsf{T} < \mathsf{T}'$ or $\mathsf{T} < \mathsf{T}''$, we distinguish three cases:

$\mathsf{T} < \mathsf{T}'$ **and** $\mathsf{T} < \mathsf{T}''$**:** $\mathsf{ti}_\mathsf{M}(\mathsf{T}, M, \Gamma, \Gamma_1)$ implies, by induction, that there are Γ' and Γ'' such that $\mathsf{ti}_\mathsf{M}(\mathsf{T}', M, \Gamma, \Gamma')$, $\mathsf{ti}_\mathsf{M}(\mathsf{T}'', M, \Gamma, \Gamma'')$, $\Gamma' < \Gamma_1$ and $\Gamma'' < \Gamma_1$. By rule (TI_M-Union) and (Sub-Union), $\mathsf{ti}_\mathsf{M}(\mathsf{T}' + \mathsf{T}'', M, \Gamma, \Gamma' + \Gamma'')$ and $\Gamma' + \Gamma'' < \Gamma_1$;

$\mathsf{T} < \mathsf{T}'$ **and not** $\mathsf{T} < \mathsf{T}''$**:** $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma_1)$ implies, by induction, that there is a Γ'_1 such that $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}',M,\Gamma,\Gamma'_1)$ and $\Gamma'_1 < \Gamma_1$, and, by rule (TI$_M$-UNION), $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}'+\mathsf{T}'',M,\Gamma,\Gamma'_1)$;

$\mathsf{T} < \mathsf{T}''$ **and** $\mathsf{T} < \mathsf{T}'$**:** in this case the proof proceeds similarly.

(SUB-UNION$_2$): by $\mathsf{T}'+\mathsf{T}'' < \mathsf{T}$ and the premises of the rule, we get $\mathsf{T}' < \mathsf{T}$ and $\mathsf{T}'' < \mathsf{T}$. By the premises of the rule (TI$_M$-UNION) and $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}'+\mathsf{T}'',M,\Gamma,\Gamma_1)$, we distinguish three cases:

$\mathrm{ti}_{\mathrm{M}}(\mathsf{T}',M,\Gamma,\Gamma_{1_1})$**,** $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}'',M,\Gamma,\Gamma_{1_2})$ **and** $\Gamma_1 = \Gamma_{1_1}+\Gamma_{1_2}$**:** by inductive hypothesis there is Γ'_1 such that $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma'_1)$ with $\Gamma'_1 < \Gamma_{1_1}$ and $\Gamma'_1 < \Gamma_{1_2}$; therefore, by rule (SUB-UNION$_1$), $\Gamma'_1 < \Gamma_{1_1}+\Gamma_{1_2} = \Gamma_1$;

$\mathrm{ti}_{\mathrm{M}}(\mathsf{T}',M,\Gamma,\Gamma_{1_1})$ **and for each** Γ_{1_2} **no** $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}'',M,\Gamma,\Gamma_{1_2})$**:** by induction there is a Γ'_1 such that $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma'_1)$ with $\Gamma'_1 < \Gamma_{1_1} = \Gamma_1$;

$\mathrm{ti}_{\mathrm{M}}(\mathsf{T}'',M,\Gamma,\Gamma_{1_2})$ **and for each** Γ_{1_1} **no** $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}',M,\Gamma,\Gamma_{1_1})$**:** By induction there is a Γ'_1 such that $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma'_1)$ with $\Gamma'_1 < \Gamma_{1_2} = \Gamma_1$.

□

Lemma C.2 (narrowing for inference) *If* $\Gamma' < \Gamma$ *then:*

- *if* $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma_1)$ *then* $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma',\Gamma'_1)$ *and* $\Gamma'_1 < \Gamma_1$*;*
- *if* $\mathrm{ti}_{\mathrm{P}}(P,\Gamma,\Gamma_1)$ *then* $\mathrm{ti}_{\mathrm{P}}(P,\Gamma',\Gamma'_1)$ *and* $\Gamma'_1 < \Gamma_1$.

PROOF: By mutual induction on the depth of the derivation of $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma_1)$ and $\mathrm{ti}_{\mathrm{P}}(P,\Gamma,\Gamma_1)$; the proof proceeds by distinguishing the last rule applied. The most interesting cases are rules (TI$_P$-OUT) and (TI$_M$-ABS):

(TI$_P$-OUT): by $\mathrm{ti}_{\mathrm{P}}(\overline{u}\langle M\rangle,\Gamma,\Gamma_1)$ and the premises of the rule, we get $\Gamma \vdash u \in \mathit{ch}(\mathsf{T})$ and $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma_1)$. We distinguish two cases:

$u = a$**:** $\Gamma \vdash u \in \mathit{ch}(\mathsf{T})$ implies $a \in \mathit{ch}(\mathsf{T})$. By inductive hypothesis, $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma_1)$ implies that there is a Γ'_1 such that $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma',\Gamma'_1)$ with $\Gamma'_1 < \Gamma_1$. By rule (TI$_P$-OUT), $\mathrm{ti}_{\mathrm{P}}(\overline{u}\langle M\rangle,\Gamma',\Gamma'_1)$;

$u = x$**:** $\Gamma(x) = \mathit{ch}(\mathsf{T}) > \Gamma'(x) = \mathit{ch}(\mathsf{T}')$ therefore $\mathsf{T}' > \mathsf{T}$, by premises of the rule (SUB-SORT). By Lemma C.1 (subtyping for inference), $\mathrm{ti}_{\mathrm{M}}(\mathsf{T},M,\Gamma,\Gamma_1)$ implies that there is a Γ'_1 such that $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}',M,\Gamma,\Gamma'_1)$ and $\Gamma'_1 < \Gamma_1$. In conclusion, by rule (TI$_P$-OUT), $\Gamma' \vdash u : \mathit{ch}(\mathsf{T}')$ and $\mathrm{ti}_{\mathrm{M}}(\mathsf{T}',M,\Gamma,\Gamma'_1)$ imply $\mathrm{ti}_{\mathrm{P}}(\overline{u}\langle M\rangle,\Gamma',\Gamma'_1)$ with $\Gamma'_1 < \Gamma_1$.

(TI$_M$-ABS): by $\mathrm{ti}_{\mathrm{M}}((\mathsf{T})\mathsf{Abs},(Q_{\tilde{x}})P,\Gamma,\Gamma_1)$ and the premises of the rule, we have:

- there is a Γ_2 such that $\mathsf{tpm}(\mathsf{T},Q,\Gamma_2)$, $(\Gamma_2)_{\lfloor\tilde{y}} > (\Gamma)_{\lfloor\tilde{y}}$ and $(\Gamma_2)_{\lfloor\tilde{y}}$ is abstraction free;
- there is a Γ_3 such that $\mathrm{ti}_{\mathrm{P}}(P,\Gamma\cup(\Gamma_2)_{|\tilde{x}},\Gamma_3)$;
- $\Gamma_1 = (\Gamma_2)_{|\tilde{x}}\cup\Gamma_3$.

By transitivity of subtyping $(\Gamma_2)_{\lfloor\tilde{y}} > (\Gamma')_{\lfloor\tilde{y}}$. Moreover, by inductive hypothesis, there is a Γ'_3 such that $\mathrm{ti}_{\mathrm{P}}(P,\Gamma'\cup(\Gamma_2)_{|\tilde{x}},\Gamma'_3)$ with $\Gamma'_3 < \Gamma_3$. In conclusion, by rule (TI$_M$-ABS), $\mathrm{ti}_{\mathrm{M}}((\mathsf{T})\mathsf{Abs},(Q_{\tilde{x}})P,\Gamma',\Gamma'_1)$ with $\Gamma'_1 = (\Gamma_2)_{|\tilde{x}}\cup\Gamma'_3 < \Gamma_1 = (\Gamma_2)_{|\tilde{x}}\cup\Gamma_3$.

□

Theorem C.2 (Theorem 5) *Suppose* $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma')$ *and* $\mathrm{fv}(M) \subseteq \mathrm{dom}(\Gamma')$.

- *If* $\Gamma_0 \vdash P_{\Gamma'} : \mathit{ok}$ *then there is* Γ *s.t.* $\mathrm{ti}_{\mathrm{P}}(P,\Gamma_0,\Gamma)$ *and* $\Gamma'_{\lfloor\mathrm{bv}(P)} > \Gamma$.
- *If* $\Gamma_0 \vdash M_{\Gamma'} : \mathcal{T}$ *then there is* Γ *s.t.* $\mathrm{ti}_{\mathrm{M}}(\mathcal{T},M,\Gamma_0,\Gamma)$ *and* $\Gamma'_{\lfloor\mathrm{bv}(M)} > \Gamma$.

PROOF: The proof proceeds by mutual induction on the depth of the derivation of $\Gamma_0 \vdash P_{\Gamma'} : ok$ and $\Gamma_0 \vdash M_{\Gamma'} : \mathsf{T}$, by distinguishing the last rule applied. The most interesting cases are:

(T-IN): by $\Gamma_0 \vdash (a.A)_{\Gamma'} = a.A_{\Gamma'} : ok$ and the premises of the rule:

- $a \in ch(\mathsf{T})$;
- $\Gamma_0 \vdash A_{\Gamma'} : (\mathsf{T})\mathsf{Abs}$; by inductive hypothesis there is a Γ such that $\mathsf{ti}_{\mathrm{M}}((\mathsf{T})\mathsf{Abs}, A, \Gamma_0, \Gamma)$ with $\Gamma < \Gamma'_{|\mathrm{bv}(A)}$.

By rule (TI$_P$-IN), $\mathsf{ti}_{\mathrm{P}}(a.A, \Gamma_0, \Gamma)$ with $\Gamma < \Gamma'_{|\mathrm{bv}(A)}$.

(T-OUT): by $\Gamma_0 \vdash (\overline{u}\langle M\rangle)_{\Gamma'} = \overline{u}\langle M_{\Gamma'}\rangle : ok$ and the premises of the rule:

- $\Gamma_0 \vdash u \in ch(\mathsf{T})$;
- $\Gamma_0 \vdash M_{\Gamma'} : \mathsf{T}$ and, by inductive hypothesis, there is a Γ such that $\mathsf{ti}_{\mathrm{M}}(\mathsf{T}, M, \Gamma_0, \Gamma)$ with $\Gamma < \Gamma'_{\mathrm{bv}(M)}$.

By rule (TI$_P$-OUT), $\mathsf{ti}_{\mathrm{P}}(\overline{u}\langle M\rangle, \Gamma_0, \Gamma)$ with $\Gamma < \Gamma'_{|\mathrm{bv}(M)}$.

(TM-ABS): by $\Gamma_0 \vdash ((Q_{\tilde{x}})P)_{\Gamma'} = (Q : \Gamma'_{|\tilde{x}}).P_{\Gamma'_{|\mathrm{bv}(P)}} : (\mathsf{T})\mathsf{Abs}$ and the premises of the rule:

- there is Γ_1 such that $\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1)$ with $(\Gamma_1)_{|\tilde{x}} < \Gamma'_{|\tilde{x}}$, $(\Gamma_1)_{\lfloor\tilde{y}} > (\Gamma_0)_{\lfloor\tilde{y}}$ and $(\Gamma_1)_{\lfloor\tilde{y}}$ abstraction free (recall that $\mathrm{bv}(Q) = \tilde{x}$);
- $\Gamma_0, \Gamma'_{|\tilde{x}} \vdash P_{\Gamma'_{|\mathrm{bv}(P)}} : ok$.

By inductive hypothesis, $\Gamma_0, \Gamma'_{|\tilde{x}} \vdash P_{\Gamma'_{|\mathrm{bv}(P)}} : ok$ implies that there is a Γ_2 such that $\mathsf{ti}_{\mathrm{P}}(P, \Gamma_0 \cup \Gamma'_{|\tilde{x}}, \Gamma_2)$ with $\Gamma_2 < \Gamma'_{|\mathrm{bv}(P)}$. By Lemma C.2 (narrowing for type inference) and $(\Gamma_1)_{|\tilde{x}} < \Gamma'_{|\tilde{x}}$ it holds that there is a Γ'_2 such that $\mathsf{ti}_{\mathrm{P}}(P, \Gamma_0 \cup (\Gamma_1)_{|\tilde{x}}, \Gamma'_2)$ with $\Gamma'_2 < \Gamma_2 < \Gamma'_{|\mathrm{bv}(P)}$. Finally, by rule (TI$_M$-ABS), $\mathsf{ti}_{\mathrm{M}}((\mathsf{T})\mathsf{Abs}, (Q_{\tilde{x}})P, \Gamma_0, (\Gamma_1)_{|\tilde{x}} \cup \Gamma'_2)$ with $(\Gamma_1)_{|\tilde{x}} \cup \Gamma'_2 < \Gamma'_{|\mathrm{bv}(Q)\cup\mathrm{bv}(P)}$.

□

D Proofs of Section 6

Proofs of Lemmas B.2, B.4 and B.7 carry over essentially unchanged to the language with dynamic abstractions.

Theorem D.1 (extension of Theorem 2) *Let P be an annotated closed process and suppose $P : ok$, then P is dynamically safe.*

PROOF: By induction on the derivation on $P : ok$. The unique change is in rule (T-IN) when $a.D : ok$. The latter implies:

- $a \in ch(\mathsf{T})$;
- $\vdash D : (\mathsf{T})\mathsf{Abs}$.

By the premises of the rule (TM-ABS-D):

- $D = (\!|Q_{\tilde{x}} : \Gamma_Q|\!)P$;
- $\mathsf{tpm}(\mathsf{T}, Q, \Gamma_1)$;
- $(\Gamma_1)_{|\tilde{x}} \lesseqgtr \Gamma_Q$, that is $\forall y \in \tilde{x}$ exists a consistent type T' s.t. $\mathsf{T}' < \Gamma_Q(y)$ and $\mathsf{T}' < \Gamma_1(y)$.

Consider the message M obtained by replacing in Q every variable $y \in \tilde{x}$ by some message $M'_y : \mathsf{T}'$ (M'_y exists because T' is consistent). Obviously $\mathsf{match}(M, Q, \sigma)$ and $\sigma(y) = M'_y : \mathsf{T}' < \Gamma_Q(y)$, $\forall y \in \tilde{x}$. Therefore Q is dynamically T-consistent and by Definition 15 the process $a.D$ is dynamically safe. □

Theorem D.2 (Extension of Theorem 3) *Let P be an annotated closed process. If $P : ok$ and $P \to P'$ then $P' : ok$.*

PROOF: By rule (COM-D) by $\overline{a}\langle M\rangle \,|\, \sum_{i\in I} a_i.A_i \to P\sigma$ and the premises of the rule, we have that for some $j \in I$:

- $a = a_j$;
- $A_j = (\!|Q_{\tilde{x}} : \Gamma_Q|\!)P$;
- $\mathsf{match}(M, Q, \sigma)$;
- $\forall y \in \mathsf{dom}(\sigma) : \quad \sigma(y) : \Gamma_Q(y)$.

We have to prove that $P\sigma : ok$. From $\overline{a}\langle M\rangle \,|\, \sum_{i\in I} a_i.A_i : ok$ and the premises of the rule (T-PAR), we have $\sum_{i\in I} a_i.A_i : ok$. Hence, by (T-SUM), (T-INP) and (TM-ABS-D) we have $A_j : (\mathsf{T})\mathsf{Abs}$ and $\Gamma_Q \vdash P : ok$. Recalling that $\forall y \in \mathsf{dom}(\sigma)$ we have $\sigma(y) : \Gamma_Q(y)$, by Lemma B.7 (substitution) we obtain $P\sigma : ok$. □

Corollary D.1 (Corollary 3) *Let P be an annotated closed process. If $P : ok$ and $P \to^* P'$ then P' is dynamically safe.*

PROOF: By Theorem D.1 and D.2. □

E Proofs of Section 7

We need two preliminary lemmas.

Lemma E.1 *Let S be a finite of set names and suppose that for no $c \in S$ it holds that $c : \mathsf{bool}$. Define*

$$A \stackrel{\triangle}{=} (\nu t)\big(\big(\prod_{\{c\in S\}} \bar{t}\langle c\rangle \,|\, !t.(\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x\rangle\big) \ \mathsf{else}\ (\bar{t}\langle \mathtt{ff}\rangle \,|\, t.(\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x\rangle)\big)\,.$$

1. *If there is a $c \in S$ such that $c : \mathsf{T}$ then $[\![\prod_{\{c\in S|c:\mathsf{T}\}} \overline{a}\langle c\rangle]\!]^E \lesssim A$;*
2. *if there are no $c \in S$ such that $c : \mathsf{T}$ then $[\![\overline{a}\langle \mathtt{ff}\rangle]\!]^E \lesssim A$.*

PROOF:

1. Let $B \triangleq [\![\prod_{\{c\in S|c:\mathsf{T}\}} \overline{a}\langle c\rangle]\!]^E$. Let us first prove that $B \dot{\lesssim} A$. It is enough to prove that the relation $\mathcal{R}$ defined below is a barbed expansion. Define, for each set of names S',

$$A_{S'} \stackrel{\triangle}{=} (\nu t)(\prod_{\{c\in S|c\notin S'\}} \bar{t}\langle c\rangle \,|\, !t.(\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x\rangle \,|\, \prod_{\{c\in S'\}} \overline{a}\langle c\rangle)\,.$$

   and

$$\mathcal{R} \stackrel{\triangle}{=} \{\langle A, B\rangle\} \cup \big\{\langle A_{S'}, B\rangle \,\big|\, S' \subseteq \{c \in S \,\big|\, c : \mathsf{T}\}\big\}\,.$$

   Both A and any $A_{S'}$ can only reduce by communicating on t, hence generating a new output on a and evolving into some $A_{S''}$: then the pair $\langle A_{S''}, B\rangle$ is still in $\mathcal{R}$. Moreover, the only barb

of A, $A_{S'}$ and B is $\downarrow_a$. This proves that that the relation $\mathcal{R}$ is a barbed expansion, hence $\mathcal{R} \subseteq \dot{\lesssim}$.

The proof then proceeds by closing $\dot{\lesssim}$ under each static context. That is, one proves that for each $\tilde{b} \in \mathcal{N}$ and D it holds that $(\nu\tilde{b})(A \mid D) \dot{\gtrsim} (\nu\tilde{b})(B \mid D)$: this is proved by exhibiting a relation $\mathcal{R}' \subseteq \dot{\lesssim}$ containing the pairs $\langle (\nu\tilde{b})(A \mid D), (\nu\tilde{b})(B \mid D) \rangle$, for each $\tilde{b} \in \mathcal{N}$ and D, and proving it a barbed expansion.

Let us define, for any $S', S'' \subseteq S$ s.t. $S' \cap S'' = \emptyset$, the following processes

$$B_{S'} \triangleq [\![\textstyle\prod_{\{c \in S \setminus S' \mid c:\mathsf{T}\}} \overline{a}\langle c \rangle]\!]^E$$
$$A_{S',S''} \triangleq (\nu t)\big(\textstyle\prod_{\{c \in S \setminus (S' \cup S'')\}} \bar{t}\langle c \rangle \mid !t.(\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x \rangle \mid \prod_{\{c \in S''\}} \overline{a}\langle c \rangle \big) \ .$$

The relation $\mathcal{R}'$ is defined as follows:

$$\mathcal{R}' \triangleq \big\{ \langle (\nu\tilde{b})(A \mid D), (\nu\tilde{b})(B \mid D) \rangle ,$$
$$\langle (\nu\tilde{b})(A_{S',S''} \mid D), (\nu\tilde{b})(B_{S'} \mid D) \rangle \,\big|\, S' \subseteq \{c \in S | c : \mathsf{T}\} \text{ and } S'' \subseteq \{c \in S \setminus S' | c : \mathsf{T}\} \big\} .$$

$\mathcal{R}'$ is an expansion relation. As to barbs, the only barb of $A_{S',S''}$ and $B_{S'}$ are $A_{S',S''} \Downarrow_a$ and $B_{S'} \downarrow_a$, while D gives rise to the same barbs on both components of the pairs. As for reductions, each reduction of a process in a pair of $\mathcal{R}'$ can be only derived by a reduction of one of its subcomponents as follows:

- $A \to A_{\emptyset,\{c\}}$, with $c \in S, c : \mathsf{T}$. Then $\langle (\nu\tilde{b})(A_{\emptyset,\{c\}} \mid D), (\nu\tilde{b})(B_{\emptyset} \mid D) \rangle \in \mathcal{R}'$ by definition;
- $A_{S',S''} \to A_{S',S'''}$, with $S'' \subset S'''$. Then $\langle (\nu\tilde{b})(A_{S',S'''} \mid D), (\nu\tilde{b})(B_{S'} \mid D) \rangle \in \mathcal{R}'$ by definition;
- $D \to D'$. Then $\langle (\nu\tilde{b})(A_{S',S''} \mid D'), (\nu\tilde{b})(B_{S'} \mid D') \rangle \in \mathcal{R}'$ by definition;
- the synchronizations $A_{S',S''} \mid D \to A_{S' \cup \{c\}, S'' \setminus \{c\}} \mid D'$ and $B_{S'} \mid D \to B_{S' \cup \{c\}} \mid D'$ match up with each other, as $\langle (\nu\tilde{b})(A_{S' \cup \{c\}, S'' \setminus \{c\}} \mid D'), (\nu\tilde{b})(B_{S' \cup \{c\}} \mid D') \rangle \in \mathcal{R}'$ by definition.

2. Again, we first prove that $[\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \dot{\lesssim} A$. It is enough to note that $\mathcal{R} \subseteq \dot{\lesssim}$, where $\mathcal{R}$ is defined as follows:
$$\mathcal{R} \triangleq \big\{ \langle A, [\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \rangle ; \langle A', [\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \rangle , \langle A'', [\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \rangle \big\}$$
with
$$A' \triangleq (\nu t)(\bar{t}\langle \mathtt{ff} \rangle \mid t.(\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x \rangle) \text{ and } A'' \triangleq (\nu t)(\overline{a}\langle \mathtt{ff} \rangle) \ .$$
We proceed by proving that $[\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \lesssim A$ by closing $\dot{\lesssim}$ for each context. We prove that for each $\tilde{b} \in \mathcal{N}$ and D it holds that $(\nu\tilde{b})(A \mid D) \dot{\gtrsim} (\nu\tilde{b})([\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \mid D)$. It suffices to note that the relation $\mathcal{R}$ below is a $\dot{\lesssim}$.
$$\mathcal{R} = \big\{ \langle (\nu\tilde{b})(A \mid D), (\nu\tilde{b})([\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \mid D) \rangle , \langle (\nu\tilde{b})(A' \mid D), (\nu\tilde{b})([\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \mid D) \rangle$$
$$\langle (\nu\tilde{b})(A'' \mid D), (\nu\tilde{b})([\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \mid D) \rangle , \langle (\nu\tilde{b})((\nu t)\mathbf{0} \mid D), (\nu\tilde{b})(D) \rangle \big\}$$
Hence, $[\![\overline{a}\langle \mathtt{ff} \rangle]\!]^E \lesssim A$.

□

Lemma E.2 (Proof of Lemma 6) *Suppose $P \in \mathcal{P}^E$.*

1. *$P \to P'$ implies that $\exists R$ such that $[\![P]\!]^E \to R$ and $[\![P']\!]^E \lesssim R$;*
2. *$[\![P]\!]^E \to R$ implies that $\exists P' \in \mathrm{XPi}^E$ such that $P \to P'$ and $[\![P']\!]^E \lesssim R$;*

3. $P \downarrow_a$ *if and only if* $[\![P]\!]^E \downarrow_a$.

PROOF:

1. The proof proceeds by induction on the derivation of $P \to P'$. The most interesting cases are the following. In the others, each reduction from P is matched by the same reduction from $[\![P]\!]^E$.

   (PUB): $d\langle S\rangle \,|\, \overline{d}^{(p)}\langle c\rangle \to d\langle S \cup \{c\}\rangle$.

   $$[\![d\langle S\rangle \,|\, \overline{d}^{(p)}\langle c\rangle]\!]^E = D(S) \,|\, \overline{d}\langle \mathtt{publish}(c)\rangle \to D(S \cup \{c\}) = [\![d\langle S \cup \{c\}\rangle]\!]^E$$

   (QUERY-T): from $d\langle S\rangle \,|\, \overline{d}^{(q)}\langle \mathsf{T}, a\rangle \to d\langle S\rangle \,|\, \prod_{\{c \in S | c:\mathsf{T}\}} \overline{a}\langle c\rangle$ and the premises of the rule, we have $\exists c \in S$ such that $c : \mathsf{T}$.

   $$[\![d\langle S\rangle \,|\, \overline{d}^{(q)}\langle \mathsf{T}, a\rangle]\!]^E = D(S) \,|\, \overline{d}\langle \mathtt{query}((\!|?z : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle z\rangle)\rangle$$
   $$\to$$
   $$A \,|\, D(S)$$

   with

   $$A \triangleq (\nu t)\big(\big(\prod_{\{c' \in S\}} \bar{t}\langle c'\rangle \,|\, !t.(\!|?z : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle z\rangle\big) \text{ else } \big(\bar{t}\langle \mathtt{ff}\rangle \,|\, t.(\!|?z : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle z\rangle\big)\big),$$

   hence by Lemma E.1 (1) $[\![\prod_{\{c \in S | c:\mathsf{T}\}} \overline{a}\langle c\rangle]\!]^E \lesssim A$ and, by definition of $\lesssim$, $[\![d\langle S\rangle \,|\, \prod_{\{c \in S | c:\mathsf{T}\}} \overline{a}\langle c\rangle]\!]^E \lesssim [\![d\langle S\rangle]\!]^E \,|\, A = D(S) \,|\, A$.

   (QUERY-F): the proof proceeds as in the previous case by applying Lemma E.1 (2).

   (CTX): by $(\nu \tilde{d})(P \,|\, R) \to (\nu \tilde{d})(P' \,|\, R)$ and the premises of the rule, we have $P \to P'$. By inductive hypothesis, there is a P'' such that $[\![P]\!]^E \to P''$ with $[\![P']\!]^E \lesssim P''$. By (CTX), $(\nu \tilde{d})([\![P]\!]^E \,|\, [\![R]\!]^E) \to (\nu \tilde{d})(P'' \,|\, [\![R]\!]^E)$ and $(\nu \tilde{d})([\![P']\!]^E \,|\, [\![R]\!]^E) \lesssim (\nu \tilde{d})(P'' \,|\, [\![R]\!]^E)$ by definition of $\lesssim$.

2. The proof is straightforward by induction on the derivation $[\![P]\!]^E \to R$. The proof proceeds by distinguishing the last reduction rule applied. The most interesting cases are the following:

   (COM): we consider the following cases:

   - $[\![P]\!]^E = D(S) \,|\, \overline{d}\langle \mathtt{publish}(c)\rangle \to D(S \cup \{c\})$. $P = d\langle S\rangle \,|\, \overline{d}^{(p)}\langle c\rangle \to d\langle S \cup \{c\}\rangle$ and $[\![d\langle S \cup \{c\}\rangle]\!]^E = D(S \cup \{c\})$.
   - $[\![P]\!]^E = D(S) \,|\, \overline{d}\langle \mathtt{query}((\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x\rangle)\rangle \to D(S) \,|\, A$ with $A \triangleq (\nu t)\big(\big(\prod_{c \in S} \bar{t}\langle c\rangle \,|\, !t.(\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x\rangle\big)\mathrm{else}\big(\bar{t}\langle \mathtt{ff}\rangle \,|\, t.(\!|?x : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle x\rangle\big)\big)$.

     Suppose there is at least one $c \in S$ such that $c : \mathsf{T}$. By (QUERY-T), $P = d\langle S\rangle \,|\, \overline{d}^{(q)}\langle \mathsf{T}, a\rangle \to d\langle S\rangle \,|\, \prod_{\{c \in S | c:\mathsf{T}\}} \overline{a}\langle c\rangle$. By Lemma E.1 (1), $[\![\prod_{\{c \in S | c:\mathsf{T}\}} \overline{a}\langle c\rangle]\!]^E \lesssim A$ and $[\![d\langle S\rangle \,|\, \prod_{\{c \in S | c:\mathsf{T}\}} \overline{a}\langle c\rangle]\!]^E \lesssim [\![d\langle S\rangle]\!]^E \,|\, A$ by definition of $\lesssim$.
     Suppose that there is no $c \in S$ such that $c : \mathsf{T}$. By (QUERY-F), $P \to d\langle S\rangle \,|\, \overline{a}\langle \mathtt{ff}\rangle$. By Lemma E.1 (2), $[\![\overline{a}\langle \mathtt{ff}\rangle]\!]^E \lesssim A$ and $[\![d\langle S\rangle \,|\, \overline{a}\langle \mathtt{ff}\rangle]\!]^E \lesssim [\![d\langle S\rangle]\!]^E \,|\, A$ again by definition of $\lesssim$.
   - in the other cases $P \to P'$ with $[\![P']\!]^E \equiv R$.

   (CTX): by $[\![(\nu \tilde{d})(R \,|\, P)]\!]^E \to (\nu \tilde{d})(R' \,|\, [\![P]\!]^E)$ and the premises of the rule, we have $[\![R]\!]^E \to R'$. By inductive hypothesis, $R \to R''$ with $[\![R'']\!]^E \lesssim R'$. $(\nu \tilde{d})(R \,|\, P) \to (\nu \tilde{d})(R'' \,|\, P)$ by (CTX) and $[\![(\nu \tilde{d})(R'' \,|\, P)]\!]^E \lesssim (\nu \tilde{d})(R' \,|\, [\![P]\!]^E)$ by definition of $\lesssim$.

3. The proof is straightforward by induction on the structure of P. The interesting cases are the following:

   $P = d\langle S\rangle$: $[\![d\langle S\rangle]\!]^E = D(S)$ and both have no barbs;

   $P = \overline{d}^{(p)}\langle c\rangle$: $[\![\overline{d}^{(p)}\langle c\rangle]\!]^E = \overline{d}\langle \mathtt{publish}(c)\rangle \downarrow_d$ and $\overline{d}^{(p)}\langle c\rangle \downarrow_d$ by definition of barb;

   $P = \overline{d}^{(q)}\langle \mathsf{T}, a\rangle$: $[\![\overline{d}^{(q)}\langle \mathsf{T}, a\rangle]\!]^E = \overline{d}\langle \mathtt{query}((\!|?z : \mathsf{T} + \mathsf{bool}|\!)\overline{a}\langle z\rangle)\rangle \downarrow_d$ and $\overline{d}^{(q)}\langle \mathsf{T}, a\rangle \downarrow_d$ by definition of barb.

□

Proofs of Theorem 6 and Corollary 4 proceed as the proofs of Theorem A.1 and Corollary A.1 by using Lemma E.2 and $\approx_{[\![\cdot]\!]^E}$ instead of Proposition A.1 and $\approx_{\langle\cdot\rangle}$, respectively.

F Service composition: an extended example

We propose here an example of service composition in a somewhat more realistic scenario than those considered in previous sections. A Search Web service offers various operations, among which a book price search operation, as described by the fragment of WSDL document below.

```
<description ...>
        ...
        <interface  name = "SearchServices" >
                 ...
                 <operation name="opSearchPrice"
                               pattern="http://www.w3.org/ns/wsdl/in-out" >
                        <input messageLabel="In"
                               element="searchPrice" />
                        <output messageLabel="Out"
                               element="searchPriceReply" />
                </operation>
                ...
        </interface>
        ...
</description>
```

The operation `opSearchPrice` receives in input a book title, contacts an on-line bookstore for obtaining its price and returns it to the client. The type associated to message `searchPrice` can be defined, using XPi syntax for conciseness, as `book[title(string)]`, while the type associated to message `searchPriceReply` is `bookPriceReply[title(string),store(string),price(real)]`.

The on-line store is described by the following WSDL document fragment.

```
<description ...>
        ...
        <interface  name = "BookStoreServices" >
                 ...
                <operation name="opCheckAvailability"
                               pattern="http://www.w3.org/ns/wsdl/in-out">
                        <input messageLabel="In"
                               element="title" />
                        <output messageLabel="Out"
                               element="checkReply" />
                </operation>
                <operation name="opBookPrice"
                               pattern="http://www.w3.org/ns/wsdl/in-out" >
                        <input messageLabel="In"
```

```
                        element="title" />
                <output messageLabel="Out"
                        element="priceReply" />
            </operation>
        </interface>
        ...
</description>
```

The bookstore offers at least two operations. A check availability operation, to check the availability of a given book, and a book price operation that lets a client know the price of a specified book. Message `title` is supposed to be of type `book[title(string)]`, message `checkReply` and `priceReply` of type `availabilityReply[title(string),copies(int)]` and `priceReply[title(string),price(real)]` respectively.

We proceed now by defining two XPi processes that implement these services. Note that, in XPi, the actual types of operation messages we consider are slightly different from those defined in the WSDL documents above: indeed, the sole difference is that in XPi the input message of each operation also contains a reply channel, where the caller waits for the reply (these reply channels could be automatically inserted by a compiler). So the types we are going to use are the following:

- T_{sp} is the type of input messages for the search price operation (WebSearch). Messages of this type contains not only the title of the book, but also a reply channel of type $ch(\mathsf{T}_{rsp})$:

$$\mathsf{T}_{sp} \triangleq \mathtt{bookPrice}[\mathtt{title}(\mathsf{string}), \mathtt{rep}(ch(\mathsf{T}_{rsp}))]$$

- T_{rsp} is the type of reply messages informing the client about the title, the bookstore and the price:

$$\mathsf{T}_{rsp} \triangleq \mathtt{bookPriceReply}[\mathtt{title}(\mathsf{string}), \mathtt{store}(\mathsf{string}), \mathtt{price}(\mathsf{real})]$$

- T_{ca} is the type of messages received by the check availability operation (Bookstore). Again, this operation receives not only the title of the book, but also a reply channel carrying messages of type T_{rca}:

$$\mathsf{T}_{ca} \triangleq \mathtt{bookAvailability}[\mathtt{title}(\mathsf{string}), \mathtt{rep}(ch(\mathsf{T}_{rca}))]$$

- T_{rca} is the type of reply messages informing the client about the availability and the title of the book:

$$\mathsf{T}_{rca} \triangleq \mathtt{availabilityReply}[\mathtt{title}(\mathsf{string}), \mathtt{copies}(\mathsf{int})]$$

- T_{bp} is the type of messages received by the check book price operation (Bookstore). Again, this operation receives not only the title of the book, but also a reply channel carrying messages of type T_{rbp}:

$$\mathsf{T}_{bp} \triangleq \mathtt{bookPrice}[\mathtt{title}(\mathsf{string}), \mathtt{rep}(ch(\mathsf{T}_{rbp}))]$$

- T_{rbp} is the type of reply messages informing the client about the price and the title of the book:

$$\mathsf{T}_{rbp} \triangleq \mathtt{priceReply}[\mathtt{title}(\mathsf{string}), \mathtt{price}(\mathsf{real})] \ .$$

We assume that the bookstore has a private channel implementing its catalog

$$cat : ch([\mathtt{title}(\mathsf{string}), \mathtt{copies}(\mathsf{int}), \mathtt{price}(\mathsf{real})])$$

which can be queried to know the price/availability of each book. For simplicity, we assume that each title occurs exactly once in the catalog.

The bookstore service can be defined in XPi as follows:

$$
\begin{array}{l}
B \stackrel{\triangle}{=} (\nu cat)\big(\prod_{i\in I}\overline{cat}\langle[\mathtt{title}(t_i),\mathtt{copies}(c_i),\mathtt{price}(p_i)]\rangle \\
\qquad \mid !opCheckAvailability.(\mathtt{bookAvailability}[\mathtt{title}(?x_t:\mathsf{string}),\mathtt{rep}(?x_r:ch(\mathsf{T}_{rca}))]) \\
\qquad\quad cat.([\mathtt{title}(x_t),\mathtt{copies}(?y:\mathsf{int}),\mathtt{price}(?z:\mathsf{real})]) \\
\qquad\qquad (\ \overline{cat}\langle[\mathtt{title}(x_t),\mathtt{copies}(y),\mathtt{price}(z)]\rangle \\
\qquad\qquad\ \mid \overline{x}_r\langle\mathtt{availabilityReply}[\mathtt{title}(x_t),\mathtt{copies}(y)]\rangle) \\
\qquad \mid !opBookPrice.(\mathtt{bookPrice}[\mathtt{title}(?x_t:\mathsf{string}),\mathtt{rep}(?x_r:ch(\mathsf{T}_{rbp}))])) \\
\qquad\quad cat.([\mathtt{title}(x_t),\mathtt{copies}(?y:\mathsf{int}),\mathtt{price}(?z:\mathsf{real})]) \\
\qquad\qquad (\ \overline{cat}\langle[\mathtt{title}(x_t),\mathtt{copies}(y),\mathtt{price}(z)]\rangle \\
\qquad\qquad\ \mid \overline{x}_r\langle\mathtt{priceReply}[\mathtt{title}(x_t),\mathtt{price}(z)]\rangle) \\
\qquad \mid !opBookOrder.\cdots \\
\qquad \mid \ldots
\end{array}
$$

with $opCheckAvailability : ch(\mathsf{T}_{ca})$ and $opBookPrice : ch(\mathsf{T}_{bp})$.

The Search Web service could be defined as follows:

$$
\begin{array}{l}
S \stackrel{\triangle}{=} !opSearchPrice.(\mathtt{bookPrice}[\mathtt{title}(?x_t:\mathsf{string}),\mathtt{rep}(?x_r:ch(\mathsf{T}_{rsp}))]) \\
\qquad (\nu s)\big(\overline{opBookPrice}\langle\mathtt{bookPrice}[\mathtt{title}(x_t),\mathtt{rep}(s)]\rangle \\
\qquad\quad \mid s.(\mathtt{priceReply}[\mathtt{title}(x_t),\mathtt{price}(?z:\mathsf{real})]) \\
\qquad\qquad\quad \overline{x}_r\langle\mathtt{bookPriceReply}[\mathtt{title}(x_t),\mathtt{store}(B),\mathtt{price}(z)]\rangle\big) \\
\qquad \mid \ldots
\end{array}
$$

with $opSearchPrice : \mathsf{T}_{sp}$ and $s : \mathsf{T}_{rbp}$.

The client C below looks for the price of book "`Title`" (we use freely an $\mathsf{if}\cdots\mathsf{then}$ construct that can be coded up in the same vein as Case):

$$
\begin{array}{l}
C \stackrel{\triangle}{=} (\nu r)\Big(\overline{opBookPrice}\langle\mathtt{bookPrice}[\mathtt{title}(\text{“}\mathtt{Title}\text{”}),\mathtt{rep}(r)]\rangle \\
\qquad \mid r.(\mathtt{bookPriceReply}[\mathtt{title}(\text{“}\mathtt{Title}\text{”}),\mathtt{store}(?x:\mathsf{string}),\mathtt{price}(?y:\mathsf{real})]) \\
\qquad\qquad \mathsf{if}\ (y < maxPrice)\ \mathsf{then}\ \overline{opBookOrder}(\cdots)\ \Big)
\end{array}
$$

with $r : \mathsf{T}_{rsp}$. Assuming there is a $j \in I$ such that "`Title`" $= t_j$, we have the following evolution of the system

$$S \mid B \mid C \rightarrow^* S \mid B \mid \big(\mathsf{if}\ (p_j < maxPrice)\ \mathsf{then}\ \overline{opBookOrder}(\cdots)\ \big).$$

This example can be generalised to more complex scenarios. For instance, we can define a Search Web that contacts more than one bookstore and offers its client all possible choices, both in terms of availability and price.