PiDuce – a project for experimenting Web services technologies

PiDuce – a project for experimenting Web services technologies [⋆]

Samuele Carpineti [a], Cosimo Laneve [a,∗], Luca Padovani [b]

[a] *University of Bologna, Department of Computer Science, Mura Anteo Zamboni, 7, 40127 Bologna, Italy*

[b] *University of Urbino, Information Science and Technology Institute, Piazza della Repubblica, 13, 61029 Urbino, Italy*

Abstract

The PiDuce project comprises a programming language and a distributed runtime environment devised for experimenting Web services technologies by relying on solid theories about process calculi and formal languages for XML documents and schemas.

The language features values and datatypes that extend XML documents and schemas with channels, an expressive type system with subtyping, a pattern matching mechanism for deconstructing XML values, and control constructs that are based on Milner's asynchronous pi calculus. PiDuce programs are compiled into typesafe object code. The runtime environment supports the execution of PiDuce object code over networks by relying on state-of-the-art technologies, such as XML schema and WSDL, thus enabling interoperability with existing Web services.

We thoroughly describe the PiDuce project: the programming language and its semantics, the architecture of the distributed runtime and its implementation. A running prototype is available at `http://www.cs.unibo.it/PiDuce/`.

Key words: pi calculus, XML schema, type system, subschema relation, WSDL, Web services.

[⋆] Aspects of this investigation were supported in part by a Microsoft initiative in concurrent computing and Web services.

[∗] Corresponding author.

Email addresses: `carpinet@cs.unibo.it` (Samuele Carpineti), `laneve@cs.unibo.it` (Cosimo Laneve), `padovani@sti.uniurb.it` (Luca Padovani).

1 Introduction

Web services are part of a recent emerging paradigm where computational elements are autonomous, platform-independent and can be described, published, discovered, and orchestrated for developing networks of collaborating applications distributed within and across organizations. Various technologies and languages have been proposed for designing Web services by the major Information Technology vendors. We recall `XLANG` [33], `BizTalk` [24], and `WS-BPEL` [5]. All these languages are informally specified and miss a mathematical model. In fact, they sometime retain vague descriptions of activities (e.g. the execution of compensation handlers in transactional activities), they lack verification tools, and they give poor guarantees about possible implementations.

Process calculi, such as pi calculus [29] and join calculus [16], are possible candidate models for Web services languages. Let us illustrate the point by an example. Consider a book-selling service that accepts requests containing the client identifier and the ordered book. When a request arrives, the book-selling service invokes the services of the Credit and Deposit Departments for verifying the client identity and the book availability. In case of success the request is confirmed, otherwise a failure message is returned. Figure 1 shows a (simplified) description of the book-selling service in `WS-BPEL` (similar definitions may be given in other Web services process languages such as `BizTalk`, `XLANG`, etc.). The format used for this description is `XML` [28], a widely-used standard for exchanging documents.

The reader familiar with process calculi will recognize the operations of sequence, input (`receive`), parallel composition (`flow`), output (`invoke`), as well as operations that are typical of sequential languages (`switch`). In particular, ignoring the `XML` details, the book selling service may be rewritten into a pi calculus-like language as follows:

```
OrderPT_in?(BookRequest, ClientId).
 ( CreditDeptPT_out!(ClientId) | DepositDeptPT_out!(BookRequest)
   | CreditDeptPT_in?(CreditResponse).
     DepositDeptPT_in?(BookResponse).
       match CreditResponse, BookResponse with
         true, true --> OrderPT_out!("OK")
         _, _ --> OrderPT_out!("NO") )
```

This process and the description of Figure 1 are actually inadequate because they oversimplify the structure of `XML` documents and the machinery for their parsing. For example lines 3 and 4 in Figure 1 are technically incorrect because the `receive` element has two attributes with the same name. These lines should be expanded into:

```
<sequence>
  <!--Receive the initial request from client -->
  <receive partnerLink="client" portType="com:OrderPT"
           operation="BookSelection"
           variable="BookRequest" variable="ClientID" />
  <!--Make concurrent invocations to Credit \& Deposit Dep-->
  <flow>
    <!--Invoke Deposit Department -->
    <invoke partnerLink="DepositDept" portType="ins:DepositDeptPT"
            operation="VerifyBookSelection"
            inputVariable="BookRequest"
            outputVariable="BookResponse" />
    <!--Invoke Credit Department -->
    <invoke partnerLink="CreditDept" portType="ins:CreditDeptPT"
            operation="VerifyCredit" inputVariable="ClientID"
            outputVariable="CreditResponse" /> </flow>
  <!--verify the responses -->
  <switch>
    <case condition="getVariableData(BookResponse) == true
            && getVariableData(CreditResponse) == true)">
      <!--Reply OK to the client -->
      <reply partnerLink="client" portType="com:OrderPT"
             operation="SelectBook" value="OK" /> </case>
    <otherwise> <!--Reply NO to the Client -->
      <reply partnerLink="client" portType="com:OrderPT"
             operation="SelectBook" value="NO" /> </otherwise>
  </switch>
</sequence>
```

Fig. 1. A (simplified) description of the book selling service in WS-BPEL.

```
  <receive partnerLink="client" portType="com:OrderPT"
    operation="BookSelection" variable="BookSelectionInput"/>
  <copy> <from variable="BookSelectionInput"
    query="/BookSelectionInput/BookRequest"/>
    <to variable="BookRequest"/> </copy>
  <copy> <from variable="BookSelectionInput"
    query="/BookSelectionInput/ClientId"/>
    <to variable="ClientId"/> </copy>
```

The above code parses the tree structure of the received XML document, it extracts two fragments located at /BookSelectionInput/BookRequest and /BookSelectionInput/ClientId, and stores such fragments in two variables called BookRequest and ClientId, respectively. Therefore, a process calculus for faithfully describing Web services processes cannot overlook XML values, XML schemas, and patterns.

```
<wsdl:definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    targetNamespace="http://buy_a_book.com/bookseller/">
  <wsdl:types> <!--the schema definitions -->
    <s:element name="BookSelectionRQ">
      <s:sequence>
        <s:element name= "BookRequest" type="s:string" />
        <s:element name= "ClientId" type="s:string" />
      </s:sequence>
    </s:element>
    <s:element name="BookSelectionRS" type="s:string" />
  </wsdl:types>
  <wsdl:operation name="BookSelection">
    <!--the operations of the service -->
    <wsdl:input message="tns:BookSelectionRQ" />
    <wsdl:output message="tns:BookSelectionRS" />
  </wsdl:operation>
  ...
  <wsdl:binding> <!--the locations of the operations -->
    <wsdl:operation name="BookSelection">
      <soap:operation
        soapAction="http://buy_a_book.com/bookseller/BookSelection"/>
    </wsdl:operation>
  <wsdl:binding>
</wsdl:definitions>
```

Fig. 2. A (simplified) `WSDL` of the book selling service.

The `PiDuce` project (`www.cs.unibo.it/PiDuce`) aims at developing a calculus of processes that may construct and deconstruct `XML` documents. The calculus is intended to serve as an intermediate language that is powerful enough to encode the common operations of Web services languages, to assess their expressive power and to develop tools for their analysis. The project also aims at designing a formally specified distributed machine running applications that may be exported to the Web. Overall, `PiDuce` is not *the* platform for Web service technologies, but rather it is a formal framework for experimenting proposals, studying their theory, and implementing the relevant features.

The design of the `PiDuce` language, as well as the prototype implementation, presents some major technical difficulties. The first difficulty regards the treatment of service references. These references are already present in `WSDL` documents [25], which are the standard files used for publishing and discovering services. Table 2 contains a (simplified) `WSDL` of the book-selling service.

Recently, a new version of `WSDL` (`WSDL 2.0` [26,27] that, at the time of this writing, is in a Candidate Recommendation status) uses service references in the `wsdl:types` part of the document. This corresponds to extending types with service constructors that collect references with a given interface. Therefore, a service invocation may contain a reference that the called service might eventually compare with some local schema before using it. Such a comparison amounts to downloading the schema of the reference and computing a sublanguage relation between schemas. This, in general, requires exponential time in the size of the tree automata of the pattern [14] and may significantly degrade the run-time efficiency of possible implementations. We alleviate this problem by designing a number of restrictions on schemas that make subschema verification polynomial.

A second difficulty regards the implementation. In `BizTalk`, a service using reliable messaging, such as `MSMQ` and `MQSeries`, may receive values from local and remote channels – `MessageQueue`s in `BizTalk` [24]. For example, the `C#` fragment below may be obtained in `BizTalk` by drawing a receive activity on a `MSMQ` adapter:

```
1   string queueAddress = @"ServerName\QUEUE";
2   MessageQueue q = new MessageQueue(queueAddress, false, false,
3                                     QueueAccessMode.Receive);
4   Message m = q.Receive();
```

Line 1 defines the address of a queue as consisting of a machine name – the `ServerName` – and the name of the queue – `QUEUE`. Line 2 defines a reference `q` to the address of line `1` specifying the operations that will be performed. In particular, the last argument of `MessageQueue` constrains the use of `q` for receiving messages. The receive operation is performed in line 4. This feature is known as *input capability* in process calculi, which is the mechanism where a received reference is used as the subject of a subsequent input. Implementing input capability in a distributed setting is a hard task because it allows the dynamic creation of large input processes in the wrong place, thus requiring comparatively large code migrations in order to avoid consensus problems. `PiDuce` admits input capability and implements it by means of linear forwarders [18]. The solution consists of allowing just a limited atom of input capability – the *linear forwarder*. A linear forwarder is a process that simply turns one message on a Uniform Resource Identifiers (URI) into a message on another. For instance

```
uri1?(m) uri2!(m)
```

is a linear forwarder. To illustrate how linear forwarders enable input capability, consider the process `uri?(u) u?(v) P`, where `P` is a continuation process. This process is encoded as

```
uri?(u) new w in ( spawn { u?(m) w!(m) } w(v) Q )
```

where the input `u?(v)` has been turned into an input `w?(v)` on a service reference `w` *created at the same location as* `uri`, and where the forwarder allows one output on `u` to interact with `w` instead. The key observation is that the linear forwarder is easy to implement: it is just a small message containing two `URL`s and directed to the location of `u`. This paper may be also seen as a formal (alternative) implementation of input capability in `BizTalk`, whose implementation details have not been published.

A third difficulty regards *interoperability.* This feature is a primary focus of the `PiDuce` project for two reasons. First of all, it allows us to carry on actual experiments, by letting `PiDuce` define and interact with real Web services. Second, it connects a formally specified system with the current technologies, thus providing such technologies with a foundational basis and possibly spotting their weaknesses, ambiguities, and lines of extension. As an example, a `PiDuce` program should be able to interoperate with the service in Figure 1. In order to do so, it can only rely on the public description of the service – its `WSDL` in Figure 2. A `PiDuce` client for the book-selling service written in `PiDuce` must import the corresponding `WSDL` to ensure that the channels used for communication with the service are typed in accordance with what declared in the client. Symmetrically, a Web service implemented in `PiDuce` must be able to export its operations by means of a `WSDL` resource. Overall, such import/export procedures entail a mapping between the `PiDuce` schemas and, say, `XML` schema, which is the language typically used in `WSDL` resources to describe the valid documents exchanged with a Web service. This mapping is problematic because the two systems do not have the same expressive power. For example, in `PiDuce` service references are first-class values; therefore `PiDuce` schemas include channel types, which are not supported in `XML` schema. More generally, `PiDuce` schema retain features that are fundamental in order to guarantee the typability of processes (*cf.* nondeterministic unions of schemas) but which are not found in `XML` schema. It turns out that the effort for making `PiDuce` interoperable is considerable because what is theoretically clean and well-founded is not necessarily what the technologies provide or what is practically widespread. In this respect the contribution of `PiDuce` is original in that all the exiting languages and distributed machines either lack a formal foundation or are mostly isolated, without a strong connection with actual technologies.

The paper is structured as follows. Section 2 is an introduction to the `PiDuce` language constructs through examples. Section 3 defines the syntax of the language. Section 4 defines the subschema relation and the static semantics of the `PiDuce` process language. Section 5 describes the pattern matching and the operational semantics of local operations. Section 6 defines the `PiDuce` distributed machine and the static and dynamic semantics of operations that deal with remote locations. In Section 7 we close the gap between `PiDuce` and Web service technologies by adding the notions of synchronous communication and service operations. Section 8 briefly describes the architecture of the

PiDuce runtime and shows how PiDuce is interoperable. Section 9 discusses related works and Section 10 concludes with an example of PiDuce program that interoperates with real-world Web services. Appendixes A and B contain proofs of the results stated in the paper. Appendix C presents an algorithmic version of the subschema relation.

2 Getting started

The basic elements of PiDuce are introduced through a few examples. The formal presentation is deferred to the next section.

PiDuce values represent XML documents. For example, the XML document

```
<msg>hello</msg><doc/>
```

is written in PiDuce as `msg["hello"],doc[ ]`.

PiDuce schemas are used to type values and approximate XML-Schemas. For example, the XML-Schema

```
<xsd:element name="a" type="xsd:integer"/>
```

describing a-labelled integers is written as a[int]. The XML-Schema

```
<xsd:sequence>
  <xsd:element name="a" type="xsd:integer"/>
  <xsd:choice>
    <xsd:element name="b" type="xsd:string"/>
    <xsd:element name="c"/>
  </xsd:choice>
</xsd:sequence>
```

is written as a[int],(b[string] + c[ ]). Schemas with a repeated structure are written in PiDuce by means of the star operator. For example, the XML-Schema

```
<xsd:sequence minOccurs="0" maxOccurs="unbound">
  <xsd:element name="a" type="xsd:string"/>
</xsd:sequece>
```

is written as a[string]*. A detailed discussion of the relationship between XML and PiDuce schemas is undertaken in Section 8.

PiDuce processes describe Web services. For example, a printer service that collects color and black-white printing requests is defined by

```
print?*(x : Pdf + JPeg)
  match x with {
    y : Pdf => printbw!(y)
  | z : JPeg => printc!(z) }
```

The `print` service accepts a value `x` of schema `Pdf + JPeg` (where "+" denotes schema union), it checks whether the received value `x` belongs to either `Pdf` or `JPeg`; in the first case it forwards the value `x` to the black-white printer, in the second case it forwards the value `x` to the color printer. The basic mechanism for interactions is message passing. For example `print!(document)` invokes the service `print` with the value `document`. Service invocation is non-blocking and asynchronous: the sender does not wait that the receiver really consumes the message. The star after the question mark in the `print` service above indicates that the service is permanent: the process is capable of handling an unlimited number of requests.

The parallel execution of several activities is defined by the `spawn` construct. For example

```
spawn { print!(document1) } print!(document2)
```

invokes `print` twice. Because of asynchrony, there is no guarantee as to which invocation will be served first. More elaborated forms of control and communication, such as sequentiality and *rendez-vous*, can be encoded using explicit continuation-passing style.

In `PiDuce` it is possible to `select` one input out of many. This operation, which is similar to the homonymous system call in socket programming, to the "pick activity" in `WS-BPEL`, and to the *input-guarded choice* in the pi calculus, permits the definition of alternative activities. For example, consider a printer service that after the printer request waits for the black-white or color request and prints the document accordingly:

```
print?*(x : Pdf + JPeg)
 select { b&w?( () ) printbw!(x)
          color?( () ) printc!(x) }
```

(note the missing `*` after `b&w?` and `color?`). In general, the `select` operation groups several input operations to be executed in mutual exclusion.

Service names may be created dynamically. In their simplest form, services have exactly one operation whose name coincides with that of the service. Services are declared as follows:

$$\texttt{new}\,\texttt{print} : \langle \texttt{Pdf} + \texttt{JPeg}\rangle^{\texttt{O}}\ \texttt{in}\ P$$

The `new` operation creates a new channel at the `URL` address of the runtime environment executing this code (each service `URL` is made unique by appending an appropriate suffix) and publishes a `WSDL` document describing `print` as an asynchronous – *one-way*, in `WSDL` jargon – service (the *capability* is "`O`") accepting documents of schema `Pdf + JPeg`. The scope of the declaration is restricted to P. In `PiDuce` channels are first-class citizens: they are values that can be sent over and received from other channels and they can be examined by pattern matching. With this standpoint, the operation `new` is intended to declare the schema of a channel literal. Multi-operation services may be also defined. For example the code

$$\texttt{new}\ \mathit{cell} : \{ \mathit{get} : \langle\langle \texttt{int} \rangle^{\texttt{O}}\rangle^{\texttt{O}} \,;\, \mathit{set} : \langle \texttt{int} \rangle^{\texttt{O}} \}\ \texttt{in}\ P$$

defines a service *cell* with the operations *get* and *set* (see Section 7). These operations may be addressed in P by *cell*`#`*get* and *cell*`#`*set*, respectively.

3 The `PiDuce` language

The syntax of `PiDuce` includes the categories *labels*, *expressions*, *schemas*, *patterns*, and *processes* that are defined in Table 1. The following countably infinite sets are used: the set of *tags*, ranged over by $a, b, \ldots$; the set of *variables*, ranged over by $x, y, z, \ldots$; the set of *schema names*, ranged over by $\texttt{U}, \texttt{V}, \ldots$; the set of *pattern names*, ranged over by $\texttt{Y}, \texttt{J}, \ldots$. Among variables we distinguish *channels*, i.e. names to be used as `URL`s, ranged over by u, v, $\ldots$.

A `PiDuce` program is

$$\texttt{U}_1 = S_1 \,;;\, \cdots \,;;\, \texttt{U}_n = S_n \,;;\, \texttt{Y}_1 = F_1 \,;;\, \cdots \,;;\, \texttt{Y}_m = F_m \,;;\, P$$

that is a sequence of schema and pattern name definitions and a process. For simplicity we assume that the names $\texttt{U}_1, \ldots, \texttt{U}_n, \texttt{Y}_1, \ldots, \texttt{Y}_m$ are pairwise different. Sequences of schema name and pattern name definitions are represented by maps with finite domain $\mathcal{E}$ and $\mathcal{F}$ that take a name and return the associated schema or pattern, respectively.

The sets $\texttt{fv}(\cdot)$ of *free variables* and $\texttt{bv}(\cdot)$ of *bound variables* are defined for expressions, patterns, and processes as follows:

$\texttt{fv}(E)$ is the set of variables occurring in E; $\texttt{bv}(E)$ is empty;
$\texttt{fv}(F)$ is the set of variables occurring in F and, recursively, in the definition of every pattern name occurring in F; $\texttt{bv}(F)$ is empty;
$\texttt{fv}(P)$ is the set of variables occurring in P that are not *bound*. An occurrence of x in P is *bound* in a branch $u\texttt{?}(F)\ P$ of a select or in the replicated input $u\texttt{?*}(F)\ P$ if $x \in \texttt{fv}(F)$; an occurrence of u in P is *bound* in $\texttt{new}\ u : \langle S \rangle^{\kappa}\ \texttt{in}\ P$. $\texttt{bv}(P)$ collects the bound variables in P.

Table 1
`PiDuce` syntax (`B` includes `int`, `string`, integer and string constants).

$L ::=$	**label**	$F ::=$	**pattern**
a	(tag)	`()`	(void pattern)
`~`	(wildcard label)	`B`	(basic schema)
$L + L$	(union)	$\langle S\rangle^{\kappa}$	(channel pattern)
$L \setminus L$	(difference)	S^*	(star pattern)
		$x : F$	(variable binder)
$E ::=$	**expression**	$L[F]$	(labelled pattern)
`()`	(void)	F`,`F	(sequence pattern)
`n`	(integer constant)	$F + F$	(union pattern)
`s`	(string constant)	`Y`	(pattern name)
x	(variable)		
$a[E]$	(labelled expression)	$P ::=$	**process**
E`,`E	(sequence)	$\mathbf{0}$	(nil)
		u`!`(E)	(output)
$S ::=$	**schema**	`select` $\{u_i$`?`$(F_i)\ P_i\ ^{i\in 1..n}\}$	
`()`	(void schema)		(select)
`B`	(basic schema)	`new` $u : \langle S\rangle^{\kappa}$ `in` P	
$\langle S\rangle^{\kappa}$	(channel schema)		(new)
$L[S]$	(labelled schema)	`match` E `with` $\{F_i \Rightarrow P_i\ ^{i\in 1..n}\}$	
S`,`S	(sequence schema)		(match)
$S + S$	(union schema)	`spawn` $\{P\}\ P$	(spawn)
S^*	(star)	u`?*`$(F)\ P$	(replication)
`U`	(schema name)		

The definitions of alpha-conversion and substitution for bound variables are standard. In the whole paper, we identify terms that are equal up-to alpha-conversion. In the following, the channel u in u`!`(E), u`?`(F) and u`?*`(F) is called *subject*.

Labels. Labels specify collections of tags. Let $\mathcal{L}$ be the set of all tags; the semantics of labels is defined by the $\widehat{\cdot}$ function:

$$\widehat{a} = \{a\} \qquad \widehat{\sim} = \mathcal{L} \qquad \widehat{L + L'} = \widehat{L} \cup \widehat{L'} \qquad \widehat{L \setminus L'} = \widehat{L} \setminus \widehat{L'}$$

We write $a \in L$ for $a \in \widehat{L}$. Label intersection is a derived operator: $L \cap L' \stackrel{\text{def}}{=} \sim \setminus ((\sim \setminus L) + (\sim \setminus L'))$.

Expressions. Expressions are the empty sequence `()`, integer and string constants, variables, labelled expressions, or sequences of expressions. The `PiDuce` prototype also includes primitive operations over basic schemas. The formal treatment of such operations is omitted as it is standard and not interesting. In the following, whenever possible, `()` is omitted: expressions such as $a[\texttt{()}]$ and $a[E]\texttt{,()}$ are shortened into $a[\,]$ and $a[E]$, respectively.

Channels are references to services. They represent `URL` addresses of the corresponding `WSDL` interfaces, such as `http://www.cs.unibo.it/PiDuce.wsdl`. Section 8 discusses how `WSDL` interfaces are related to `PiDuce` services.

A relevant subset of expressions is that of *values*, which are are the normalized expressions exchanged during communications. As in pi calculus, `PiDuce` values may contain channels, which are variables. For example $a[x]$ is a value inasmuch as x is a channel. Values with variables in a set Z, called Z*-values*, are possibly empty sequences of constants, variables in Z, and labelled Z-values. In particular, expressions such as $\texttt{()},a[\texttt{1}]$ or $a[\texttt{1}]\texttt{,()},b[\texttt{true}]$ or $a[\texttt{1}]\texttt{,()}$ or u are not $\emptyset$-values. Values are ranged over by V, W, $\dots$, and, in the following, the set Z is omitted when it is clear from the context.

The evaluation function $\Downarrow_Z$, where Z is a set of variables, turns expressions into values and is defined by the following rules:

$$\texttt{()} \Downarrow_Z \texttt{()} \qquad \texttt{n} \Downarrow_Z \texttt{n} \qquad \texttt{s} \Downarrow_Z \texttt{s}$$

$$\frac{u \in Z}{u \Downarrow_Z u} \qquad \frac{E \Downarrow_Z V}{a[E] \Downarrow_Z a[V]} \qquad \frac{E \Downarrow_Z V \quad E' \Downarrow_Z V'}{E\texttt{,}E' \Downarrow_Z V@V'}$$

where the concatenation @ of two values is defined as follows:

$$\begin{array}{rcll}
\texttt{()}@V &=& V@\texttt{()} = V & \\
V@V' &=& V\texttt{,}V' & \text{if } (V = \texttt{n} \text{ or } V = \texttt{s} \text{ or } V = x \text{ or } V = a[V'']) \\
&&& \text{and } V' \neq \texttt{()} \\
(V\texttt{,}V')@V'' &=& V@(V'@V'') &
\end{array}$$

Schemas. Schemas describe collections of structurally similar values. The syntax of schemas includes the category of basic types `B` that, in this paper, are integers (`int`), strings (`string`) and integer and string constants `n` and `s`, respectively. The basic types `n` and `s` represent the sets $\{\texttt{n}\}$ and $\{\texttt{s}\}$, respectively. The schema `()` describes the empty value. The schema $\langle S\rangle^\kappa$ describes channels that carry messages of schema S and that may be used with *capability* $\kappa \in \{\texttt{I}, \texttt{O}, \texttt{IO}\}$. The capabilities $\texttt{I}, \texttt{O}, \texttt{IO}$ mean that the channel can be used for performing inputs, outputs, and both inputs and outputs, respectively. For example $\langle\texttt{int}\rangle^{\texttt{O}}$ describes the set of channels on which it is possible to send integer values. The schema $L[S]$ describes labelled values whose tag is in L and containing a value of schema S. The schema $S\texttt{,}S'$ describes sequences having a prefix of schema S and the remaining suffix of schema S'. In what follows $L[\texttt{()}]$, $\texttt{()},S$ and $S\texttt{,()}$ are shortened into $L[\,]$, S, and S, respectively.

The schema $S + S'$ describes the set of values whose schema is either S or S'. The schema S^* describes the set of values that are described by every finite (possibly empty) sequence $S\texttt{,}\dots\texttt{,}S$. Schemas include schema names that are bound by *finite* maps $\mathcal{E}$ from schema names to schemas such that, for every $\mathtt{U} \in \mathtt{dom}(\mathcal{E})$, the schema names in $\mathcal{E}(\mathtt{U})$ belong to $\mathtt{dom}(\mathcal{E})$. Maps $\mathcal{E}$ are *well-formed* according to the definition below. Let $\mathtt{tls}(S)$ be the least function such that:

$$\mathtt{tls}(S) = \begin{cases} \{\mathtt{U}\} \cup \mathtt{tls}(\mathcal{E}(\mathtt{U})) & \text{if } S = \mathtt{U} \\ \mathtt{tls}(T) & \text{if } S = T^* \\ \mathtt{tls}(T) \cup \mathtt{tls}(T') & \text{if } S = T + T' \text{ or } S = T\texttt{,}T' \\ \emptyset & \text{otherwise} \end{cases}$$

Then $\mathcal{E}$ is well-formed if, for every $\mathtt{U} \in \mathtt{dom}(\mathcal{E})$, $\mathtt{U} \notin \mathtt{tls}(\mathcal{E}(\mathtt{U}))$. The well-formedness and the finiteness of the domain of $\mathcal{E}$ guarantee that `PiDuce` schemas only define *regular tree languages* (such languages retain a decidable sublanguage relation, which is a fundamental operation in `PiDuce` type checking and pattern-matching) [14].

The following definitions will be used in the rest of the paper:

$\mathtt{Empty} = \texttt{~}[\mathtt{Empty}]$ `;;`
$\mathtt{AnyChan} = \langle\mathtt{Empty}\rangle^{\mathtt{O}} + \langle\mathtt{Any}\rangle^{\mathtt{I}}$ `;;`
$\mathtt{Any} = (\mathtt{int} + \mathtt{string} + \mathtt{AnyChan} + \texttt{~}[\mathtt{Any}])^*$ `;;`

The name `Empty` describes the empty set of values; `AnyChan` describes any channel; `Any` describes any value. `Empty` and `Any` are respectively the least and the greatest schema according to the subschema relation of Section 4 (Proposition 2(9)).

Patterns. Patterns permit the deconstruction of values using matching. The patterns `()`, `B`, $\langle S\rangle^\kappa$, and S^* match values of the corresponding schemas. The pattern $x : F$ matches the same values matched by F and additionally it binds such values to the variable x. The pattern $L[F]$ matches values of the form $a[V]$, when $a \in L$ and F matches V. The pattern $F\texttt{,}F'$ matches values $V = V'@V''$ such that V' and V'' are matched by F and F', respectively. The pattern $F + F'$ matches values V that are matched by either F or F'. The pattern matching algorithm is *deterministic*:

(1) in a pattern $S^*\texttt{,}F$, the partition of V into $V'@V''$ is such that V' is the longest prefix of V that is matched by S^* (*longest match policy*);
(2) in a pattern $F+F'$ the match with the left-hand side pattern is attempted first; in case of failure, the match with the right-hand side pattern is attempted (*first match policy*).

Patterns include pattern names that are bound by finite maps $\mathcal{F}$ from pattern names to patterns such that, for every $\mathtt{Y} \in \mathtt{dom}(\mathcal{F})$, the pattern names in $\mathcal{F}(\mathtt{Y})$ belong to $\mathtt{dom}(\mathcal{F})$. Pattern definitions must obey the same well-formedness restrictions of schema definitions. In addition, `PiDuce` patterns are *linear*, namely the following three conditions hold:

(1) every pattern $x : F$ is such that $x \notin \mathtt{fv}(F)$;
(2) every pattern $F\mathtt{,}F'$ is such that $\mathtt{fv}(F) \cap \mathtt{fv}(F') = \emptyset$;
(3) every pattern $F + F'$ is such that $\mathtt{fv}(F) = \mathtt{fv}(F')$.

In the following we write $\mathtt{schof}(F)$ for the schema obtained by erasing all the variables in F.

Processes. Processes are the computing entities of `PiDuce`. $\mathbf{0}$ is the idle process; $u\mathtt{!}(E)$ evaluates E to a value and outputs it on the channel u. The process $\mathtt{select}\ \{u_i\mathtt{?}(F_i)\ P_i\ ^{i\in 1..n}\}$ inputs a value on the channel u_i, matches the value with F_i yielding a substitution σ and behaves as $P_i\sigma$. We always abbreviate $\mathtt{select}\ \{u\mathtt{?}(F)\ P\}$ to $u\mathtt{?}(F)\ P$. The process $\mathtt{new}\ u : \langle S\rangle^\kappa\ \mathtt{in}\ P$ defines a fresh channel u and binds it within the continuation P, where u may be used as subject of input and output operations, whereas the capability κ is exposed in the `WSDL` interface associated with the channel (see Section 8). The process $\mathtt{match}\ E\ \mathtt{with}\ \{F_i \Rightarrow P_i\ ^{i\in 1..n}\}$ tests whether the value to which E evaluates is matched by one of the patterns F_i's. The order of the branches is relevant, so that the first matching pattern determines the continuation (first match policy). If the match with F_k succeeds, the continuation $P_k\sigma$ is run, where σ is the substitution yielded by the pattern matching algorithm. The process $\mathtt{spawn}\ \{P\}\ Q$ spawns the execution of P on a separate thread and continues as Q. The replicated input $u\mathtt{?*}(F)\ P$ consumes a message on u, it spawns the continuation $P\sigma$, where σ is the substitution yielded by matching the message with the pattern F, and then it becomes available for other messages on u.

The syntax of Table 1 will be extended in Section 6 with operations regarding remote machines, such as the creation of channels at remote locations or the select on remote channel.

4 The subschema relation and the type system

A basic check in the `PiDuce` compiler and runtime is the language containment of schemas, called *subschema relation.* In [21] this notion is defined in terms of set-containment. In particular, let $[\![S]\!] \stackrel{\text{def}}{=} \{V \mid V \text{ is of schema } S\}$. Then S is a subschema of T if $[\![S]\!] \subseteq [\![T]\!]$. This approach is inadequate in `PiDuce` because of the presence of channels. Indeed, the values of $\langle S\rangle^{\texttt{O}}$ are sets of names that may be defined at runtime. To circumvent this problem we follow an approach proposed in [4] and already used in pi calculus [31].

Let $S \downarrow R$, read S *has handle* R, be the least relation such that:

$$\begin{array}{ll}
\texttt{()} \downarrow \texttt{()} & \\
\texttt{B} \downarrow \texttt{B,()} & \\
\langle S\rangle^{\kappa} \downarrow \langle S\rangle^{\kappa}\texttt{,()} & \\
L[S] \downarrow L[S]\texttt{,()} & \text{if } L \neq \emptyset \text{ and, for some } R,\ S \downarrow R \\
S\texttt{,}S' \downarrow R & \text{if } S \downarrow \texttt{()} \text{ and } S' \downarrow R \\
S\texttt{,}S' \downarrow R\texttt{,}S' & \text{if } S \downarrow R \text{ and } R \neq \texttt{()} \text{ and, for some } R',\ S' \downarrow R' \\
S + S' \downarrow R & \text{if } S \downarrow R \text{ or } S' \downarrow R \\
\texttt{U} \downarrow R & \text{if } E(\texttt{U}) \downarrow R \\
S^{*} \downarrow \texttt{()} & \\
S^{*} \downarrow R\texttt{,}S^{*} & \text{if } S \downarrow R \text{ and } R \neq \texttt{()}
\end{array}$$

The relation "$\downarrow$" singles out the branches of the syntax tree of a schema. For example $(a[\texttt{int}]\texttt{,string} + b[\texttt{string}]\texttt{,int}) \downarrow a[\texttt{int}]\texttt{,string}$. We observe that `Empty` has no handle. The schema $a[\texttt{int}]\texttt{,Empty}$ has no handle as well; the reason is that a sequence has a handle provided that every element of the sequence has a handle. We also remark that a channel $\langle S\rangle^{\kappa}$ always retains a handle. Let S be *not-empty* if and only if S has a handle; it is *empty* otherwise.

Definition 1 *Let $\leq$ be the least partial order on capabilities such that $\texttt{IO} \leq \texttt{I}$ and $\texttt{IO} \leq \texttt{O}$. Let $\sqsubseteq$ be the least partial order on basic schemas such that $\texttt{n} \sqsubseteq \texttt{int}$ and $\texttt{s} \sqsubseteq \texttt{string}$. A* subschema *$\mathcal{R}$ is a relation on schemas such that $S \mathrel{\mathcal{R}} T$ implies:*

(1) $S \downarrow \texttt{()}$ implies $T \downarrow \texttt{()}$;
(2) $S \downarrow \texttt{B,}S'$ implies $T \downarrow \texttt{B}'_i\texttt{,}T'_i$, for $1 \leq i \leq n$, with $\texttt{B} \sqsubseteq \texttt{B}'_i$ and $S' \mathrel{\mathcal{R}} \sum_{1\leq i\leq n} T'_i$;
(3) $S \downarrow \langle S'\rangle^{\kappa}\texttt{,}S''$ implies $T \downarrow \langle T_i\rangle^{\kappa_i}\texttt{,}T'_i$, for $1 \leq i \leq n$, with $\kappa \leq \kappa_i$, $S'' \mathrel{\mathcal{R}} \sum_{1\leq i\leq n} T'_i$, and, for every $1 \leq i \leq n$, one of the following conditions holds:
   (a) $\kappa_i = \texttt{O}$ and $T'_i \mathrel{\mathcal{R}} S'$, or
   (b) $\kappa_i = \texttt{I}$ and $S' \mathrel{\mathcal{R}} T'_i$, or
   (c) $\kappa_i = \texttt{IO}$ and $S' \mathrel{\mathcal{R}} T'_i$ and $T'_i \mathrel{\mathcal{R}} S'$;
(4) $S \downarrow L[S']\texttt{,}S''$ implies that one of the following conditions holds:
   (a) $T \downarrow L'[T']\texttt{,}T''$ with $\widehat{L} \cap \widehat{L'} \neq \emptyset$, $\widehat{L} \not\subseteq \widehat{L'}$, $(L \setminus L')[S']\texttt{,}S'' \mathrel{\mathcal{R}} T$, and $(L \cap L')[S']\texttt{,}S'' \mathrel{\mathcal{R}} T$, or
   (b) $T \downarrow L_i[T_i]\texttt{,}T'_i$, for $1 \leq i \leq n$, with $\widehat{L} \subseteq \bigcap_{i\in\{1,\ldots,n\}} \widehat{L_i}$ and, for every $J \subseteq \{1,\ldots,n\}$, either $S' \mathrel{\mathcal{R}} \sum_{i\in J} T_i$ or $S'' \mathrel{\mathcal{R}} \sum_{i\in\{1,\ldots,n\}\setminus J} T'_i$.

Let `<:` *be the largest* subschema relation.

The definition of subschema is commented upon below. Item 1 constraints greater schemas to manifest a void handle if the smaller one retains such a handle. Item 2 deals with basic schemas $\mathtt{B},S'$: a set of handles $\mathtt{B}_i,T'_i$ of the greater schema is selected such that $\mathtt{B}$ is smaller than $\mathtt{B}_i$ and S' is smaller than the union of the T'_i's. Item 3 is similar to item 2, except for the heads of handles, which are channel schemas. In order to check the subschema relation between $\langle S\rangle^{\kappa}$ and $\langle T\rangle^{\kappa'}$, the capability κ must be smaller than κ'. Additionally, in case $\kappa' = \mathtt{O}$ the subschema is inverted on the arguments (contravariance); in case $\kappa' = \mathtt{I}$ the subschema is the same as for the arguments (covariance), in case $\kappa' = \mathtt{IO}$ the relation reduces to check the equivalence of the arguments (invariance). For example $\langle \mathtt{int} + \mathtt{string}\rangle^{\mathtt{O}}$ `<:` $\langle \mathtt{int}\rangle^{\mathtt{O}}$ because every channel that may carry either integers or strings can carry integers only. On the contrary, $\langle \mathtt{int}\rangle^{\mathtt{I}}$ `<:` $\langle \mathtt{int} + \mathtt{string}\rangle^{\mathtt{I}}$ because every channel that may serve invocations carrying either integers or strings can serve invocations with integers only.

Item 4 is the most complex one. It deals with handles $L[S'],S''$. We illustrate the point by means of an example. The case (a) accounts for subschema relations between $S = (a+b)[\mathtt{int}],\mathtt{int}$ and $T = a[\mathtt{int}],\mathtt{int} + b[\mathtt{int}],\mathtt{int}$. Since $T \downarrow a[\mathtt{int}],\mathtt{int}$, according to 4.a, the relation may be reduced to the check whether $((a+b)\backslash a)[\mathtt{int}],\mathtt{int}$ and $((a+b)\cap a)[\mathtt{int}],\mathtt{int}$ are subschema of T. The case (b) accounts for subschema relations between $S = a[\mathtt{int} + \mathtt{string}],\mathtt{int}$ and $T = a[\mathtt{int}],\mathtt{int} + a[\mathtt{string}],\mathtt{int}$. We explain this case by using an argument similar to that used in [22]. Let us admit a schema intersection operator $\cap$ such that $S \cap T$ describes the values that belong to both S and T. Then $L[S],T$ may be rewritten as $L[S],\mathtt{Any} \cap \mathtt{\sim}[\mathtt{Any}],T$ using the fact that $\mathtt{Any}$ is the greatest schema (see Proposition 2.6). Then:

$$\begin{aligned} L_1[S_1],T_1 + L_2[S_2],T_2 &= (L_1[S_1],\mathtt{Any} \cap \mathtt{\sim}[\mathtt{Any}],T_1) + (L_2[S_2],\mathtt{Any} \cap \mathtt{\sim}[\mathtt{Any}],T_2) \\ &= (L_1[S_1],\mathtt{Any} + L_2[S_2],\mathtt{Any}) \cap (\mathtt{\sim}[\mathtt{Any}],T_1 + \mathtt{\sim}[\mathtt{Any}],T_2) \\ &\quad \cap (L_1[S_1],\mathtt{Any} + \mathtt{\sim}[\mathtt{Any}],T_2) \cap (\mathtt{\sim}[\mathtt{Any}],T_1 + L_2[S_2],\mathtt{Any}) \end{aligned}$$

where the last equality follows by distributivity of $\cap$ with respect to union. Therefore, if one intends to derive that $L[S],T$ is a subschema of $L_1[S_1],T_1 + L_2[S_2],T_2$ when $\widehat{L} \subseteq \widehat{L_1} \cap \widehat{L_2}$, it is possible to reduce to:

$$\text{for every } J \subseteq \{1,2\} \text{ either} \quad S \;\mathcal{R} \sum_{j\in J} S_j \quad \text{or} \quad T \;\mathcal{R} \sum_{j\in\{1,2\}\backslash J} T_j$$

This is exactly item 4.b when $I = \{1,2\}$. A particular case is when $I = \{1\}$. For example verifying that $a[S],T$ is a subschema of $(a+b)[S'],T'$. In this case the subsets of I are $\emptyset$ and $\{1\}$ and one is reduced to prove (we let $\sum_{j\in\emptyset} S_j = \mathtt{Empty}$):

$$\Big(S \;\mathcal{R}\; \mathtt{Empty} \quad \text{or} \quad T \;\mathcal{R}\; T'\Big) \quad \text{and} \quad \Big(S \;\mathcal{R}\; S' \quad \text{or} \quad T \;\mathcal{R}\; \mathtt{Empty}\Big)$$

That is, when S and T are not subschema of `Empty`, we are reduced to $S \ \mathcal{R} \ S'$ and $T \ \mathcal{R} \ T'$.

The schemas `AnyChan` and `Any` own relevant properties. `AnyChan` collects all the channel schemas, no matter what they can carry; `Any` collects all the values, namely possibly empty sequences of possibly labelled values, including channels. We observe that $\langle\texttt{Empty}\rangle^{\texttt{O}}$ and $\langle\texttt{Any}\rangle^{\texttt{O}}$ are very different. $\langle\texttt{Empty}\rangle^{\texttt{O}}$ collects every channel with either capability "`O`" or "`IO`", $\langle\texttt{Any}\rangle^{\texttt{O}}$ refers only to channels where arbitrary data can be sent. For instance $\langle a[\,]\rangle^{\texttt{O}}$ is a subschema of $\langle\texttt{Empty}\rangle^{\texttt{O}}$ but not of $\langle\texttt{Any}\rangle^{\texttt{O}}$. The channel schemas $\langle\texttt{Any}\rangle^{\texttt{I}}$ and $\langle\texttt{Empty}\rangle^{\texttt{I}}$ are different as well. $\langle\texttt{Any}\rangle^{\texttt{I}}$ refers to references that may receive arbitrary data; $\langle\texttt{Empty}\rangle^{\texttt{I}}$ refers to a reference that cannot receive anything.

A few properties of `<:` are in order. The proofs can be found in Appendix A.

Proposition 2 *(1)* `<:` *is reflexive and transitive;*
(2) If S is empty, then S `<:` `Empty`*;*
(3) (Contravariance of $\langle\cdot\rangle^{\texttt{O}}$) S `<:` T if and only if $\langle T\rangle^{\texttt{O}}$ `<:` $\langle S\rangle^{\texttt{O}}$;
(4) (Covariance of $\langle\cdot\rangle^{\texttt{I}}$) S `<:` T if and only if $\langle S\rangle^{\texttt{I}}$ `<:` $\langle T\rangle^{\texttt{I}}$;
(5) (Invariance of $\langle\cdot\rangle^{\texttt{IO}}$) S `<:` T and T `<:` S if and only if $\langle S\rangle^{\texttt{IO}}$ `<:` $\langle T\rangle^{\texttt{IO}}$;
(6) If S `<:` T, then S`,()` `<:` T; if `(),`S `<:` T, then S `<:` T;
(7) If S `<:` T and S' `<:` T', then S`,`S' `<:` T`,`T';
(8) If $(S + S')$`,`S'' `<:` T, then S`,`S'' `<:` T and S'`,`S'' `<:` T;
(9) For every S, `Empty` `<:` *S* `<:` `Any` *and $\langle S\rangle^{\kappa}$* `<:` `AnyChan` *and $\langle\texttt{Any}\rangle^{\texttt{IO}}$* `<:` *$\langle S\rangle^{\texttt{O}}$ and $\langle\texttt{Empty}\rangle^{\texttt{IO}}$* `<:` *$\langle S\rangle^{\texttt{I}}$.*

Remark 3 *The algorithm for computing the subschema relation in* `PiDuce` *is similar to the one developed for* `XDuce` *[22]. It is computationally expensive: the cost of the algorithm for subschema is exponential in the size of the schemas. Paying this cost at compile time may be acceptable. However, in* `PiDuce` *the subschema relation is invoked at runtime by pattern matching (see Section 5). Paying an exponential cost at runtime becomes fateful because the performance degradation might be unacceptable. For instance an attacker might block a service by invoking it with channels of complex schemas, thus yielding a* denial of service. *A set of constraints on schemas that reduce the cost of the subschema algorithm has been designed in [11]. The* `PiDuce` *compiler warns the user when programs use schemas that do not meet such constraints. We defer this issue to Appendix C.*

4.1 The `PiDuce` *type system*

Few preliminary notations are introduced. Let Γ, Δ, called *environments*, be finite maps from variables to schemas. We write $\texttt{dom}(\Gamma)$ for the set of names in the domain of Γ. Let $\Gamma + \Delta$ be $(\Gamma \setminus \texttt{dom}(\Delta)) \cup \Delta$, where $\Gamma \setminus X$ removes from Γ all

Table 2
Typing rules.

Expressions :

$$\Gamma \vdash \texttt{()} : \texttt{()} \qquad \Gamma \vdash \texttt{n} : \texttt{n} \qquad \Gamma \vdash \texttt{s} : \texttt{s}$$

$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S} \qquad \frac{a \in L \quad \Gamma \vdash E : S}{\Gamma \vdash a[E] : L[S]} \qquad \frac{\Gamma \vdash E : S \quad \Gamma \vdash E' : S'}{\Gamma \vdash E\texttt{,}E' : S\texttt{,}S'}$$

Processes :

$$\overset{\text{(NIL)}}{\Gamma; \Delta \vdash \mathbf{0}} \qquad \overset{\text{(OUT)}}{\frac{\Gamma \vdash E : S \quad \Gamma + \Delta \vdash u : T \quad T \texttt{<:} \langle S\rangle^{\texttt{O}}}{\Gamma; \Delta \vdash u\texttt{!}(E)}}$$

$$\overset{\text{(SELECT)}}{\frac{\left(\Gamma + \Delta \vdash u_i : S_i \quad (\Gamma; \Delta) + \texttt{Env}(F_i) \vdash P_i \quad S_i \texttt{<:} \langle \texttt{schof}(F_i)\rangle^{\texttt{I}}\right)^{i \in 1..n}}{\Gamma; \Delta \vdash \texttt{select}\ \{u_i\texttt{?}(F_i)P_i\ ^{i \in 1..n}\}}}$$

$$\overset{\text{(NEW)}}{\frac{\Gamma + u : \langle S\rangle^{\kappa}; \Delta + u : \langle S\rangle^{\texttt{IO}} \vdash P}{\Gamma; \Delta \vdash \texttt{new}\ u : \langle S\rangle^{\kappa}\ \texttt{in}\ P}}$$

$$\overset{\text{(MATCH)}}{\frac{\Gamma + \Delta \vdash E : S \quad \left((\Gamma; \Delta) + \texttt{Env}(F_i) \vdash P_i\right)^{i \in 1..n} \quad S \texttt{<:} \sum_{i \in 1..n} \texttt{schof}(F_i)}{\Gamma; \Delta \vdash \texttt{match}\ E\ \texttt{with}\ \{F_i \Rightarrow P_i^{i \in 1..n}\}}}$$

$$\overset{\text{(SPAWN)}}{\frac{\Gamma; \Delta \vdash P \quad \Gamma; \Delta \vdash P'}{\Gamma; \Delta \vdash \texttt{spawn}\ \{P\}\ P'}} \qquad \overset{\text{(REPIN)}}{\frac{\Delta \vdash u : S \quad (\Gamma; \Delta) + \texttt{Env}(F) \vdash P \qquad S \texttt{<:} \langle \texttt{schof}(F)\rangle^{\texttt{I}}}{\Gamma; \Delta \vdash u\texttt{?*}(F)P}}$$

the bindings of names in X. Let also $(\Gamma; \Delta) + \Gamma'$ be the pair $\Gamma + \Gamma'; \Delta \setminus \texttt{dom}(\Gamma')$. Finally, let $\texttt{Env}(\cdot)$ be the least function such that:

$$\begin{array}{rll}
\texttt{Env}(S) &= \emptyset & \\
\texttt{Env}(u : F) &= u : \texttt{schof}(F) + \texttt{Env}(F) \ (u \notin \texttt{dom(Env)}(F)) & \\
\texttt{Env}(L[F]) &= \texttt{Env}(F) & \\
\texttt{Env}(F\texttt{,}F') &= \texttt{Env}(F) + \texttt{Env}(F') & (\texttt{dom(Env)}(F) \cap \texttt{dom(Env)}(F') = \emptyset) \\
\texttt{Env}(F + F') &= \texttt{Env}(F) & (\texttt{Env}(F) = \texttt{Env}(F')) \\
\texttt{Env}(\texttt{Y}) &= \texttt{Env}(\mathcal{F}(\texttt{Y})) &
\end{array}$$

The judgments $\Gamma \vdash E : S$ – read E has schema S in the environment Γ – and $\Gamma; \Delta \vdash P$ – read P is well typed in the environment Γ and local environment Δ – are the least relations satisfying the rules in Table 2.

Rules for expressions, (NIL) and (SPAWN) are standard. Rule (OUT) types outputs. By definition of subschema, the premise $T \texttt{<:} \langle S\rangle^{\texttt{O}}$ entails that u may carry messages of schema S. We note that u can be typed as a union of channel schemas, for example $u : \langle \texttt{a}[\texttt{int}] + \texttt{()}\rangle^{\texttt{O}} + \langle \texttt{b}[\texttt{string}] + \texttt{()}\rangle^{\texttt{O}}$. When this is the case, E must be a subschema of *every* schema carried by u. In this example, the unique possible schema for E is `()`. Rule (SELECT) types input-guarded choices. The first hypothesis types subjects. The second hypothesis types the continuation of every summand in the environment $\Gamma\,;\,\Delta$ plus that defined by the pattern. The third hypothesis checks the exhaustiveness of every pattern. As for outputs the hypothesis S_i `<:` $\langle \mathtt{schof}(F_i)\rangle^{\mathtt{I}}$ does not strictly require u_i to be a channel schema. Rule (NEW) types `new` $u : \langle S\rangle^{\kappa}$ `in` P in $\Gamma\,;\,\Delta$ provided that P is typable with in $\Gamma + u : \langle S\rangle^{\kappa}\,;\,\Delta + u : \langle S\rangle^{\mathtt{IO}}$. The first component of the pair of environments is extended with the exported schema $\langle S\rangle^{\kappa}$ of the channel; this definition is used for typing expressions to be sent as messages (see rule (OUT)). The second component is extended with the internal schema of the channel $\langle S\rangle^{\mathtt{IO}}$; this definition is used for typing subjects of inputs and outputs (see rules (OUT), (SELECT), and (REPIN)). Rule (MATCH) derives the typing of `match` E `with` $\{F_i \Rightarrow P_i^{\,i\in 1..n}\}$ provided E and P_i are well typed in the environments $\Gamma + \Delta$ and $(\Gamma\,;\,\Delta) + \mathtt{Env}(F_i)$, respectively. The third hypothesis checks the exhaustiveness of patterns with respect to the schema of E. Rule (REPIN) is similar to (SELECT) but the subject is checked to be local.

Remark 4 *The* `PiDuce` *compiler also verifies whether patterns in* `match` *operators are redundant. In particular in rule (*MATCH*) the compiler verifies that, for every* $1 \le i \le n-1$, $\mathtt{schof}(F_i)$ `<:` S *and, for every* $2 \le j \le n$, $\mathtt{schof}(F_j) \not<: \sum_{k<j} \mathtt{schof}(F_j)$*. In case, the user is warned with suitable messages.*

5 Pattern matching and local operational semantics

This section defines the semantics of patterns and processes. In order to cope with values that may carry channels, both the pattern matching and the transition relation take an associated environment into account. As regards processes, this section details the semantics of operations that are performed by a single `PiDuce` runtime environment. The operations retaining a distributed semantics are discussed in Section 6.

5.1 Pattern matching

Let σ and σ' be two substitutions with disjoint domains. We write $\sigma + \sigma'$ to denote the substitution that is the union of σ and σ'. Every union in the following rules is always well defined because of the linearity constraint on patterns. Let a *marker* be an object of the form x/V; let Φ be a possibly empty sequence of patterns or markers separated by $::$ and let $[\,]$ be the empty

Table 3
Pattern matching rules.

(PM1)
$$\Delta \vdash \texttt{()} \in [\,] \rightsquigarrow \emptyset$$

(PM2)
$$\frac{\Delta \vdash V \in \Phi \rightsquigarrow \sigma}{\Delta \vdash V \in \texttt{()} :: \Phi \rightsquigarrow \sigma}$$

(PM3)
$$\frac{\Delta \vdash V \in \Phi \rightsquigarrow \sigma \quad V' = V''@V}{\Delta \vdash V \in x/V' :: \Phi \rightsquigarrow \sigma + [x \mapsto V'']}$$

(PM4)
$$\frac{\texttt{b <: B} \quad \Delta \vdash V \in \Phi \rightsquigarrow \sigma}{\Delta \vdash \texttt{b}@V \in \texttt{B} :: \Phi \rightsquigarrow \sigma}$$

(PM5)
$$\frac{\Delta(u) \texttt{ <: } S \quad \Delta \vdash V \in \Phi \rightsquigarrow \sigma}{\Delta \vdash u@V \in S :: \Phi \rightsquigarrow \sigma}$$

(PM6)
$$\frac{a \in L \quad \Delta \vdash V \in F \rightsquigarrow \sigma \quad \Delta \vdash V' \in \Phi \rightsquigarrow \sigma'}{\Delta \vdash a[V]@V' \in L[F] :: \Phi \rightsquigarrow \sigma + \sigma'}$$

(PM7)
$$\frac{\Delta \vdash V \in F :: x/V :: \Phi \rightsquigarrow \sigma}{\Delta \vdash V \in (x : F) :: \Phi \rightsquigarrow \sigma}$$

(PM8)
$$\frac{\Delta \vdash V \in F :: \Phi \rightsquigarrow \sigma}{\Delta \vdash V \in (F + F') :: \Phi \rightsquigarrow \sigma}$$

(PM9)
$$\frac{\Delta \vdash V \in F' :: \Phi \rightsquigarrow \sigma \quad \Delta \vdash V \notin F :: \Phi}{\Delta \vdash V \in (F + F') :: \Phi \rightsquigarrow \sigma}$$

(PM10)
$$\frac{\Delta \vdash V \in F :: F' :: \Phi \rightsquigarrow \sigma}{\Delta \vdash V \in (F\texttt{,}F') :: \Phi \rightsquigarrow \sigma}$$

(PM11)
$$\frac{\Delta \vdash V \in \mathcal{F}(\texttt{Y}) :: \Phi \rightsquigarrow \sigma}{\Delta \vdash V \in \texttt{Y} :: \Phi \rightsquigarrow \sigma}$$

(PM12)
$$\frac{\begin{array}{c}\Delta \vdash V \in S^n \rightsquigarrow \emptyset \qquad \Delta \vdash V' \in \Phi \rightsquigarrow \sigma \\ (V' = W@V'' \text{ and } W \neq \texttt{()}) \text{ implies } (\Delta \vdash V@W \notin S^* \text{ or } \Delta \vdash V'' \notin \Phi)\end{array}}{\Delta \vdash V@V' \in S^* :: \Phi \rightsquigarrow \sigma}$$

sequence. In the following, tailing [ ]'s are always omitted.

The *pattern matching* of a value V with respect to a sequence Φ in an environment Δ, written $\Delta \vdash V \in \Phi \rightsquigarrow \sigma$ is defined by the rules in Table 3. We write $\Delta \vdash V \in \Phi$ if there exists σ such that $\Delta \vdash V \in \Phi \rightsquigarrow \sigma$; we write $\Delta \vdash V \notin \Phi$ if not $\Delta \vdash V \in \Phi$. Let S^n be $S\texttt{,}\dots\texttt{,}S$ with n repetitions of S; let S^0 be `()`.

Rule (PM1) matches `()` with the empty sequence. This rule must be read in conjunction with (PM2), which removes void patterns in head position of sequences. Rule (PM3) defines markings. A marking x/V' is inserted in Φ by patterns $x : F$ – see rule (PM7); it records the value V' that must be matched by Φ when a variable binder is found. The rule binds x to the prefix of V' that has been matched by F. Rule (PM4) matches constants with basic schemas. Rule (PM5) matches channels with patterns that do not contain variables and, assuming that $\Delta(\texttt{u})$ is a channel schema, that are greater than a channel schema. Rule (PM6) deals with labelled values. Rule (PM7) defines the pattern matching of a sequence $(x : F) :: \Phi$. In this case, the value V must match with $F :: \Phi$ and the prefix of V matching with F must be bound to x. This is the purpose of the marking that is inserted between F and Φ. Rules (PM8) and (PM9) define the pattern matching for union patterns. They implement the *first match policy*: in a pattern $F+F'$ the match with F is attempted and, if this fails, the match with F' is tried. Rule (PM10) turns sequence patterns into sequences Φ. Rule (PM11) defines pattern matching of pattern names in the obvious way. Finally, rule (PM12) defines the pattern matching for $S^* :: \Phi$ sequences. The pattern S^* is equal to the choice $\texttt{()}+S+(S\texttt{,}S)+(S\texttt{,}S\texttt{,}S)+\cdots$ but it is managed by a policy different than the one of rules (PM8) and (PM9). In this case the standard policy is the *longest match* one: a partition $V@V'$ of the value is looked for such that V is the longest prefix matching with S^* and V' is a suffix matching with Φ.

Rules (PM8), (PM9), and (PM12) make the parsing for patterns $F + F'$ and S^* deterministic. The pattern matching of Table 3 is therefore unambiguous.

Proposition 5 *If $\Delta \vdash V \in \Phi$ then there exists a unique σ such that $\Delta \vdash V \in \Phi \rightsquigarrow \sigma$.*

Notwithstanding this unicity property, the implementation of the pattern matching algorithm is not straightforward. The critical rule is (PM12), because it is not obvious where the value to be matched should be split. It is worth to remark that expanding S^* into $S\texttt{,}S^* + \texttt{()}$, thus relying on the first match policy for the choice schema to yield the longest matching prefix, does not always produce the desired results, as noted in [34]. Indeed, consider the schema $(a[\,] + a[\,]\texttt{,}b[\,])^*\texttt{,}(b[\,] + \texttt{()})$. This would be expanded into $((a[\,] + a[\,]\texttt{,}b[\,])\texttt{,}(a[\,] + a[\,]\texttt{,}b[\,])^* + \texttt{()})\texttt{,}(b[\,] + \texttt{()})$ and the value $a[\,]\texttt{,}b[\,]$ would be split into $a[\,]$ matching with $(a[\,]+a[\,]\texttt{,}b[\,])^*$ and $b[\,]$ matching with $(b[\,]+\texttt{()})$. However, $a[\,]\texttt{,}b[\,]$ is the longest prefix matching with $(a[\,]+a[\,]\texttt{,}b[\,])^*$ with $\texttt{()}$ matching with $(b[\,] + \texttt{()})$. In order to implement the matching of a value V against a pattern list $S^* :: \Phi$, a naive implementation may attempt splitting the value beginning from its *right end*, trying first to match $\texttt{()}$ with Φ and V with S^*. If this fails, the smallest non-void suffix of V is matched against Φ, and the remaining prefix against S^*, and so forth. More efficient solutions are discussed in the literature [17].

5.2 The (local) transition relation

Let $\texttt{l}, \texttt{l}', \ldots$ range over a countably infinite set of *locations*. We assume a relation @ mapping channels to locations and we write $u@\texttt{l}$ for *u located at* $\texttt{l}$. With an abuse of notation, we extend -$@\texttt{l}$ to variables. The relation $x@\texttt{l}$ is always true (since variables may be instantiated by channels located at $\texttt{l}$). The following transition relation is defined when subjects of selects and replications

Table 4
Local transition relation.

(TR1)
$$\frac{E \Downarrow_{\mathtt{dom}(\Gamma)} V}{\Gamma \vdash_{\mathtt{l}} u\,\mathtt{!}(E) \xrightarrow{u\,\mathtt{!}(V)} \mathbf{0}}$$

(TR2)
$$\frac{(u_i @ \mathtt{l})^{i \in I}}{\Gamma \vdash_{\mathtt{l}} \mathtt{select}\ \{u_i?(F_i)P_i^{\,i \in I}\} \xrightarrow{u_i\mathtt{?}(F_i)} P_i}$$

(TR3)
$$\frac{\Gamma + u\!:\!\langle S\rangle^{\kappa} \vdash_{\mathtt{l}} P \xrightarrow{\mu} Q \quad u \notin \mathtt{fv}(\mu) \cup \mathtt{bv}(\mu)}{\Gamma \vdash_{\mathtt{l}} \mathtt{new}\ u\!:\!\langle S\rangle^{\kappa}\ \mathtt{in}\ P \xrightarrow{\mu} \mathtt{new}\ u\!:\!\langle S\rangle^{\kappa}\ \mathtt{in}\ Q}$$

(TR4)
$$\frac{\Gamma + v\!:\!\langle S\rangle^{\kappa} \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u\,\mathtt{!}(V)} Q \quad v \neq u \quad v \in \mathtt{fv}(V) \setminus \mathtt{dom}(\Gamma')}{\Gamma \vdash_{\mathtt{l}} \mathtt{new}\ v\!:\!\langle S\rangle^{\kappa}\ \mathtt{in}\ P \xrightarrow{(\Gamma' + v:\langle S\rangle^{\kappa})u\,\mathtt{!}(V)} Q}$$

(TR5)
$$\frac{E \Downarrow_{\mathtt{dom}(\Gamma)} V \quad (\Gamma \vdash V \notin F_i)^{i \in 1..j-1} \quad \Gamma \vdash V \in F_j \rightsquigarrow \sigma}{\Gamma \vdash_{\mathtt{l}} \mathtt{match}\ E\ \mathtt{with}\ \{F_i \Rightarrow P_i^{\,i \in 1..n}\} \xrightarrow{\tau} P_j\sigma}$$

(TR6)
$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\mu} P' \quad \mathtt{bv}(\mu) \cap \mathtt{fv}(Q) = \emptyset}{\Gamma \vdash_{\mathtt{l}} \mathtt{spawn}\ \{P\}\ Q \xrightarrow{\mu} \mathtt{spawn}\ \{P'\}\ Q}$$

(TR7)
$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\mu} P' \quad \mathtt{bv}(\mu) \cap \mathtt{fv}(Q) = \emptyset}{\Gamma \vdash_{\mathtt{l}} \mathtt{spawn}\ \{Q\}\ P \xrightarrow{\mu} \mathtt{spawn}\ \{Q\}\ P'}$$

(TR8)
$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u\,\mathtt{!}(V)} P' \quad \Gamma \vdash_{\mathtt{l}} Q \xrightarrow{u\mathtt{?}(F)} Q' \quad \mathtt{dom}(\Gamma') \cap \mathtt{fv}(Q) = \emptyset \quad \Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma}{\Gamma \vdash_{\mathtt{l}} \mathtt{spawn}\ \{P\}\ Q \xrightarrow{\tau} \mathtt{new}\ \Gamma'\ \mathtt{in}\ \mathtt{spawn}\ \{P'\}\ Q'\sigma}$$

(TR9)
$$\frac{u @ \mathtt{l}}{\Gamma \vdash_{\mathtt{l}} u?{*}(F)P \xrightarrow{u\mathtt{?}(F)} \mathtt{spawn}\ \{P\}\ u?{*}(F)P}$$

are local to the PiDuce runtime environment. The general case is discussed in Section 6.

Let μ range over input labels $u?(F)$, bound output labels $(\Gamma)u\,\mathtt{!}(V)$ with $\mathtt{dom}(\Gamma) \subseteq \mathtt{fv}(V)$, and τ. Let also $\mathtt{fv}(u?(F)) = \{u\}$, $\mathtt{fv}((\Gamma)u\,\mathtt{!}(V)) = \{u\} \cup (\mathtt{fv}(V) \setminus \mathtt{dom}(\Gamma))$, $\mathtt{bv}(u?(F)) = \mathtt{fv}(F)$, $\mathtt{bv}((\Gamma)u\,\mathtt{!}(V)) = \mathtt{dom}(\Gamma)$, and $\mathtt{fv}(\tau) = \mathtt{bv}(\tau) = \emptyset$. The *(local) transition relation* of PiDuce, $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\mu} Q$, is the least relation satisfying the rules in Table 4 plus the symmetric of the communication rule (TR8).

The transition relation is also closed under alpha-conversion. For example, if $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u\,\mathtt{!}(V)} Q$ then $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma'\alpha)u\,\mathtt{!}(V\alpha)} Q\alpha$ for every alpha-conversion α.

The transition relation of Table 4 is similar to that of the pi calculus [29], except for the environment Γ, which is partially supplied by enclosing `new` operators and partially by the global environment.

We discuss rules (TR1), (TR3), (TR4), (TR5), and (TR8); the arguments about the other rules are omitted. Rule ($\textsc{tr}1$) defines the semantics of $u\texttt{!}(E)$. According to this semantics, E is evaluated into a $\texttt{dom}(\Gamma)$-value V and V is delivered. Rules ($\textsc{tr}3$) and ($\textsc{tr}4$) define the semantics of outputs when they are underneath local definitions of channels. There are two cases: (i) the local channel does not occur in the message, (ii) the local channel does occur. The case (i) is managed by ($\textsc{tr}3$): in this case the output operation is simply lifted outside the `new` and the label of the transition does not change. The case (ii) is managed by ($\textsc{tr}4$). The label gathers the local channels (and their schema) that are transmitted. The third hypothesis of ($\textsc{tr}4$) verifies that the channel v occurs in the message; in this case the environment of the label in the conclusion is extended with v and its schema. This extension of the label, which is different from pi calculus for the presence of schemas, is meant to capture the property that when a Web service `URL` is shipped, the `WSDL` document is also sent. (This `WSDL` contains, for instance, the protocol that must be used to invoke the service and the schemas of arguments and of the result.) Rule ($\textsc{tr}5$) defines the semantics of $\texttt{match}\ E\ \texttt{with}\ \{F_i \Rightarrow P_i^{\,i\in 1..n}\}$. According to this rule, E is evaluated, then the first pattern F_j matching the value is chosen and the continuation P_j is run with the substitution returned by the pattern matching algorithm. Rule ($\textsc{tr}8$) makes two parallel processes emitting and receiving a message on the same channel communicate. To this aim the message is matched against the pattern and the resulting substitution is applied to the receiver process. It is worth to notice that our semantics admits communications on variables that are channels. This case intends to model those communications involving channels that have not been published (the `WSDL` has not been created) as their declaration has been lifted to the label of the transition relation, but they do not occur in the domain of the environment. The publication happens as soon as the channel is extruded to a remote machine (see rule ($\textsc{dtr}1$) in Section 6). [1]

6 Distributed operational semantics

The underlying model of `PiDuce` is distributed; it consists of a number of runtime environments – that may be `PiDuce` runtimes or not –, which execute at different locations and interact by exchanging messages over channels. In this section we describe the distributed semantics of the `PiDuce` language.

A `PiDuce` *machine* is a collection of runtime environments:

$$\Gamma_1 \vdash_{\mathtt{l}_1} P_1 \parallel \cdots \parallel \Gamma_n \vdash_{\mathtt{l}_n} P_n$$

such that

[1] The `PiDuce` implementation eagerly creates the `WSDL` interface of a service as soon as the corresponding `new` in the object code is executed.

(1) $\mathtt{l}_1, \ldots, \mathtt{l}_n$ are pairwise different;
(2) $\Gamma_1, \ldots, \Gamma_n$ are *localized* with respect to $\mathtt{l}_1, \ldots, \mathtt{l}_n$, namely $u \in \mathtt{dom}(\Gamma_i)$ and $u@\mathtt{l}_j$ implies $u \in \mathtt{dom}(\Gamma_j)$.

`PiDuce` machines are ranged over by M, N, etc. We also let $\mathtt{dom}(\Gamma_1 \vdash_{\mathtt{l}_1} P_1 \parallel \cdots \parallel \Gamma_n \vdash_{\mathtt{l}_n} P_n) = \bigcup_{i \in 1..n} \mathtt{dom}(\Gamma_i)$. Processes in the runtime environments extend those of Table 1 with operations dealing with remote locations:

- *input on remotely located channels*: the subjects u_i in $\mathtt{select}\ \{u_i?(F_i)P_i{}^{i \in I}\}$ may be non-local;
- ***new** at remote location*: the process $\mathtt{new}\ u : \langle S \rangle^{\mathtt{IO}}\ \mathtt{at}\ \mathtt{l}\ \mathtt{in}\ P$ delegates the runtime environment located at $\mathtt{l}$, which may be remote, to create the runtime support for u. The syntax requires the capability of the schema to be `IO` because in order for the operation to be useful the continuation P needs to be able to perform both input and output operations on u;
- *import of services*: the process $\mathtt{import}\ u : S \rightarrow T = v\ \mathtt{in}\ P$ downloads the `WSDL` of the channel v, verifies that it is a subschema of $\langle S\mathtt{,} \langle T \rangle^{\mathtt{O}} \rangle^{\mathtt{O}}$ and replaces u with v in the continuation P. The channel v represents a synchronous – *request-response* in `WSDL` jargon – operation in a remote service. A special case of this process is $\mathtt{import}\ u : \langle S \rangle^{\mathtt{O}} = v\ \mathtt{in}\ P$ that verifies the `WSDL` of v to be a subschema of $\langle S \rangle^{\mathtt{O}}$. In this section the notation $S \rightarrow T$ may be considered as syntactic sugar for the schema $\langle S\mathtt{,} \langle T \rangle^{\mathtt{O}} \rangle^{\mathtt{O}}$; the differences between $S \rightarrow T$ and $\langle S\mathtt{,} \langle T \rangle^{\mathtt{O}} \rangle^{\mathtt{O}}$ have to do with interoperability and will be discussed in Section 7.1.

Among these operators, `import` is the most interesting one because it permits `PiDuce` processes to access existing services. For example, the code

```
import fact : Int → Int = "www.mathfunctions.edu/fact"
in new u : ⟨int⟩O
in spawn { fact!(5,u) } u?(v:Int) printInt!(v)
```

imports the operation `fact` which is provided by a Web service located at `www.mathfunctions.edu/fact`, invokes `fact` with `5`, and prints the result.

The runtime environment also uses a further operation dealing with remote locations:

- *linear forwarder* $u \multimap v$ that forwards a message on a channel u to v. This operator implements input operations on remotely located channels; its theory has been developed in [18] and will be recalled below.

The type system of Table 2 is extended with the rules in Table 5 for `new` binders at remote locations, imports and linear forwarders. Rule (NEWAT) types the creation of channels at remote locations; the typing rule is similar to (NEW). Rules (IMPORT) and (IMPORT-A) type import of channels by checking P to

Table 5
Typing rules for distributed PiDuce.

(NEWAT)
$$\frac{(\Gamma\,;\,\Delta) + u\!:\!\langle S\rangle^{\mathtt{IO}} \vdash P}{\Gamma\,;\,\Delta \vdash \mathtt{new}\ u\!:\!\langle S\rangle^{\mathtt{IO}}\ \mathtt{at}\ \mathtt{l}\ \mathtt{in}\ P}$$

(IMPORT)
$$\frac{(\Gamma\,;\,\Delta) + u\!:\!\langle S\,\mathtt{,}\,\langle T\rangle^{\mathtt{O}}\rangle^{\mathtt{O}} \vdash P \quad \Gamma(v) \mathrel{\mathtt{<:}} \langle S\,\mathtt{,}\,\langle T\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}}{\Gamma\,;\,\Delta \vdash \mathtt{import}\ u\!:\!S \rightarrow T = v\ \mathtt{in}\ P}$$

(IMPORT-A)
$$\frac{(\Gamma\,;\,\Delta) + u\!:\!\langle S\rangle^{\mathtt{O}} \vdash P \quad \Gamma(v) \mathrel{\mathtt{<:}} \langle S\rangle^{\mathtt{O}}}{\Gamma\,;\,\Delta \vdash \mathtt{import}\ u\!:\!\langle S\rangle^{\mathtt{O}} = v\ \mathtt{in}\ P}$$

(LFORWD)
$$\frac{\Gamma \vdash v : \langle S\rangle^{\mathtt{O}} \quad \Gamma(u) \mathrel{\mathtt{<:}} \langle S\rangle^{\mathtt{I}}}{\Gamma\,;\,\Delta \vdash u \multimap v}$$

be well typed in $(\Gamma\,;\,\Delta) + u : R$ (u is removed from Δ because it is not a local channel), where R is $\langle S\,\mathtt{,}\,\langle T\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}$ or $\langle S\rangle^{\mathtt{O}}$, according to whether v is a request-response operation or not. The rules also verify that the schema of the imported channel, which is stored in the global environment, is compatible with R. Rule (LFORWD) types linear forwarders. The hypotheses, which require that u and v can be used for respectively receiving and sending values, are in correspondence with those for typing the process $\mathtt{select}\ \{u\mathtt{?}(x : R)\ v\mathtt{!}(x)\}$ – where R is the schema of the messages accepted by u – with the following additional constraints:

(1) the schema of u is taken from the global environment because u is not local;
(2) the schema of v is taken from the global environment as well, because the linear forwarder process is executed on a remote machine;
(3) no subschema of $\Gamma(v)$ is considered because processes $u \multimap v$ are generated by the PiDuce runtime and, by definition, v always has a schema of shape $\langle S\rangle^{\mathtt{O}}$.

Typing is extended to machines as follows. Let $[\Gamma]^{\mathtt{IO}}_{\mathtt{l}}$ be the environment

$$[\Gamma]^{\mathtt{IO}}_{\mathtt{l}}(u) \;=\; \begin{cases} \langle S\rangle^{\mathtt{IO}} & \text{if } u@\mathtt{l} \text{ and } \Gamma(u) = \langle S\rangle^{\kappa} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The operation $[\Gamma]^{\mathtt{IO}}_{\mathtt{l}}$ is meant to define the environment for local channels: it extracts the channels local at l out of Γ and replaces the capability with IO because IO is the capability of local channels (*cf.* rule (NEW) in Table 2). We recall that, according to our notation, if x is a variable in $\mathtt{dom}(\Gamma)$ and $\Gamma(x)$ is a channel schema, then $x \in \mathtt{dom}([\Gamma]^{\mathtt{IO}}_{\mathtt{l}})$, too, because $x@\mathtt{l}$ is always true.

Let $\vdash \mathsf{M}$, read M is *well-typed*, if the following properties hold:

(i) for every $\Gamma \vdash_{\mathtt{l}} P$ in M: $\Gamma\,;\,[\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash P$ and
(ii) (*machine consistency*) if $\Gamma \vdash_{\mathtt{l}} P$ and $\Gamma' \vdash_{\mathtt{l}'} P'$ in M and $u \in \mathtt{dom}(\Gamma')$ and $u@\mathtt{l}$, then $u \in \mathtt{dom}(\Gamma)$ and $\Gamma(u) \mathrel{\mathtt{<:}} \Gamma'(u)$. (This constraint only

regards variables with channel schemas.)

Therefore, a machine is well-typed if every runtime environment in it is well-typed and the runtime environments access to remote channels with schemas that are superschemas of the actual ones. This is also the case for global accesses that are located at the same runtime environment (take $\mathtt{l} = \mathtt{l}'$ in case (ii)). For instance, when v is located at the same runtime environment executing $\mathtt{import}\ u : \langle S \rangle^{\mathsf{O}} = v\ \mathtt{in}\ P$. We notice that if $\vdash \mathsf{M} \parallel \mathsf{N}$ then $\vdash \mathsf{M}$ and $\vdash \mathsf{N}$.

Next we extend the (local) transition relation with the semantics of the operations dealing with remote locations. To this aim we drop the assumption in Section 5 that subjects of selects are local to the PiDuce runtime environment, as well as that new channels are always created locally to the runtime environment. In order to account for the new operations we extend the notation so that μ also ranges over the labels $u : S$, $(u@\mathtt{l} : S)$, and $(\Gamma) u \multimap v$ with $\mathtt{dom}(\Gamma) \subseteq \{v\}$, too. Let $\mathtt{fv}(u@\mathtt{l} : S) = \{u\}$, $\mathtt{fv}((u : S)) = \emptyset$, $\mathtt{fv}((\Gamma) u \multimap v) = \{u, v\} \setminus \mathtt{dom}(\Gamma)$ and let $\mathtt{bv}(u : S) = \emptyset$, $\mathtt{bv}((u@\mathtt{l} : S)) = \{u\}$, $\mathtt{bv}((\Gamma) u \multimap v) = \mathtt{dom}(\Gamma)$. We write $\mathtt{spawn}_{i \in 1..n}\ \{P_i\}\ \ Q$ for $\mathtt{spawn}\ \{P_1\}\ \cdots \mathtt{spawn}\ \{P_n\}\ \ Q$. As usual $\uplus$ denotes disjoint union. The transition relations use the following operations on environments:

$\Gamma@\mathtt{l}$ restricts Γ to variables located at $\mathtt{l}$:

$$(\Gamma@\mathtt{l})(u) = \begin{cases} \Gamma(u) & \text{if } u \in \mathtt{dom}(\Gamma) \text{ and } u@\mathtt{l} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\Gamma \setminus \mathtt{l}$ removes from Γ the variables located at $\mathtt{l}$:

$$(\Gamma \setminus \mathtt{l})(u) = \begin{cases} \Gamma(u) & \text{if } u \in \mathtt{dom}(\Gamma) \text{ and not } (u@\mathtt{l}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We write $\Gamma \setminus \mathtt{l}, \mathtt{l}'$ for $(\Gamma \setminus \mathtt{l}) \setminus \mathtt{l}'$.

$\Gamma\, \mathsf{meet}\, \Gamma'$ defines an environment that includes the domains of Γ and Γ' and that associates to every variable a subschema of those associated by Γ and Γ':

$$(\Gamma\, \mathsf{meet}\, \Gamma')(u) = \begin{cases} \Gamma(u) & \text{if } u \in \mathtt{dom}(\Gamma) \setminus \mathtt{dom}(\Gamma') \\ \Gamma'(u) & \text{if } u \in \mathtt{dom}(\Gamma') \setminus \mathtt{dom}(\Gamma) \\ S & \text{if } u \in \mathtt{dom}(\Gamma) \cap \mathtt{dom}(\Gamma') \\ & \quad \text{and } S \mathrel{<:} \Gamma(u) \text{ and } S \mathrel{<:} \Gamma'(u) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The meet operation is used in the transition relation to guess the schema of channels in messages that are located at neither the source nor the destination runtime environment.

Table 6
Distributed transition relation.

rules for $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\mu} Q$

(TR10)

$$\frac{(u_i @ \mathtt{l})^{i \in I} \qquad \left(u_j \not@ \mathtt{l} \qquad \Gamma \vdash u_j : \langle S_j \rangle^{\kappa}\right)^{j \in J} \qquad J \neq \emptyset}{\begin{array}{l}\Gamma \vdash_{\mathtt{l}} \mathtt{select}\ \{u_i\mathtt{?}(F_i)P_i{}^{i \in I \uplus J}\} \xrightarrow{\tau} \\ \qquad \mathtt{new}\ (v_j : \langle S_j \rangle^{\mathtt{0}})^{j \in J}\ \mathtt{in} \\ \qquad\quad \mathtt{spawn}_{j \in J}\ \{u_j \multimap v_j\} \\ \qquad\quad \mathtt{select}\ \{\quad u_i\mathtt{?}(F_i)(\mathtt{spawn}_{k \in J}\ \{v_k\mathtt{?}(x : S_k)\ u_k\mathtt{!}(x)\}\ P_i)^{i \in I} \\ \qquad\qquad\qquad v_j\mathtt{?}(F_j)(\mathtt{spawn}_{k \in J \setminus j}\ \{v_k\mathtt{?}(x : S_k)\ u_k\mathtt{!}(x)\}\ P_j)^{j \in J} \quad \}\end{array}}$$

(TR11)

$$\frac{\mathtt{l} \neq \mathtt{l}'}{\Gamma \vdash_{\mathtt{l}} \mathtt{new}\ u : \langle S \rangle^{\mathtt{IO}}\ \mathtt{at}\ \mathtt{l}'\ \mathtt{in}\ P \xrightarrow{(u @ \mathtt{l}' : \langle S \rangle^{\mathtt{IO}})} P}$$

(TR12)

$$\Gamma \vdash_{\mathtt{l}} \mathtt{import}\ u : S = v\ \mathtt{in}\ P \xrightarrow{\tau} P\{^{v}/_{u}\}$$

(TR13)

$$\Gamma \vdash_{\mathtt{l}} u \multimap v \xrightarrow{u \multimap v} \mathbf{0}$$

(TR14)

$$\frac{\Gamma + v : \langle S \rangle^{\kappa} \vdash_{\mathtt{l}} P \xrightarrow{u \multimap v} Q}{\Gamma \vdash_{\mathtt{l}} \mathtt{new}\ v : \langle S \rangle^{\kappa}\ \mathtt{in}\ P \xrightarrow{(v : \langle S \rangle^{\kappa}) u \multimap v} Q}$$

rules for $\mathsf{M} \xrightarrow{\Delta} \mathsf{N}$

(DTR1)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(v_i : S_i{}^{i \in I}) u \mathtt{!}(V)} Q \qquad u @ \mathtt{l}' \qquad (v_i @ \mathtt{l} \qquad v_i \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Gamma'))^{i \in I} \\ \Delta = v_i : S_i{}^{i \in I} + ((\Gamma|_{\mathtt{fv}(V)}) \setminus \mathtt{l}')\ \mathsf{meet}\ ((\Gamma'|_{\mathtt{fv}(V)}) \setminus \mathtt{l})\end{array}}{\Gamma \vdash_{\mathtt{l}} P \parallel \Gamma' \vdash_{\mathtt{l}'} R \quad \xrightarrow{\Delta \setminus \mathtt{l}, \mathtt{l}'} \Gamma + v_i : S_i{}^{i \in I} \vdash_{\mathtt{l}} Q \parallel \Gamma' + \Delta \vdash_{\mathtt{l}'} \mathtt{spawn}\ \{u\mathtt{!}(V)\}\ R}$$

(DTR2)

$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(u @ \mathtt{l}' : \langle S \rangle^{\mathtt{IO}})} Q \quad u \notin \mathtt{dom}(\Gamma') \cup \mathtt{dom}(\Gamma)}{\Gamma \vdash_{\mathtt{l}} P \parallel \Gamma' \vdash_{\mathtt{l}'} R \quad \longrightarrow \quad \Gamma + u : \langle S \rangle^{\mathtt{IO}} \vdash_{\mathtt{l}} Q \parallel \Gamma' + u : \langle S \rangle^{\mathtt{IO}} \vdash_{\mathtt{l}'} R}$$

(DTR3)

$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma'') u \multimap v} Q \quad u @ \mathtt{l}' \quad \Gamma' \vdash u : \langle S \rangle^{\kappa} \quad \mathtt{dom}(\Gamma'') \cap \mathtt{dom}(\Gamma') = \emptyset \quad \Gamma''' = \Gamma|_{\{v\}} + \Gamma''}{\Gamma \vdash_{\mathtt{l}} P \parallel \Gamma' \vdash_{\mathtt{l}'} R \quad \longrightarrow \quad \Gamma + \Gamma'' \vdash_{\mathtt{l}} Q \parallel \Gamma' + \Gamma''' \vdash_{\mathtt{l}'} \mathtt{spawn}\ \{u\mathtt{?}(x : S)\ v\mathtt{!}(x)\}\ R}$$

(DTR4)

$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\tau} Q}{\Gamma \vdash_{\mathtt{l}} P \quad \longrightarrow \quad \Gamma \vdash_{\mathtt{l}} Q}$$

(DTR5)

$$\frac{\mathsf{M} \xrightarrow{\Delta} \mathsf{N} \quad (\mathtt{dom}(\mathsf{N}) \setminus \mathtt{dom}(\mathsf{M})) \cap \mathtt{dom}(\Gamma) = \emptyset \quad \Delta @ \mathtt{l} \subseteq \Gamma}{\mathsf{M} \parallel \Gamma \vdash_{\mathtt{l}} P \xrightarrow{\Delta \setminus \mathtt{l}} \mathsf{N} \parallel \Gamma \vdash_{\mathtt{l}} P}$$

The *transition relation* $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\mu} Q$ and the *distributed transition relation* $\mathsf{M} \xrightarrow{\Delta} \mathsf{N}$ of `PiDuce` are the least relations satisfying the rules in Section 5 plus those in Table 6 (for the sake of brevity we omit Δ when it is the empty context). The distributed transition relation is closed under commutativity and associativity of $\parallel$. The label Δ on the distributed transition relation represents a set of assumptions regarding the type of free channels that two machines have exchanged between each other, where none of the machines hosts the exchanged channels.

Rule (TR10) defines selects with remote subjects. It translates the select process on-the-fly into another one using a local select. (This translation has been proposed for encoding distributed choice in [18].) To explain the transition we discuss the case of a select with three branches, one with a local subject u and the others with remote subjects v and w:

$$\texttt{select}\ \{u\texttt{?}(F)P \quad v\texttt{?}(F')Q \quad w\texttt{?}(F'')R\}$$

This select may be turned into a local one by creating two (local) siblings for v and w, let them be v' and w', respectively, and communicating to the channel managers of v and w the presence of these siblings. So the above process may be translated into

$$\begin{array}{ll}\texttt{new}\ v', w' : S', T'\ \texttt{in} & \texttt{spawn}\ \{v \multimap v'\}\ \texttt{spawn}\ \{w \multimap w'\} \\ & \texttt{select}\ \{u\texttt{?}(F)P \quad v'\texttt{?}(F')Q \quad w'\texttt{?}(F'')R\}\end{array}$$

However this translation is too rough because of the following problem. The purpose of the linear forwarder $v \multimap v'$ is to migrate to the remote location of v and forward one message to the location of v'. Similarly for $w \multimap w'$. By rule (TR2), the branch $u\texttt{?}(F)P$ may be chosen because of the presence of a message on u. This choice destroys the branches $v'\texttt{?}(F')Q$ and $w'\texttt{?}(F'')R$. Therefore, when messages for v' and w' will be delivered by the remote machines, such messages will never be consumed. To avoid these misbehaviors, one has to compensate the previous emission of linear forwarders by undoing them with $v'\texttt{?}(x : S')\ v\texttt{!}(x)$ and $w'\texttt{?}(x : T')\ w\texttt{!}(x)$. In case the picked branch is $v'\texttt{?}(F')Q$, by a similar argument, we have to compensate only one linear forwarder – the $w \multimap w'$. Therefore the correct translation for the distributed select is:

$$\begin{array}{lll}\texttt{new}\ v', w' : S', T'\ \texttt{in} & \multicolumn{2}{l}{\texttt{spawn}\ \{v \multimap v'\}\ \texttt{spawn}\ \{w \multimap w'\}} \\ & \texttt{select}\ \{ & u\texttt{?}(F)(\texttt{spawn}\ \{v'\texttt{?}(x : S')\ v\texttt{!}(x)\} \\ & & \quad\quad \texttt{spawn}\ \{w'\texttt{?}(x : T')\ w\texttt{!}(x)\}\ \ P) \\ & & v'\texttt{?}(F')(\texttt{spawn}\ \{w'\texttt{?}(x : T')\ w\texttt{!}(x)\}\ \ Q) \\ & & w'\texttt{?}(F'')(\texttt{spawn}\ \{v'\texttt{?}(x : S')\ v\texttt{!}(x)\}\ \ R)\ \ \}\end{array}$$

that is the term yielded by the (TR10) in this case. Rule (TR11) creates a channel remotely located at $\texttt{l}'$. To this aim a channel located at $\texttt{l}'$ is taken and the local name is replaced by this channel in the continuation. When $\texttt{l} = \texttt{l}'$, the process $\texttt{new}\ u : \langle S\rangle^{\texttt{IO}}\ \texttt{in}\ P$ is simply an abbreviation for $\texttt{new}\ u : \langle S\rangle^{\texttt{IO}}\ \texttt{at}\ \texttt{l}'\ \texttt{in}\ P$. In this case its semantics is defined by rules (TR3) and (TR4). Rule (DTR2) guarantees that such a channel is fresh at the remote location. Rule (TR12) imports a channel (the compiler type-checks the continuation under the assumption $u : S$ – see (IMPORT)). Rule (TR13) lifts the linear forwarder to the label. This rule and rule (DTR3) define a linear forwarder $u \multimap v$ as a small atom migrating to the remote location of u and becoming the process $u\texttt{?}(x : S)v\texttt{!}(x)$. Rule (TR14) accounts for linear forwarders $u \multimap v$ where v is local to the sender. In this case the environment of the receiver must be extended adequately.

Rule (DTR1) models the delivery of a message to a remote runtime environment $\mathtt{l}'$. When this occurs all the bound channels are created in the sender location $\mathtt{l}$ and the message is put in parallel with every process running at $\mathtt{l}'$. The rule extends the environments of $\mathtt{l}$ and $\mathtt{l}'$ with the new channels ${v_i}^{i \in I}$. Additionally, the environment Γ' of $\mathtt{l}'$ is extended with channels in $\mathtt{fv}(V) \setminus \{{v_i}^{i\in I}\}$ that are either undefined in Γ' or whose associated schema is too large. This is a subtle problem to deal with. Consider a channel $v \in \mathtt{fv}(V) \setminus \{{v_i}^{i\in I}\}$ that is located at $\mathtt{l}$. The machine at $\mathtt{l}'$ may already be aware of such channel either because it has been imported or because it has been received during a previous communication. The point is that $\Gamma(v)$ and $\Gamma'(v)$ are not equal in general. In particular, by the definition of $\vdash \mathsf{M}$, $\Gamma(v)$ <: $\Gamma'(v)$. Therefore the rule (DTR1) updates the environment of $\mathtt{l}'$ with $(\Gamma|_{\mathtt{fv}(V)})|_{\mathtt{l}}$. A similar problem is manifested by channels $v \in \mathtt{fv}(V) \setminus \{{v_i}^{i\in I}\}$ that are not located at $\mathtt{l}$ nor at $\mathtt{l}'$. In this case $\Gamma(v)$ and $\Gamma'(v)$ may be incomparable, as in general they are superschemas of the actual schema of v, which is defined on a machine $\mathtt{l}''$ other than $\mathtt{l}$ and $\mathtt{l}'$. Therefore we guess the right schema – the operation meet – and publish our guess in the label of the transition. It is the rule (DTR5) that checks the correctness of our guess when the right context environment is found. The rule removes the checked bindings from the environment, that is a successful distributed transition of a `PiDuce` machine has always labels with empty environments.[2] The other rules have been already described, except (DTR4) that lifts transitions in components to composite machines.

We conclude this section by asserting the soundness of the static semantics. Proofs are reported in the Appendix B. The first property, subject reduction, states that well-typed processes always transit to well-typed processes.

Theorem 6 (Subject Reduction) *Let* $\Gamma;\, [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash P$. *Then*

(1) if $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u\,!(V)} Q$, *then (a)* $\Gamma + \Gamma';\, [\Gamma + \Gamma']^{\mathtt{IO}}_{\mathtt{l}} \vdash Q$, *(b)* $\Gamma + [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash u{:}S$, $\Gamma + \Gamma' \vdash V{:}T$ *and* S <: $\langle T\rangle^{\mathtt{O}}$;
(2) if $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{u?(F)} Q$, *then (a)* $(\Gamma;\, [\Gamma]^{\mathtt{IO}}_{\mathtt{l}}) + \mathtt{Env}(F) \vdash Q$ *and (b)* $\Gamma + [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash u{:}S$ *with* S <: $\langle \mathtt{schof}(F)\rangle^{\mathtt{I}}$;
(3) if $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u \multimap v} Q$, *then (a)* $\Gamma + \Gamma';\, [\Gamma + \Gamma']^{\mathtt{IO}}_{\mathtt{l}} \vdash Q$ *and (b)* $\Gamma \vdash u : S$, $\Gamma + \Gamma' \vdash v{:}\langle T\rangle^{\mathtt{O}}$ *and* S <: $\langle T\rangle^{\mathtt{I}}$;
(4) if $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(u@\mathtt{l}':\langle S\rangle^{\mathtt{IO}})} Q$, *then* $(\Gamma;\, [\Gamma]^{\mathtt{IO}}_{\mathtt{l}}) + u{:}\langle S\rangle^{\mathtt{IO}} \vdash Q$;
(5) if $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\tau} Q$, *then* $\Gamma;\, [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash Q$.

[2] In the implementation this problem does not arise and there is no need for the meet operation as there is only one global environment that is shared among all the runtime environments.

Let $\vdash \mathsf{M}$. *Then*

(6) if $\mathsf{M} \xrightarrow{\Delta} \mathsf{N}$, *then* $\vdash \mathsf{N}$.

The first item of the subject reduction entails that the reduct Q of a $(\Gamma')u\,!\,(V)$-transition is typable provided the initial process P is typable. To this aim, the environment $\Gamma\,;\,[\Gamma]_{\mathtt{1}}^{\mathtt{IO}}$ must be suitably extended with the bindings in Γ'. This extension is similar to the one used in the rule (NEW) of the type system. In facts, bindings in Γ' are collected by surrounding new binders – see rule (TR4). The second item deals with inputs and entails the typability of the reduct in an environment extended with that of patterns. The subject reduction guarantees the exhaustivity of inputs. The third item is about linear forwarders. Such operations are introduced by `PiDuce` runtimes as described by rule (TR10). Therefore v must have schema $\langle T\rangle^{\mathtt{O}}$, for some T; the theorem guarantees that u has a schema S "compatible" with $\langle T\rangle^{\mathtt{O}}$, namely S <: $\langle T\rangle^{\mathtt{I}}$. The fourth item deals with creation of remote channels. The other items are not commented because obvious.

The second soundness property concerns *progress*, that is, an output on a channel will be consumed if an input on the same channel is available and a message or a linear forwarder is delivered to the remote runtime when it is present (we are assuming the absence of failures). In order to guarantee progress, it is necessary to restrict (well-formed) environments. To illustrate the problem, consider the following judgment:

$$\begin{array}{l} u : \langle \mathtt{int} + \mathtt{string}\rangle^{\kappa}, v : \mathtt{int} + \mathtt{string} \vdash_{\mathtt{1}} \\ \quad \mathtt{spawn}\ \{u\,!\,(v)\}\ \ u?(x : \mathtt{int} + \mathtt{string}) \\ \qquad \mathtt{match}\ x\ \mathtt{with}\ \{\mathtt{int} \Rightarrow P \quad \mathtt{string} \Rightarrow Q\} \end{array}$$

The reader may verify that this judgment can be derived in our type system. However, after the communication, the pattern matching fails because the schema of v is neither a subschema of `int` nor of `string` (see rule (PM5)). Another example is the following. Let Γ be $u : a[b[\ ]]$, $V = u$, and $F = a[v : b[\ ]]$. Then $\Gamma \vdash V : S$ and S <: $\mathtt{schof}(F)$ but there is no σ such that $\Gamma \vdash V \in F \leadsto \sigma$. In fact these circumstances never occur in practice: if a value is sent, it may contain either labels or constants or channels. Under this constraint, progress is always guaranteed.

We say that Γ is *channeled* if, for every $u \in \mathtt{dom}(\Gamma)$, $\Gamma(u)$ is a channel schema.

Theorem 7 (Progress) *Let* Γ *be channeled.*

(1) If $\Gamma \vdash V : S$ *and* S <: $\mathtt{schof}(F)$, *then there is* σ *such that* $\Gamma \vdash V \in F \leadsto \sigma$;

(2) If $\Gamma\,;\,[\Gamma]_{\mathtt{1}}^{\mathtt{IO}} \vdash P$, $\Gamma \vdash_{\mathtt{1}} P \xrightarrow{(\Gamma')u\,!\,(V)} Q'$, *and* $\Gamma \vdash_{\mathtt{1}} P \xrightarrow{u?(F)} Q''$, *then there is* Q *such that* $\Gamma \vdash_{\mathtt{1}} P \xrightarrow{\tau} Q$;

(3) If $\vdash (\Gamma \vdash_{\mathtt{1}} P \parallel \mathsf{M})$, $\Gamma \vdash_{\mathtt{1}} P \xrightarrow{(\Gamma')u\,!\,(V)} Q$, *and* u *is located at a location of*

M, *then* $\Gamma \vdash_{\mathtt{l}} P \parallel \mathsf{M} \xrightarrow{\Delta} \Gamma \vdash_{\mathtt{l}} Q \parallel \mathsf{N}$, *for some* N. *Similarly when the label is* $(\Gamma')u \multimap v$.

7 `PiDuce` and Web services

The language presented in the previous sections deals with all the fundamental aspects of Web service definitions and interactions. However, there is still a gap between `PiDuce` and the current technologies related to Web services. Such gap is finally closed in this section by extending `PiDuce` with additional constructs, though the primitive operations of the calculus are unchanged in their essence.

7.1 Defining request-response services

The basic communication mechanism in `PiDuce` is the asynchronous message passing. Other mechanisms that are primitive in Web services, such as *rendez-vous*, must be programmed by means of explicit continuations. In Section 6 we have already discussed the semantics of a construct that permits to import request-response operations. In that case, a request-response operation is typed with a schema $\langle S, \langle T \rangle^{\mathtt{O}} \rangle^{\mathtt{O}}$ and has the following intended behavior. When invoked, a fresh channel is sent with the actual data of type S. At the same time, the invoker spawns an input process catching the response on the fresh channel. This behavior is actually a well-known encoding of rendez-vous, which is incongruous with respect to reality where request-response operations return results using the *same* connection. This is the reason why an explicit schema constructor $S \to T$ has been used rather than $\langle S \mathtt{,} \langle T \rangle^{\mathtt{O}} \rangle^{\mathtt{O}}$. The `PiDuce` runtime (in particular, the Web interface, see Section 8.1) implements the invocations of a channel with schema $S \to T$ by extracting the actual data and continuation channel from the sent message, establishing a connection and sending the actual data over the connection, receiving the response from the same connection, and forwarding it on the continuation channel.

We can adopt a similar mechanism to define a service implementing a request-response operation. `PiDuce` processes are extended with

$\mathtt{new}\; u : S \to T \;\mathtt{in}\; P$

which differs from the `new` of Table 1 because the associated `WSDL` has its interaction pattern set to request-response, an `input message` part set to S, and an `output message` part set T. The behavior of u is the same as for the corresponding `import`.

So far a one-to-one correspondence between `PiDuce` channels and Web services (hence between `PiDuce` channels and `WSDL` resources) has been assumed. This assumption falls short in faithfully modeling real Web services where a `WSDL` resource corresponds to a set of *operations*. To overcome this limitation we need to extend schemas and processes in Table 1. The extension, illustrated in Table 7, is folklore in the community except for the definition of the subschema relation.

Table 7
`PiDuce` syntax with service extensions (I is finite).

$S ::=$	**schema**	$P ::=$	**process**
$\cdots$	as in Table 1	$\cdots$	as in Table 1
$\{m_i : S_i{}^{i \in I}\}$	(record schema)	$\texttt{new}\ r : S\ \texttt{in}\ P$	(new)
		$\texttt{import}\ r : S = v\ \texttt{in}\ P$	(import)
$E ::=$	**expression**		
$\cdots$	as in Table 1		
$r\texttt{\#}m$	(service operation)		

The extended syntax uses the countably infinite sets of *operation names*, ranged over by m, n, ... Among variables we distinguish *services* ranged over by r, s, ... In the new syntax, u and v range over channels and expressions $r\texttt{\#}m$.

The schema $\{m_i : S_i{}^{i \in I}\}$, with I finite, describes services that offer a set of operations m_i whose schema is S_i. Operation names in records are pairwise different; the schemas S_i are always channel schemas of shape $\langle S \rangle^\kappa$ or $S \to T$. The definition of handle and the subschema relation of Definition 1 are extended with a further entry dealing with record schemas. Let $\{m_i : S_i{}^{i \in I}\} \downarrow \{m_i : S_i{}^{i \in I}\}\texttt{,()}$. A subschema $\mathcal{R}$ is a relation such that $S\ \mathcal{R}\ T$ implies the items listed in Definition 1 and, in addition:

(5) $S \downarrow \{m_i : S_i{}^{i \in I}\}\texttt{,}S'$ implies $T \downarrow \{m_j : T_j{}^{j \in J_k}\}\texttt{,}T'_k$, for $1 \leq k \leq n$, with $J_k \subseteq I$ and, for every $j \in J_k$, $S_j\ \mathcal{R}\ T_j$ and $S'\ \mathcal{R}\ \sum_{k \in 1..n} T'_k$.

For example $\{m : \langle \texttt{int} \rangle^{\texttt{O}}\, ; n : \langle \texttt{int} + \texttt{string} \rangle^{\texttt{O}}\}$ `<:` $\{n : \langle \texttt{int} \rangle^{\texttt{O}}\}$ and $\{m : \langle \texttt{int} \rangle^{\texttt{O}}\, ; n : \langle \texttt{string} \rangle^{\texttt{O}}\}\texttt{,(}\texttt{int} + \texttt{string}\texttt{)}$ `<:` $\{m : \langle \texttt{int} \rangle^{\texttt{O}}\}\texttt{,int} + \{n : \langle \texttt{string} \rangle^{\texttt{O}}\}\texttt{,string}$.

The process $\texttt{new}\ r : \{m_i : S_i{}^{i \in I}\}\ \texttt{in}\ P$ creates a service r exposing the operations m_i, $i \in I$. The continuation P addresses such operations with $r\texttt{\#}m_i$. In particular, since now u and v also range over expressions of the form $r\#m$, outputs, selects, and replications may also have the shape $r\texttt{\#}m\texttt{!}(E)$,

Table 8
Typing rules with service extensions.

Expressions :

$$\frac{\Gamma \vdash r : \{m_i : S_i{}^{i \in I}\} \qquad k \in I}{\Gamma \vdash r\texttt{\#}m_k : S_k}$$

Processes :

(NEW-S)

$$\frac{\Gamma + r : \{m_i : S_i{}^{i \in I}\}\,;\, \Delta + r : \{m_i : [S_i]_{\texttt{IO}}{}^{i \in I}\} \vdash P}{\Gamma\,;\, \Delta \vdash \texttt{new}\; r : \{m_i : S_i{}^{i \in I}\}\; \texttt{in}\; P}$$

(IMPORT-S)

$$\frac{(\Gamma\,;\,\Delta) + r : \{m_i : S_i{}^{i \in I}\} \vdash P \qquad \Gamma(v) \texttt{ <: } \{m_i : S_i{}^{i \in I}\}}{\Gamma\,;\, \Delta \vdash \texttt{import}\; r : \{m_i : S_i{}^{i \in I}\} = v\; \texttt{in}\; P}$$

`select` $\{r_j\texttt{\#}m_j\texttt{?}(F_j)\ P_j\ {}^{j \in J}\}$, and $r\texttt{\#}m\texttt{?*}(F)\ P$, respectively. The relevant upshot for the implementation of `PiDuce` is that only one `WSDL` resource is published and associated with the service r.

The process `import` $r : \{m_i : S_i{}^{i \in I}\} = v$ `in` P imports the service whose `WSDL` interface is located at v. This operation is successful provided that the schema of v contains *at least* the operations m_i, and that the schema constraints are satisfied as described in Section 6.

The type system of Table 2 is also extended in order to cope with records. The extension is detailed in Table 8. The operation $[S]_{\texttt{IO}}$ is defined as follows:

$$[S]_{\texttt{IO}} \;=\; \begin{cases} \langle T \rangle^{\texttt{IO}} & \text{if } S = \langle T \rangle^{\kappa} \\ \langle T\,\texttt{,}\,\langle R \rangle^{\texttt{O}} \rangle^{\texttt{IO}} & \text{if } S = T \to R \end{cases}$$

The new rules (NEW-S) and (IMPORT-S) generalize (NEW) and (IMPORT) to references that are services. Theorem 6 and Theorem 7 still hold for this extension.

8 `PiDuce` architecture and interoperability

`PiDuce` runtime environments consist of three components: the *virtual machine*, the *channel manager*, and the *Web interface* – see Figure 3.

The *virtual machine* executes threads by interpreting `PiDuce` object code resulting from the compilation of `PiDuce` programs. The implementation of the

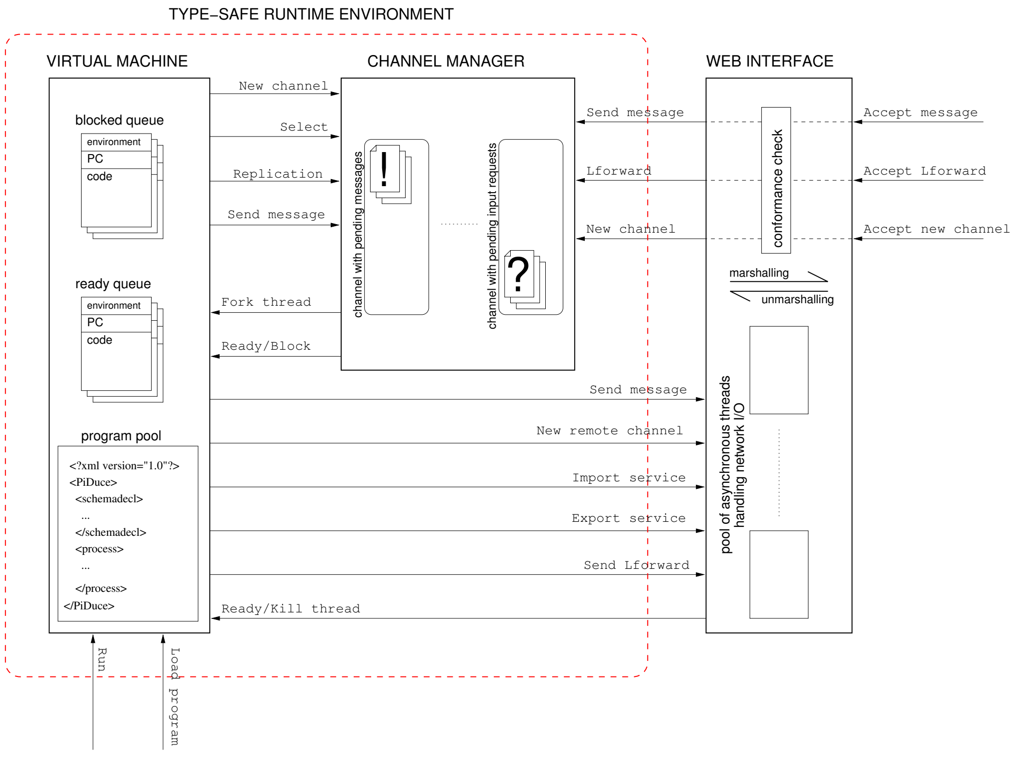

Fig. 3. `PiDuce`: the runtime environment.

virtual machine is standard. The virtual machine stores its data in three structures: the *program pool*, containing the object code of the processes that have been loaded; the *ready queue*, containing threads that are ready to execute; the *blocked queue*, containing threads awaiting for some message. Threads are executed by means of a round-robin scheduler.

The *channel manager* handles the pool of channels that are local to the runtime environment. It is thus responsible for any operation involving local channels, in particular creation, send, and receive operations. Within the channel manager, each channel consists of a *schema*, describing the values that are carried, a *message queue* containing all the messages that have been sent but not consumed, and a *request queue* containing the threads waiting for a message on that channel. Whenever a new message arrives, the first thread in the request queue, if any, is *awakened*; otherwise the message is moved into the message queue.

The `PiDuce` runtime environment interacts with the external environment through a *Web interface*, which is responsible for bridging `PiDuce` processes and standard Web service technologies. In the outgoing direction, the Web interface is responsible for publishing appropriate `WSDL` resources for the `PiDuce` services created by the local virtual machine, for exporting `PiDuce` schemas into corresponding `XML`-Schemas, and for marshalling `PiDuce` values into `XML` messages. In the incoming direction, the Web interface is responsible for importing `WSDL` resources as `PiDuce` services, for decoding `XML`-Schemas into `PiDuce` schemas, and for unmarshalling incoming `XML` messages into `PiDuce` values. Additionally, incoming `XML` messages are checked to be conformant to the schema of the channels they are targeted to, so as to prevent runtime errors within the virtual machine. The Web interface is also responsible for handling request-response channels and services as described in Section 7, so that within the virtual machine communication is purely asynchronous, whereas externally request-response services are handled in the standard way.

The modular design of this architecture has four main consequences: (1) the channel manager and the Web interface may be used stand-alone for providing `PiDuce`-compatible communication primitives in (native) programs that are written in a language other than `PiDuce`; (2) the virtual machine and the channel manager are decoupled from the actual transport protocols and technologies used in distributed communication. In this way a large part of `PiDuce` may be adapted to different contexts with minimum effort; (3) communications occurring within the same runtime environment are short-circuited and do not entail any additional overhead because they solely rely on internal data structures, rather than passing through the Web interface; (4) the virtual machine and the channel manager realize a type-safe environment: every operation performed therein can never manifest a type error.

8.1 Mechanisms interfacing `PiDuce` *channels and Web services*

Web services are published by interfaces that are written in a standard format: the `WSDL` – Web Service Description Language [25]. Every `WSDL` interface contains two parts: the *abstract* part defines the set of operations supported by the service; the *concrete* part binds every operation to a concrete network protocol and to a concrete location. Every operation is described by a name and by the schema of the *messages* that the operation accepts and/or produces. Albeit `WSDL` does not make any commitment on the schema language to be used, `XML`-Schema is the schema language universally adopted. Operations have an associated interaction pattern that conforms to one out of four models: *one-way interaction* (the client invokes a service by sending a message); *notification* (the service sends the message); *request-response* (the client sends a message and waits for the response); *solicit-response* (the service makes a request and waits for the response).

We discuss the possible `WSDL` interfaces by analyzing a number of examples. Consider the process `new` $u : \langle S \rangle^{\kappa}$ `in` P. This process creates a channel u and publishes it in a `WSDL` interface whose abstract part is:

```
<schema>
  <complexType name="InSchema">⟦ S ⟧</complexType>
</schema>
<message name="Input">
  <part name="par" type="InSchema"/>
</message>
<portType name="service">
  <operation name="operation" piduce:operationCapability="κ">
    <input message="Input"/>
  </operation>
</portType>
```

where $[\![S]\!]$ is the XML-Schema encoding of the PiDuce schema S (see Section 8.2.) This operation, being one-way, defines the "Input" message only and its schema "InSchema". It is worth to notice the use of the nonstandard attribute piduce:operationCapability which informs PiDuce clients that the service may support remote inputs if $\kappa \leq \mathtt{I}$, as such information cannot be inferred from the WSDL interface. Since the attribute is in the piduce namespace, it will be ignored by standard Web services. The concrete part of the WSDL interface for u is specified by two elements, binding and service:

```
1  <binding name="serviceSoap" type="service">
2    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
3    <operation name="operation">
4      <soap:operation style="document"
5        soapAction="http://www.cs.unibo.it:1811/x" />
6      <input><soap:body use="literal"/></input>
7    </operation>
8  </binding>
```

The element binding defines the concrete message formats and the protocols to be used for accessing the operation. Currently, PiDuce supports the SOAP-over-HTTP binding – see line 2 of the above document. When using the SOAP-over-HTTP binding, the Web interface communicates SOAP messages (XML documents with the shape Envelope[Header[*headers*], Body[*parameters*]] where the Header is optional) using the HTTP protocol. The soap:operation element on line 4 has two attributes: style specifies that the operation style is document (the current prototype supports also the RPC style); soapAction specifies the SOAPAction header used in the HTTP request. The information in these two attributes, together with the attribute use of the soap element, specifies the format of the XML message to be sent. When the attribute use is literal then the transported XML message appears directly under the SOAP Body element without any additional encoding information. When the attribute use is encoded then the XML message is annotated with additional schema information. Therefore a possible SOAP message for invoking a service having schema $\langle \mathtt{a}[\mathtt{int}] + \mathtt{b}[\mathtt{string}] \rangle^{\mathtt{O}}$ is

```
  <?xml version="1.0" encoding="utf-8"?>
```

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <a>1</a>
  </env:Body>
</env:Envelope>
```

The element `service` connects a binding to a specific `URL`. This `URL` is given by the location of the `PiDuce` runtime environment followed by a unique path, which is typically formed by appending the `?wsdl` suffix to the name of the channel. For instance, the following `service` element asserts that the service is located at `http://www.cs.unibo.it:1811/u`:

```
<service name="service">
  <port name="service" binding="serviceSoap">
    <soap:address location="http://www.cs.unibo.it:1811/u" />
  </port>
</service>
```

In addition to defining new channels, `PiDuce` also permits to import externally defined services. The process `import` $u : S =$ `URL in` P imports a one-way interaction service located at `URL` and gives it the name u. When the bytecode corresponding to the import process is loaded into the virtual machine, the `XML`-Schema of the service u is extracted from the `WSDL` located at `URL`, it is decoded into a `PiDuce` schema T, and the decoded schema is verified to be compatible with S following rule (IMPORT-A). If the attribute `piduce:operationCapability="`κ`"` is found in the `WSDL` (implying that u has been published by a `PiDuce` runtime), compatibility means S `<:` $\langle T\rangle^{\kappa}$. Otherwise compatibility means S `<:` $\langle T\rangle^{\mathtt{O}}$. The Web interface also verifies whether the binding is SOAP over HTTP. In case of success the value of the attribute `location` in the `service` element is used as target for future invocations. In case of failure of any of the above checks, the continuation P is not executed.

When the externally defined service is request-response, it may be imported by `import` $u : S \rightarrow T =$ `URL in` P. The schema of u is retrieved as before but, in this case, the `WSDL` interface has a `portType` element whose shape is:

```
<portType name="op-request-response">
  <operation name="request-response">
    <input message="Input"/>
    <output message="Output"/>
  </operation>
</portType>
```

The Web interface decodes `Input` and `Output` into the schemas $S_{\mathtt{I}}$ and $S_{\mathtt{O}}$, respectively. Then it verifies that $\langle S, \langle T\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}$ `<:` $\langle S_{\mathtt{I}}, \langle S_{\mathtt{O}}\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}$. The remaining behavior is similar to the previous case.

The correspondence between `PiDuce` schemas and `XML`-Schema is established by suitable encoding and decoding procedures implemented by the Web interface. By encoding we mean the translation of `PiDuce` schemas into `XML`-Schema, and by decoding we mean the inverse transformation.

Although `PiDuce` schemas and `XML`-Schema have a significant common intersection, there are features of `XML`-Schema not supported by `PiDuce` schemas and, conversely, features of `PiDuce` schemas that cannot be represented in `XML`-Schema. Regarding `XML`-Schema and the decoding function:

- `XML` attributes have been ignored because they would have entangled `PiDuce` schemas without giving any substantial contribution to their semantic relevance;
- features such as keys, references, and facets have been ignored because they are used mainly for validation (verifying that a value belongs to a given schema) rather than for typechecking (verifying that two schemas are related by the subschema relation).

In both cases such features are unused in the description of existing Web services (in particular in the `XML`-Schemas of the exchanged messages), hence omitting their treatment does not impede actual experimentation with `PiDuce`. For this subset of `XML`-Schema the decoding into `PiDuce` schemas is mostly straightforward. The only problematic case is for the `all` particle of `XML`-Schema that is used for defining sequences where elements can appear in any order. In this case the naive decoding into a `PiDuce` schema would result in a schema having an exponential size with respect to the number of elements occurring in the `all` particle. To alleviate this problem `all` is decoded as a single `PiDuce` sequence where elements are canonically ordered. When a value is received and validated by the Web interface against a `PiDuce` sequence originated by an `all` particle, the elements of the value are rearranged with the canonical order.

As regards the encoding function, `PiDuce` schemas that have a natural representation in `XML`-Schema are encoded by using standard elements in the `XML`-Schema namespace. The remaining `PiDuce` schemas are encoded using extension elements in a dedicated `PiDuce` namespace. In particular, extension elements are used for

- *channel schemas*, because current technologies do not provide any standard representation and description of them. We expect that this lack of expressiveness will be remedied in the near future, if higher-order Web services will prove to be a valuable feature;
- *schema names*, when these names are not the content of labelled values,

because such limitation of XML-Schema finds no justification in PiDuce schemas;
- *unions and differences of labels*, because these operations have been introduced in PiDuce mostly for pattern matching rather than for typing. In this case the lack of corresponding constructs in XML-Schema must not be interpreted as a weakness in XML-Schema itself. In fact, standard query and pattern languages such as XPath [13] and XQuery [8] provide for label wildcards.

It is understood that any WSDL interface containing schemas with extension elements will not be compatible with standard Web services.

9 Related work

The PiDuce prototype falls within the domain of distributed abstract machines for pi-like calculi. Among them we recall Facile [20], the Jocaml prototype [15], Distributed pi calculus [3], Nomadic Pict [32], the Ambient Calculus [10,30]. The differences between our model and the other ones are as follows. Facile uses two classes of distributed entities: (co-)located processes which execute, and channel-managers which mediate interaction. This forces it to use a handshake discipline for communication. Jocaml simplifies the model by combining input processes with channel-managers. However, it uses a quite different form of interaction, which does not relate that closely to pi calculus communication. It also forces a coarser granularity, in which every channel must be co-located with at least another one. Unlike Jocaml, our machine has finer granularity and uses the same form of interaction as the pi calculus. The other models add explicit location constructs to the pi calculus and use agent migrations for remote interactions.

PiDuce's type system has been strongly influenced by the one in XDuce, a functional language for XML processing [21]. In XDuce, values do not carry channels, and the subschema relation is never needed at run-time. Our paper may also be read as an investigation of the extension of XDuce values and schemas with channels.

Several integrations of processes and semi-structured data have been studied in recent years. Two similar contributions, that are contemporary and independent to this one, are [12] and [2]. The schema language in [12] is the one of [6] enriched with the channel constructors for input, output, and input-output capability. No apparent restriction to reduce the computational complexity of pattern matching is proposed and no prototyping effort is undertaken. The schema language of [2] is simpler than that of PiDuce. In particular recursion is omitted and labeled schemas have singleton labels.

Other contributions integrating semi-structured data and processes are discussed in order. `TulaFale` [7], a process language with `XML` data, is especially designed to address Web services security issues such as vulnerability to `XML` rewriting attacks. The language has no static semantics. The integration of `PiDuce` with the security features of `TulaFale` seems a promising direction of research. X$d\pi$ [19] is a language that supports dynamic Web page programming. This language is basically pi calculus with locations enriched with explicit primitives for process migration, for updating data, and for running a script. The emphasis of X$d\pi$ is towards behavioral equivalences and analysis techniques for behavioral properties. A contribution similar to [19] is Active `XML` [1] that uses an underlying model consisting of a set of peer locations with data and services.

10 Conclusions

In this contribution we have presented the `PiDuce` project, a distributed implementation of the asynchronous pi calculus with tree-structured datatypes and pattern matching. The resulting language incorporates constructs that are suitable for modeling Web services, and this motivates our choice of `XML` idioms, such as `XML`-Schema and `WSDL` for types and interfaces, respectively. In this respect, `PiDuce` fills the gap between theory and practice by formally defining a programming language and showing its implementation using industrial standards.

Figure 4 shows a `PiDuce` client that interacts with the Web services provided by the on-line store Amazon and the Google serach engine. The code shown is actually an approximation of the actual client, which needs a long preamble of complex schema definitions and slightly more complex messages to be sent to the two services; the full example can be found in the latest `PiDuce` distribution. The client starts by defining the relevant schemas that are published in the `WSDL`'s of the two Web services (lines 1 and 2). In fact, `PiDuce` provides an utility for extracting such declarations automatically, given the `URL` of the service's `WSDL` file. Lines 3 to 9 import the two Web services. For each service we only import the relevant operations. In this case they are both request-response operations, as can be noticed by the arrow schema. The `URL`'s after the keyword `location` refer to the `WSDL` files provided by Amazon and Google. Line 10 defines a local channel to be used as the continuation for the interaction with the Amazon Web service. While `PiDuce`'s Web interface interoperates natively with request-response operations, the language only provides for asynchronous communication primitives. Adding syntactic sugar for invoking synchronous operations is trivial. Line 11 defines a special channel used to write values on the terminal, so that the process can be monitored and the results can be printed. Lines 12 to 15 invoke Amazon by searching for a partic-

```
1   schema ProductInfo = ...
2   and GoogleSearchResult = ...
3   in import Amazon {
4     KeywordSearchRequest
5       : KeywordSearchRequest[KeywordRequest] → return[ProductInfo]
6   } location="http://soap.amazon.com/schemas2/AmazonWebServices.wsdl"
7   in import Google {
8    doGoogleSearch : q[string] → return[GoogleSearchResult]
9   } location="http://api.google.com/GoogleSearch.wsdl"
10  in new amazonReply { get : ⟨return[ProductInfo]⟩ }
11  in new stdout { print : ⟨Any⟩ } location="stdout"
12  in spawn {
13    Amazon.KeywordSearchRequest!
14      (KeywordSearchRequest[keyword["Nocturama"]], amazonReply.get)
15  }
16  amazonReply.get?(return[product : ProductInfo])
17    match product with {
18      Any, Details[Item[Any, Artists[artistList : Item[string]*],
19                         Any], Any] ⇒
20        match artistList with {
21          () ⇒ stdout.print! "no artist found"
22        | Item[name : string], Any ⇒
23            Google.doGoogleSearch!(q[name], stdout.print)
24        }
25    | Any => stdout.print!("no product or artist found")
26    }
```

Fig. 4. A PiDuce client interacting with both Amazon and Google Web services.

ular keyword, and the process starting on line 16 waits for the response. Once this arrives, a query is done on the received document (lines 17 to 22) and one piece of extracted information is used to start the Google search engine on line 23. The result is directly printed on the terminal.

Regarding the description of Web services interfaces, it is remarkable that WSDL 1.1 (already published as a W3C Note [25]) does not consider service references as first class values, that is natural in a distributed setting, in pi calculus, and, thereafter, in PiDuce. This lack of expressiveness has been at least partly amended in WSDL 2.0 [26,27] that, at the time of this writing, is in a Candidate Recommendation status. Still, we note significant differences between our approach and the way "Web services as values" are handled in WSDL 2.0. For example the client receiving a service reference u must eventually compare the schema in the WSDL of u with some local schema before using u or forwarding u to a third party. While this comparison, called subschema relation in this paper, is fundamental in PiDuce, it has been completely overlooked in WSDL 2.0.

Few remarks about XML-Schema are in order. First of all there is a large overlapping between XML-Schema and PiDuce schemas, which has been discussed in Section 8. Apart from channel schemas, the other major departure from XML schema is the support for nondeterministic labelled schemas. These schemas make the computational complexity of the subschema relation exponential, but they are essential for the static semantics of a basic operator in PiDuce, the pattern-matching (see the third premise of rule (MATCH) in Table 2). Noticeably, the constraint of label-determinedness on channel schemas guarantees a polynomial cost for the subschema relation (and for the pattern matching) at runtime (see Appendix C).

Future work in the PiDuce project is planned in two directions: the first direction is rather pragmatic, and is aimed to improving interoperability and support to existing protocols. The goal is to interface PiDuce with more real-world Web services and to carry on more advance experimentation. The other direction regards conceptual features that are desirable and that cannot be expressed conveniently in the current model. In particular error handling and transactional mechanisms. These mechanisms, which are basic in BPEL [5], permit the coordination of processes located on different machines by means of time constraints. This is a well-known problematic issue in concurrency theory. An initial investigation about transactions in the setting of the asynchronous pi calculus has been undertaken in [23]. A core BPEL language without such advanced coordination mechanisms should be compilable in PiDuce without much effort, thus equipping BPEL with a powerful static semantics. We expect to define a translation in the near future.

Another direction of research is about dynamic XML data, namely those data containing active parts that may be executed on clients' machines. This is obtained by transmitting processes during communications, a feature called process migration. The PiDuce prototype disallows program deployments on the network. However, the step towards migration is quite short due to the fact that object code is in XML format. Therefore it suffices to introduce two new schemas: the object code schema and the environment schema, and admit channels carrying messages of such schemas.

References

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and Web services integration. In *Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases (VLDP 2002), Hong Kong SAR, China*, pages 1087–1090. Morgan Kaufmann Publishers, 2002.

[2] L. Acciai and M. Boreale. XPi: a typed process calculus for XML messaging. In *7th Formal Methods for Object-Based Distributed Systems (FMOODS'05)*,

volume 3535 of *Lecture Notes in Computer Science*, pages 47 – 66. Springer-Verlag, 2005.

[3] R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus (extended abstract). In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1999)*, volume 1738 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 1999.

[4] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[5] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services, 2003. Available at `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`.

[6] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, pages 51–63, New York, 2003. ACM Press.

[7] K. Bhargavan, C. Fournet, A. Gordon, and R. Pucella. TulaFale: A Security Tool for Web Services. In *Proceedings of the 2nd International Symposium on Formal Methods for Components and Objects (FMCS'03)*, volume 3188 of *LNCS*, pages 197–222. Springer-Verlag, 2004.

[8] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon (eds). XQuery 1.0: An XML query language, W3C Candidate Recommendation, June 2006. `http://www.w3.org/TR/xquery/`.

[9] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In R. Hindley, editor, *3rd International Conference on Typed Lambda Calculi and Application (TLCA), Nancy, France*, volume 1210 of *Lecture Notes in Computer Science*, pages 63 – 81. Springer-Verlag, April 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309-338, 1998.

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

[11] S. Carpineti and C. Laneve. A basic contract language for Web services. In *Proceedings of the European Symposium on Programming (ESOP 2006)*, volume 3924 of *LNCS*, pages 197–213. Springer-Verlag, 2006.

[12] G. Castagna, R. D. Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE Computer Society, 2005.

[13] J. Clark and S. DeRose (eds). XML Path Language (XPath) Version 1.0, W3C Recommendation. Available at `http://www.w3c.org/TR/xpath/`, Nov. 1999.

[14] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata, 1997. released October, 1st 2002.

[15] S. Conchon and F. L. Fessant. Jocaml: Mobile agents for objective-caml. In *First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, pages 22–29. IEEE Computer Society Press, October, 1999.

[16] C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. In *Proceedings of 25th Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *LNCS*, pages 844–855. Springer-Verlag, 1998.

[17] A. Frisch and L. Cardelli. Greedy regular expression matching. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2004.

[18] P. Gardner, C. Laneve, and L. Wischik. Linear forwarders. In *14th International Conference on Concurrency Theory (CONCUR 2003)*, volume 2761 of *Lecture Notes in Computer Science*, pages 415 – 430. Springer-Verlag, 2002.

[19] P. Gardner and S. Maffeis. Modelling dynamic web data. *Theoretical Computer Science*, 342:104–131, 2005.

[20] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[21] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[22] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[23] C. Laneve and G. Zavattaro. Foundations of Web Transactions. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS'05)*, volume 3441 of *LNCS*, pages 282–298. Springer-Verlag, 2005.

[24] Microsoft Corporation. Biztalk server. http://www.microsoft.com/biztalk/.

[25] Web Services Description Working Group. Web Services Description Language (WSDL) 1.1). Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315. W3C Note 15 March 2001.

[26] Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. Available at http://www.w3.org/TR/2005/WD-wsdl20-primer-20050803/. W3C Working Draft 3 August 2005.

[27] Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Available at http://www.w3.org/TR/2005/WD-wsdl20-20050803/. W3C Working Draft 3 August 2005.

[28] XML Core Working Group. Extensible Markup Language (XML) 1.1. Available at `http://www.w3.org/TR/2004/REC-xml11-20040204/`. 04 February 2004, edited in place 15 April 2004.

[29] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.

[30] A. Phillips, N. Yoshida, and S. Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proceedings of the European Symposium on Programming (ESOP 2004)*, LNCS, pages 155–170. Springer-Verlag, Apr. 2004.

[31] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.

[32] P. Sewell, P. Wojciechowski, and B. Pierce. Location independence for mobile agents. In H. E. Bal, B. Belkhouche, and L. Cardelli, editors, *ICCL 1998*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer-Verlag, 1999.

[33] S. Thatte. XLANG: Web services for business process design. Available at `www.gotdotnet.com/team/xml\_wsspecs/xlang-c/default.htm`. Microsoft Corporation, 2001.

[34] S. Vansummeren. Unique pattern matching in strings. *CoRR*, cs.PL/0302004, 2003.

A Properties of the subschema relation

This appendix contains the proofs of Proposition 2. The statement is recalled for readability sake.

Proposition 2 *(1)* $\texttt{<:}$ *is reflexive and transitive;*
(2) If S *is empty, then* $S \texttt{ <: } \texttt{Empty}$*;*
(3) (Contravariance of $\langle\cdot\rangle^{\texttt{O}}$*)* $S \texttt{ <: } T$ *if and only if* $\langle T\rangle^{\texttt{O}} \texttt{ <: } \langle S\rangle^{\texttt{O}}$*;*
(4) (Covariance of $\langle\cdot\rangle^{\texttt{I}}$*)* $S \texttt{ <: } T$ *if and only if* $\langle S\rangle^{\texttt{I}} \texttt{ <: } \langle T\rangle^{\texttt{I}}$*;*
(5) (Invariance of $\langle\cdot\rangle^{\texttt{IO}}$*)* $S \texttt{ <: } T$ *and* $T \texttt{ <: } S$ *if and only if* $\langle S\rangle^{\texttt{IO}} \texttt{ <: } \langle T\rangle^{\texttt{IO}}$*;*
(6) If $S \texttt{ <: } T$*, then* $S\texttt{,()} \texttt{ <: } T$*; if* $\texttt{(),}S \texttt{ <: } T$*, then* $S \texttt{ <: } T$*;*
(7) If $S \texttt{ <: } T$ *and* $S' \texttt{ <: } T'$*, then* $S\texttt{,}S' \texttt{ <: } T\texttt{,}T'$*;*
(8) If $(S + S')\texttt{,}S'' \texttt{ <: } T$*, then* $S\texttt{,}S'' \texttt{ <: } T$ *and* $S'\texttt{,}S'' \texttt{ <: } T$*;*
(9) For every S*,* $\texttt{Empty} \texttt{ <: } S \texttt{ <: } \texttt{Any}$ *and* $\langle S\rangle^{\kappa} \texttt{ <: } \texttt{AnyChan}$ *and* $\langle\texttt{Any}\rangle^{\texttt{IO}} \texttt{ <: } \langle S\rangle^{\texttt{O}}$ *and* $\langle\texttt{Empty}\rangle^{\texttt{IO}} \texttt{ <: } \langle S\rangle^{\texttt{I}}$*.*

Proof: We prove items 1 (transitivity), 7, and 9.

As regards transitivity of item 1, let $\mathcal{R}$ be a subschema relation and let $\mathcal{R}^+$ be the least relation that contains $\mathcal{R}$ and is closed under the following operations

(1) if $S \, \mathcal{R}^+ \, T$ then $S \, \mathcal{R}^+ \, T + R$;
(2) if $S \, \mathcal{R}^+ \, T$ and $S' \, \mathcal{R}^+ \, T$ then $S + S' \, \mathcal{R}^+ \, T$;
(3) if $S \, \mathcal{R}^+ \, T$ and $S \downarrow L[S']\,,S''$ then $L'[S']\,,S'' \, \mathcal{R}^+ \, T$ with $\widehat{L'} \subseteq \widehat{L}$;

It is easy to verify that $\mathcal{R}^+$ is a subschema relation. Let $\mathcal{R}$ and $\mathcal{S}$ be two subschema relations such that $S \, \mathcal{R} \, T$ and $T \, \mathcal{S} \, R$. We prove that

$$\mathcal{T} = \{(S, R) \quad | \quad S \, \mathcal{R}^+ \, T \text{ and } T \mathcal{S}^+ R\}$$

is a subschema relation. Let $S \; \mathcal{T} \; R$. The critical case is when $S \downarrow L[S']\,,S''$. According to the definition of $\mathcal{T}$, there exists T such that $S \, \mathcal{R}^+ \, T$ and $T \mathcal{S}^+ R$. By Definition 1, $T \downarrow L'[T']\,,T''$ with $\widehat{L} \cap \widehat{L'} \neq \emptyset$. There are two cases:

(a) $T \downarrow L'[T']\,,T''$ with $\widehat{L} \not\subseteq \widehat{L'}$ and $\widehat{L} \cap \widehat{L'} \neq \emptyset$. We are reduced to $(L \cap L')[S']\,,S'' \; \mathcal{T} \; R$ and $(L \setminus L')[S']\,,S'' \; \mathcal{T} \; R$, which are immediate by definition of $\mathcal{T}$.
(b) $T \downarrow L_i[T_i']\,,T_i''$ with $i \in I$ and $\widehat{L} \subseteq \bigcap_{i \in I} \widehat{L_i}$ and, for every $K \subseteq I$:

$$\text{either} \quad S' \mathcal{R} \sum_{k \in K} T_k' \quad \text{or} \quad S'' \mathcal{R} \sum_{k \in I \setminus K} T_k'' \; . \tag{A.1}$$

There are two subcases:
(b1) $R \downarrow M[R']\,,R''$ with $\widehat{L} \cap \widehat{M} \neq \emptyset$ and $\widehat{L} \not\subseteq \widehat{M}$. In this case we must prove $(L \cap M)[S']\,,S'' \; \mathcal{T} \; R$ and $(L \setminus M)[S']\,,S'' \; \mathcal{T} \; R$, which are immediate by definition of $\mathcal{T}$.
(b2) $R \downarrow M_j[R_j']\,,R_j''$ with $j \in J$ and $\widehat{L} \subseteq \bigcap_{j \in J} \widehat{M_j}$. There are again two subcases: (b2.1) there are i, k such that $\widehat{L_i} \not\subseteq \widehat{M_k}$; (b2.2) the contrary of (b2.1). In case (b2.1) we apply the simulation case 4.(a): it must be $(L_i \cap M_k)[T_i']\,,T_i'' \; \mathcal{S} \; R$ and $(L_i \setminus M_k)[T_i']\,,T_i'' \; \mathcal{S} \; R$. As far as $(L_i \cap M_k)[T_i']\,,T_i'' \; \mathcal{S} \; R$ is concerned, $\widehat{L} \subseteq \widehat{L_i \cap M_k}$. If $L_i \cap M_k$ is not contained in every M_j we reiterate the argument (b2.1) on the schema $(L_i \cap M_k)[T_i']\,,T_i''$. We end up with a set of schemas $L_i'[T_i']\,,T_i''$ with $i \in I$ such that $L_i'[T_i']\,,T_i'' \; \mathcal{S} \; R$ and the case (b2.2) holds. From now on the arguments of the two cases are the same. We let $L_i' = L_i$. From $L_i[T_i']\,,T_i'' \; \mathcal{S} \; R$ we have: for every $K' \subseteq J$:

$$\text{either} \quad T_i' \, \mathcal{S} \sum_{k \in K'} R_k' \quad \text{or} \quad T_i'' \, \mathcal{S} \sum_{k \in J \setminus K'} R_k'' \tag{A.2}$$

Let $K \subseteq J$. Since $\widehat{L} \subseteq \bigcap_{j \in J} \widehat{M_j}$, we must prove:

$$\text{either} \quad S' \; \mathcal{T} \sum_{k \in K} R_k' \qquad \text{or} \qquad S'' \; \mathcal{T} \sum_{k \in J \setminus K} R_k'' \tag{A.3}$$

For every $i \in I$, the constraint (A.2) implies

$$\text{either} \quad T_i' \; \mathcal{S}^+ \sum_{k \in J} R_k' \quad \text{or} \quad T_i'' \; \mathcal{S}^+ \sum_{k \in J \setminus K} R_j'' \tag{A.4}$$

where the relation is $\mathcal{S}^+$. Let $H_K = \{h \in I \mid T'_h\ \mathcal{S}^+\ \sum_{k\in K} R'_k\}$. By definition $H_K \subseteq I$ and $T''_{h'}\ \mathcal{S}^+\ \sum_{k\in J\setminus K} R''_k$ for every $h' \in I \setminus H_K$. The constraint (A.1) implies

$$\text{either}\quad S'\ \mathcal{R}^+ \sum_{h\in H_K} T'_h \quad\text{or}\quad S''\ \mathcal{R}^+ \sum_{h\in I\setminus H_K} T''_h \tag{A.5}$$

The constraint (A.3) follows from (A.5) and (A.4).

The case (b2.2) is similar to (b2.1) but we apply the simulation case 4.(b)

As regards the item 7, let $\mathcal{R}$ be a subschema relation such that $S\ \mathcal{R}\ T$ and $S'\ \mathcal{R}\ T'$. Let $\widehat{\mathcal{R}}$ be the least relation that contains $\mathcal{R}$ and that is closed under reflexivity and under the following operation:

- if $S\ \widehat{\mathcal{R}}\ T$, then $S\texttt{,}R\ \widehat{\mathcal{R}}\ T\texttt{,}R$ and $R\texttt{,}S\ \widehat{\mathcal{R}}\ R\texttt{,}T$.

The relation $\widehat{\mathcal{R}}$ is a subschema relation. We demonstrate the case $S\texttt{,}R\ \widehat{\mathcal{R}}\ T\texttt{,}R$ and omit the other one because trivial. Let $S\texttt{,}R \downarrow R'$. If $S \downarrow \texttt{()}$ and $R \downarrow R'$ then, by $S\ \widehat{\mathcal{R}}\ T$, we have $T \downarrow \texttt{()}$ and $T\texttt{,}R \downarrow R'$. We can conclude by reflexivity of $\widehat{\mathcal{R}}$. If $S \downarrow \texttt{B,}S'$, then $R' = \texttt{B,}S'\texttt{,}R$. From $S\ \widehat{\mathcal{R}}\ T$ we have that $T \downarrow \texttt{B}'_i\texttt{,}T'_i$ for $1 \le i \le n$, with $\texttt{B} \sqsubseteq \texttt{B}'_i$ and $S'\ \widehat{\mathcal{R}}\ \sum_{1\le i\le n} T'_i$. Hence $T\texttt{,}R \downarrow \texttt{B}'_i\texttt{,}T'_i\texttt{,}R$ for $1 \le i \le n$ and now $S'\texttt{,}R\ \widehat{\mathcal{R}}\ \sum_{1\le i\le n} T'_i\texttt{,}R$ by definition of $\widehat{\mathcal{R}}$. The remaining cases are similar. We conclude by remarking that $(S\texttt{,}S', T\texttt{,}T')$ is in the transitive closure of $\widehat{\mathcal{R}}$.

As regards the item 9, let $\mathcal{R}$ be the least relation containing the identity and the pairs:

$$(\texttt{Empty}, S), (S, \texttt{Any}), (\langle S\rangle^{\kappa}, \texttt{AnyChan}), (\langle\texttt{Any}\rangle^{\texttt{IO}}, \langle S\rangle^{\texttt{O}}), (\langle\texttt{Empty}\rangle^{\texttt{IO}}, \langle S\rangle^{\texttt{I}})$$
$$(S, (\texttt{int} + \texttt{string} + \texttt{AnyChan} + \texttt{\~{}[Any]})^*), (\texttt{n}, \texttt{int}), (\texttt{s}, \texttt{string})$$

The proof that $\mathcal{R}$ is a subschema relation is straightforward, except for the pairs $(S, \texttt{Any})$ and $(S, (\texttt{int} + \texttt{string} + \texttt{AnyChan} + \texttt{\~{}[Any]})^*)$. We analyze the first pair, the other being similar. We show that every R such that $S \downarrow R$ is simulated by $\texttt{Any}$. The interesting case is when $R = L[S']\texttt{,}S''$. In this case $\texttt{Any} \downarrow \texttt{\~{}[Any],}(\texttt{int} + \texttt{string} + \texttt{AnyChan} + \texttt{\~{}[Any]})^*$ and we are in case 4.b of Definition 1. Since $S \downarrow R$ then S is not-empty, similarly for $\texttt{Any}$. Therefore we are reduced to $(S', \texttt{Any}), (S'', (\texttt{int} + \texttt{string} + \texttt{AnyChan} + \texttt{\~{}[Any]})^*) \in \mathcal{R}$, which hold by definition. $\square$

B Soundness of the static semantics

The basic statements below are standard preliminary results for the subject reduction theorem.

Lemma 8 (Weakening) *(1) If $\Gamma \vdash E : S$ and $x \notin \mathtt{fv}(E)$, then $\Gamma + x : T \vdash E : S$;*
(2) If $\Gamma ; \Delta \vdash P$ and $x \notin \mathtt{fv}(P)$, then both (a) $\Gamma + x : S ; \Delta \vdash P$ and (b) $\Gamma + x : \langle S \rangle^{\kappa} ; \Delta + x : \langle S \rangle^{\mathtt{IO}} \vdash P$.

Actually, the premises of the second statement of Lemma 8 also entail $\Gamma + x : S ; \Delta + x : S \vdash P$, but this property is never used in the following. When a local channel is created, the property that is used is (b). A somewhat converse statement of weakening is the following.

Lemma 9 (Strengthening) *If $\Gamma \vdash E : S$ and $x \notin \mathtt{fv}(E)$, then $\Gamma \setminus x \vdash E : S$. Similarly, if $\Gamma ; \Delta \vdash P$ and $x \notin \mathtt{fv}(P)$, then $\Gamma \setminus x ; \Delta \setminus x \vdash P$.*

The following proposition collects properties about judgments of values. We recall that Γ is channeled when it binds variables to channel schemas.

Proposition 10 *Let $\Gamma \vdash V : S$.*

(1) If $S = L[S']\mathtt{,}S''$, then L is a singleton;
(2) If $S \mathrel{\mathtt{<:}} \langle T \rangle^{\kappa}$, then V is a variable;
(3) If Γ is channeled and $S \mathrel{\mathtt{<:}} T_1 + T_2$, then either $S \mathrel{\mathtt{<:}} T_1$ or $S \mathrel{\mathtt{<:}} T_2$;
(4) If $S \mathrel{\mathtt{<:}} T_1\mathtt{,}T_2$, then there exist V_1 and V_2 such that $V = V_1 @ V_2$ and $\Gamma \vdash V_1 : S_1$ and $\Gamma \vdash V_2 : S_2$ and $S_1 \mathrel{\mathtt{<:}} T_1$ and $S_2 \mathrel{\mathtt{<:}} T_2$;
(5) If and $S \mathrel{\mathtt{<:}} T^*$, then either $V = \mathtt{()}$ or $V = V_1 @ V_2$ with $V_1 \neq \mathtt{()}$ and $\Gamma \vdash V_1 : S_1$ and $\Gamma \vdash V_2 : S_2$ and $S_1 \mathrel{\mathtt{<:}} T$ and $S_2 \mathrel{\mathtt{<:}} T^*$.

Proof: Item (1) follows from the definition of judgment for expressions.

Item (2) follows from the definitions of values (a void expression or a sequence of non-void values) and of judgment for expressions.

Regarding item (3), we proceed by induction on the derivation of $\Gamma \vdash V : S$. The base case are:

- $S = \mathtt{()}$. By definition of $\mathtt{<:}$ we have either $T_1 \downarrow \mathtt{()}$ or $T_2 \downarrow \mathtt{()}$, then we conclude;
- $S = \mathtt{B}$. Since $S \downarrow \mathtt{B,()}$ we have three cases. If $S \mathrel{\mathtt{<:}} T_1$ or $S \mathrel{\mathtt{<:}} T_2$ we immediatly conclude. Otherwise, by definition of $\mathtt{<:}$, we obtain:

$$T_1 \downarrow \mathtt{B}_i\mathtt{,}Q_i \qquad \mathtt{B} \sqsubseteq \mathtt{B}_i \qquad 1 \leq i \leq n \tag{B.1}$$
$$T_2 \downarrow \mathtt{B}_j\mathtt{,}Q_j \qquad \mathtt{B} \sqsubseteq \mathtt{B}'_j \qquad n+1 \leq j \leq m \tag{B.2}$$
$$\mathtt{()} \;\mathtt{<:}\; \sum_{1 \leq i \leq m} Q_i \tag{B.3}$$

Since B.3 implies $Q_k \downarrow \mathtt{()}$ for some $k \in \{1, \dots, m\}$, we conclude S <: $\mathtt{B}_k\mathtt{,}Q_k$ by either B.1 or B.2.

- $S = \langle S' \rangle^\kappa$. Similar to the previous case.

The inductive cases are:

- $S = \mathtt{B,}S_1$. If S <: T_1 or S <: T_2 we immediately conclude. Otherwise, by definition of <:, we have $T_1 \downarrow \mathtt{B}_i\mathtt{,}Q_i$ with $\mathtt{B} \sqsubseteq \mathtt{B}_i$ for $1 \leq i \leq n$, and $T_2 \downarrow \mathtt{B}_j\mathtt{,}Q_j$ with $\mathtt{B} \sqsubseteq \mathtt{B}'_j$ for $n+1 \leq j \leq m$ and S_1 <: $\sum_{1 \leq i \leq m} Q_i$. We conclude by the inductive hypothesis.
- $S = \langle S_1 \rangle^\kappa\mathtt{,}S_2$. Similar to the previous case.
- $S = a[S_1]\mathtt{,}S_2$. If S <: T_1 or S <: T_2 we immediately conclude. Otherwise, by definition of <:, we have $T_1 \downarrow L_i[Q_i]\mathtt{,}Q'_i$ for $1 \leq i \leq n$, and $T_2 \downarrow L_j[Q_j]\mathtt{,}Q'_j$ for $n+1 \leq j \leq m$. Since a is a singleton (4).b of <: applies. We assume by contradiction that $a[S_1]\mathtt{,}S_2 \not<: L_i[Q_i]\mathtt{,}Q'_i$ for any $i \in \{1, \dots, m\}$ (i.e. $S_1 \not<: Q_i \vee S_2 \not<: Q'_i$ for any $i \in \{1, \dots, m\}$). Then we choose J_i as follows:
  (1) $J_1 = \emptyset$ implies S_2 <: $\sum_{i \in \{1,\dots,m\}} Q'_i$ and, by the inductive hypothesis, there exists k_1 such that S_2 <: Q'_{k_1};
  (2) $J_{k_1} = \{k_1\}$, since $S_1 \not<: Q_{k_1}$, we have S_2 <: $\sum_{i \in \{1,\dots,m\} \setminus \{k_1\}} Q'_i$ and, by the inductive hypothesis, there exists $k_2 \neq k_1$ such that S_2 <: Q'_{k_2};
  (3) $J_{k_1,k_2} = \{k_1, k_2\}$, since $S_1 \not<: Q_{k_1}$ and $S_1 \not<: Q_{k_2}$, by the inductive hypothesis we have $S_1 \not<: Q_{k_1}+Q_{k_2}$. Then we must have S_2 <: $\sum_{i \in \{1,\dots,m\} \setminus \{k_1,k_2\}} Q'_i$ that implies, by the inductive hypothesis, k_3 with $k_3 \neq k_1$ and $k_3 \neq k_2$ such that S_2 <: Q'_{k_3};
  $(\dots)$
  $(m+1)$ $J_{\{k_1,k_2,\dots,k_m\}} = \{1, \dots, m\}$ then we have to prove S_1 <: $\sum_{i \in \{1,\dots,m\}} Q_i$ that, by inductive hypothesis, implies S_1 <: Q_k for some $k \in \{1, \dots, m\}$. But this is not possible because of the previous m judgements ($S_1 \not<: Q_{k_1}$ for (1), $S_1 \not<: Q_{k_2}$ for (2), ..., $S_1 \not<: Q_{k_m}$).
  Therefore we obtain $a[S_1]\mathtt{,}S_2 \not<: T_1 + T_2$ which contradicts the hypothesis.
- If $S = a[S_1]$, since $S \downarrow a[S_1]\mathtt{,()}$ we reduce to the previous case.

Regarding item (4), we proceed by induction on V. For the base case assume that $T_1 \downarrow \mathtt{()}$ and S <: T_2 (notice that this case includes the one where $V = \mathtt{()}$). We conclude by taking $V_1 = \mathtt{()}$ and $V_2 = V$. For the inductive case assume that either $T_1 \downarrow R$ implies $R \neq \mathtt{()}$ or that $T_1 \downarrow \mathtt{()}$ and $S \not<: T_2$. We reason by cases on the structure of V, we only show the case when $V = \mathtt{b,}V'$, the others are similar. We have $S = \mathtt{b,}S'$ where $\Gamma \vdash V' : S'$. We must have $T_1 \downarrow \mathtt{B,}T'_1$ with $\mathtt{b}$ <: $\mathtt{B}$ and S' <: $T'_1\mathtt{,}T_2$. By induction hypothesis there exist V'_1 and V_2 such that $V' = V'_1@V_2$ and $\Gamma \vdash V'_1 : S'_1$ and $\Gamma \vdash V_2 : S_2$ and S'_1 <: T'_1 and

S_2 <: T_2. We conclude by taking $V_1 = \texttt{b,}V_1'$.

Regarding item (5), if $V = \texttt{()}$ we conclude immediately. Assume $V \neq \texttt{()}$. Then we must have $T \downarrow R$, with $R \neq \texttt{()}$ and S <: R,T^*. By item (4) we obtain $V = V_1@V_2$ and $\Gamma \vdash V_1 : S_1$ and $\Gamma \vdash V_2 : S_2$ and S_1 <: R and S_2 <: T^*. Since R is a handle and $R \neq \texttt{()}$ we must have $V_1 \neq \texttt{()}$. Furthermore, since R is a handle of T, we have R <: T hence we conclude S_1 <: T. □

Lemma 11 (Substitution) *Let V be a* $\texttt{dom}(\Gamma)$*-value and* $\Gamma \vdash V : S$.

(1) If $\Gamma \vdash E : T$, $\Gamma \vdash x : R$ *and* S <: R, *then* $\Gamma \vdash E\{^V/_x\} : T'$ *with* T' <: T.
(2) If $\Gamma ; \Delta \vdash P$, $\Gamma + \Delta \vdash x : R$ *and* S <: R, *then* $\Gamma ; \Delta \vdash P\{^V/_x\}$.

Proof: The proof is by induction on the structure of the derivations of $\Gamma \vdash E : T$ and $\Gamma ; \Delta \vdash P$.

For (1) we only discuss the case when E is a sequence E_1,E_2. By definition of $\vdash$, $\Gamma \vdash E_1 : T_1$ and $\Gamma \vdash E_2 : T_2$, and by inductive hypothesis we have

$$\Gamma \vdash E_1\{^V/_x\} : T_1' \qquad \text{and} \qquad T_1' \mathrel{<:} T_1 \tag{B.4}$$
$$\Gamma \vdash E_2\{^V/_x\} : T_2' \qquad \text{and} \qquad T_2' \mathrel{<:} T_2 \tag{B.5}$$

From (B.4), and (B.5) we obtain $\Gamma \vdash (E_1\texttt{,}E_2)\{^V/_x\} : T_1'\texttt{,}T_2'$. By Proposition 2(6), $T_1'\texttt{,}T_2'$ <: $T_1\texttt{,}T_2$ and we conclude.

For (2) we only discuss the case when the last rule is (OUT). Then $P = u\texttt{!}(E)$ and the premises of the rule are the judgments $\Gamma \vdash E : T$ and $\Gamma ; \Delta \vdash u : R$, and the predicate

$$R \mathrel{<:} \langle T \rangle^{\texttt{O}} \tag{B.6}$$

We must prove $\Gamma ; \Delta \vdash u\texttt{!}(E)\{^V/_x\}$. By $\Gamma \vdash E : T$, the hypothesis $\Gamma \vdash V : S$, S <: R, and the substitution lemma for expressions, we obtain

$$\Gamma \vdash E\{^V/_x\} : T' \tag{B.7}$$
$$T' \mathrel{<:} T \tag{B.8}$$

As regards the subject of the output, there are two subcases: (a) $x \neq u$ and (b) $x = u$. Case (a) follows by (B.6), (B.8), contravariance of $\langle\cdot\rangle^{\texttt{O}}$ and transitivity of <:. Case (b) implies $S = R$. Therefore, by Proposition 10, V is a variable. The lemma follows by (B.7), the hypotheses $\Gamma \vdash x : S$, the (B.6), the contravariance of $\langle\cdot\rangle^{\texttt{O}}$, and the transitivity of <:. □

The weakening, strengthening, and substitution lemmas entail a subsumption property that is useful for the correctness of the rule (DTR1) in the subject reduction.

Proposition 12 *If* $\Gamma + x : T\,;\Delta \vdash P$ *and* $x \notin \mathtt{dom}(\Delta)$ *and* S `<:` T *then* $\Gamma + x{:}S\,;\Delta \vdash P$*.*

In the rest of this appendix, we generalize all the functions defined over patterns to markers and to sequences of patterns and markers $\Phi = F_1 :: F_2 :: \cdots :: F_n$ where a marker is treated like the empty sequence `()` and a sequence $F_1 :: F_2 :: \cdots :: F_n$ is treated like the pattern F_1`,`F_2`,` $\ldots$ `,`F_n which reduces to `()` when $n = 0$. In particular, we generalize the functions $\mathtt{schof}(\cdot)$, $\mathtt{fv}(\cdot)$, $\mathtt{Env}(\cdot)$. The next two statements regard the soundness of the evaluation of expressions and of pattern matching. Straightforward proofs are omitted.

Lemma 13 (Evaluation) *Let* $\Gamma \vdash E : S$*. If* $E \Downarrow_{\mathtt{dom}(\Gamma)} V$*, then* $\Gamma \vdash V : T$ *and* T `<:` S*.*

Lemma 14 (Pattern Matching) *Let* $\Gamma \vdash V : S$ *and* $\Gamma \vdash V \in \Phi \leadsto \sigma$*.*

(1) S `<:` $\mathtt{schof}(\Phi)$*;*
(2) If $u \notin \mathtt{fv}(V)$*, then* $\Gamma + u : S \vdash V \in \Phi \leadsto \sigma$*;*
(3) for every $u \in \mathtt{fv}(\Phi)$*,* $\Gamma \vdash \sigma(u) : T$ *and* T `<:` $\mathtt{Env}(\Phi)(u)$*.*

Proof: items (1) and (2) are trivial. Regarding item (3), we proceed by induction on the proof tree of $\Gamma \vdash V \in \Phi \leadsto \sigma$. The only interesting case is when the last rule in the proof of $\Gamma \vdash V \in \Phi \leadsto \sigma$ is (PM7):

$$\frac{\Gamma \vdash V \in F :: x/V :: \Phi' \leadsto \sigma}{\Gamma \vdash V \in (x : F) :: \Phi' \leadsto \sigma}\ (\textsc{pm}7)$$

and take $u = x$. Eventually, in the proof tree of $\Delta \vdash V \in F :: x/V :: \Phi' \leadsto \sigma$, there will be an application of rule (PM5):

$$\frac{\Gamma \vdash V' \in \Phi' \leadsto \sigma' \quad V = V''@V'}{\Gamma \vdash V' \in x/V :: \Phi' \leadsto \sigma' + [x \mapsto V'']}\ (\textsc{pm}3)$$

By letting $\Phi' = [\ ]$ and $V' =$ `()` and $\sigma' = \emptyset$ we obtain a proof tree of $\Gamma \vdash V'' \in (x : F) \leadsto \sigma''$. From item (1) we derive that $\Gamma \vdash V'' : S$ implies S `<:` $\mathtt{schof}(F)$. We conclude by observing that $\mathtt{Env}(\Phi)(x) = \mathtt{schof}(F)$ and that $\sigma(x) = V''$. $\square$

Every preliminary is set for the subject reduction. For readability sake we recall the statement.

Theorem 6 (Subject Reduction) *Let* $\Gamma;\, [\Gamma]_{\mathtt{1}}^{\mathtt{IO}} \vdash P$*. Then*

(1) if $\Gamma \vdash_1 P \overset{(\Gamma')u!(V)}{\longrightarrow} Q$*, then (a)* $\Gamma + \Gamma'; [\Gamma + \Gamma']_1^{\mathtt{IO}} \vdash Q$*, (b)* $\Gamma + [\Gamma]_1^{\mathtt{IO}} \vdash u{:}S$*,* $\Gamma + \Gamma' \vdash V{:}T$ *and* $S <: \langle T\rangle^{\mathtt{O}}$*;*

(2) if $\Gamma \vdash_1 P \overset{u?(F)}{\longrightarrow} Q$*, then (a)* $(\Gamma; [\Gamma]_1^{\mathtt{IO}}) + \mathtt{Env}(F) \vdash Q$ *and (b)* $\Gamma + [\Gamma]_1^{\mathtt{IO}} \vdash u{:}S$ *with* $S <: \langle \mathtt{schof}(F)\rangle^{\mathtt{I}}$*;*

(3) if $\Gamma \vdash_1 P \overset{(\Gamma')u \multimap v}{\longrightarrow} Q$*, then (a)* $\Gamma + \Gamma'; [\Gamma + \Gamma']_1^{\mathtt{IO}} \vdash Q$ *and (b)* $\Gamma \setminus \mathtt{dom}([\Gamma + \Gamma']_1^{\mathtt{IO}}) \vdash u : S$*,* $\Gamma + \Gamma' \vdash v{:}\langle T\rangle^{\mathtt{O}}$ *and* $S <: \langle T\rangle^{\mathtt{I}}$*;*

(4) if $\Gamma \vdash_1 P \overset{(u@1':\langle S\rangle^{\mathtt{IO}})}{\longrightarrow} Q$*, then* $(\Gamma; [\Gamma]_1^{\mathtt{IO}}) + u{:}\langle S\rangle^{\mathtt{IO}} \vdash Q$*;*

(5) if $\Gamma \vdash_1 P \xrightarrow{\tau} Q$*, then* $\Gamma; [\Gamma]_1^{\mathtt{IO}} \vdash Q$*.*

Let $\vdash \mathsf{M}$*. Then*

(6) if $\mathsf{M} \xrightarrow{\Delta} \mathsf{N}$*, then* $\vdash \mathsf{N}$*.*

Proof: The proof proceeds by induction on the structure of the derivation of $\Gamma \vdash_1 P \xrightarrow{\mu} Q$ and by cases on the last rule that has been applied for the first five items. Item (6) is similar, but the induction is on the structure of the derivation of $\vdash \mathsf{M}$. We omit the cases that are straightforward.

When the last rule is an instance of (TR4) we have:

$$\frac{\Gamma + v{:}\langle S\rangle^{\kappa} \vdash_1 P \overset{(\Gamma')u!(V)}{\longrightarrow} Q \quad v \neq u \quad v \in \mathtt{fv}(V) \setminus \mathtt{dom}(\Gamma')}{\Gamma \vdash_1 \mathtt{new}\ v : \langle S\rangle^{\kappa}\ \mathtt{in}\ P \overset{(\Gamma' + v:\langle S\rangle^{\kappa})u!(V)}{\longrightarrow} Q}$$

By inductive hypotheses applied to $\Gamma + v : \langle S\rangle^{\kappa} \vdash_1 P \overset{(\Gamma')u!(V)}{\longrightarrow} Q$ we obtain

$$\Gamma + v{:}\langle S\rangle^{\kappa} + \Gamma'; [\Gamma + v{:}\langle S\rangle^{\kappa} + \Gamma']_1^{\mathtt{IO}} \vdash Q \tag{B.9}$$
$$\Gamma + v{:}\langle S\rangle^{\kappa} + [\Gamma + v{:}\langle S\rangle^{\kappa}]_1^{\mathtt{IO}} \vdash u{:}S' \tag{B.10}$$
$$\Gamma + v{:}\langle S\rangle^{\kappa} + \Gamma' \vdash V{:}T \tag{B.11}$$
$$S' <: \langle T\rangle^{\mathtt{O}} \tag{B.12}$$

The conclusion (a) follows from (B.9); the conclusion (b) follows by (B.10), (B.11), and (B.12) because $u \neq v$.

When the last rule is an instance of (TR5) we have:

$$\overset{\text{(TR5)}}{\frac{E \Downarrow V \quad (\Gamma \vdash V \notin F_i)^{i \in 1..j-1} \quad \Gamma \vdash V \in F_j \rightsquigarrow \sigma}{\Gamma \vdash_1 \mathtt{match}\ E\ \mathtt{with}\ \{F_i \Rightarrow P_i^{i\in 1..n}\} \xrightarrow{\tau} P_j\sigma}}$$

By the hypothesis $\Gamma; [\Gamma]_1^{\mathtt{IO}} \vdash P$, Lemma 13, and rule (MATCH) we have:

$$\Gamma; [\Gamma]_1^{\mathtt{IO}} \vdash V : S \qquad S <: \sum_{i\in 1..n} \mathtt{schof}(F_i) \tag{B.13}$$

By Lemma 14 applied to $\Gamma \vdash V \in F_i \rightsquigarrow \sigma$ and (B.13) we obtain that, for every $v \in \mathtt{fv}(F)$, $\Gamma + \Gamma' \vdash \sigma(v) : T'$ and T' <: $\mathtt{Env}(F)(v)$. By Lemma 11 applied to this last judgment, we derive $\Gamma\,;\, [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash P_j\sigma$.

When the last rule is an instance of (TR8) we have:

$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u\,!(V)} P' \quad \Gamma \vdash_{\mathtt{l}} Q \xrightarrow{u?(F)} Q' \quad \mathtt{dom}(\Gamma') \cap \mathtt{fv}(Q) = \emptyset \quad \Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma}{\Gamma \vdash_{\mathtt{l}} \mathtt{spawn}\ \{P\}\ Q \xrightarrow{\tau} \mathtt{new}\ \Gamma'\ \mathtt{in}\ \mathtt{spawn}\ \{P'\}\ Q'\sigma}$$

By inductive hypotheses on $\Gamma \vdash P \xrightarrow{(\Gamma')u\,!(V)} P'$ and $\Gamma \vdash Q \xrightarrow{u?(F)} Q'$ we have:

$$\Gamma + \Gamma'\,;\, [\Gamma + \Gamma']^{\mathtt{IO}}_{\mathtt{l}} \vdash P' \tag{B.14}$$
$$\Gamma + \Gamma' \vdash V : T \tag{B.15}$$
$$\Gamma + [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash u : S \quad S <: \langle \mathtt{schof}(F)\rangle^{\mathtt{I}} \quad S <: \langle T\rangle^{\mathtt{O}} \tag{B.16}$$
$$(\Gamma\,;\, [\Gamma]^{\mathtt{IO}}_{\mathtt{l}}) + \mathtt{Env}(F) \vdash Q' \tag{B.17}$$

By Lemma 14 applied to $\Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma$, (B.15), and (B.16) we obtain that, for every $v \in \mathtt{fv}(F)$, $\Gamma + \Gamma' \vdash \sigma(v) : T'$ and T' <: $\mathtt{Env}(F)(v)$. By Lemma 11 applied to this last judgment, (B.16), and (B.17) we derive $(\Gamma\,;\, [\Gamma]^{\mathtt{IO}}_{\mathtt{l}}) + \Gamma' \vdash Q'\sigma$. We conclude with (B.14), (SPAWN), and (NEW).

The case (TR10) is omitted because the resulting process is complex and the demonstration requires a long uninteresting analysis of the proof tree.

When the last rule is an instance of (DTR1) we have:

(DTR1)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(v_i : S_i{}^{i\in I})u\,!(V)} Q \qquad u@\mathtt{l}' \qquad (v_i@\mathtt{l} \qquad v_i \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Gamma'))^{i\in I} \\ \Delta = v_i : S_i{}^{i\in I} + ((\Gamma|_{\mathtt{fv}(V)}) \setminus \mathtt{l}')\ \mathsf{meet}\ ((\Gamma'|_{\mathtt{fv}(V)}) \setminus \mathtt{l})\end{array}}{\Gamma \vdash_{\mathtt{l}} P \parallel \Gamma' \vdash_{\mathtt{l}'} R \quad \xrightarrow{\Delta\setminus\mathtt{l},\mathtt{l}'} \quad \Gamma + v_i : S_i{}^{i\in I} \vdash_{\mathtt{l}} Q \parallel \Gamma' + \Delta \vdash_{\mathtt{l}'} \mathtt{spawn}\ \{u!(V)\}\ R}$$

In order to prove $\vdash (\Gamma + v_i : S_i{}^{i\in I} \vdash_{\mathtt{l}} Q \parallel \Gamma' + \Delta \vdash_{\mathtt{l}'} \mathtt{spawn}\ \{u!(V)\}\ R)$ we may reduce to demonstrate

$$\Gamma + v_i : S_i{}^{i\in I}\,;\, [\Gamma + v_i : S_i{}^{i\in I}]^{\mathtt{IO}}_{\mathtt{l}} \vdash Q \tag{B.18}$$
$$\Gamma' + \Delta\,;\, [\Gamma']^{\mathtt{IO}}_{\mathtt{l}'} \vdash \mathtt{spawn}\ \{u!(V)\}\ R \tag{B.19}$$

because the machine consistency follows by definition of meet and the fact that v_i are fresh. (We notice that, by definition of Δ, $[\Gamma' + \Delta]^{\mathtt{IO}}_{\mathtt{l}'} = [\Gamma']^{\mathtt{IO}}_{\mathtt{l}'}$.) The judgment (B.18) and

$$\Gamma + [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash u : S \tag{B.20}$$
$$\Gamma + v_i : S_i{}^{i\in I} \vdash V : T \qquad S <: \langle T\rangle^{\mathtt{O}} \tag{B.21}$$

are a consequence of the inductive hypothesis on $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(v_i{:}S_i{}^{i\in I})\mathtt{u}\,!\,(V)} Q$. As regards (B.19), by $\vdash \mathsf{M}$ we derive $\Gamma'\,;\,[\Gamma']^{\mathtt{IO}}_{\mathtt{l}'} \vdash R$ and by Lemma 8 and Proposition 12 we obtain $\Gamma' + \Delta\,;\,[\Gamma']^{\mathtt{IO}}_{\mathtt{l}'} \vdash R$. To demonstrate $\Gamma' + \Delta\,;\,[\Gamma']^{\mathtt{IO}}_{\mathtt{l}'} \vdash u\,!\,(V)$ we reason as follows ((B.19) is entailed by (SPAWN) applied to these last judgments). By (B.20), $u@\mathtt{l}'$, and the well-typedness of M, we derive

$$\Gamma' + [\Gamma']^{\mathtt{IO}}_{\mathtt{l}'} \vdash u{:}S' \qquad S' <: S \tag{B.22}$$

By (B.21), Lemmas 8 and 9 and Proposition 12 we derive

$$\Gamma' + \Delta \vdash V{:}T' \qquad T' <: T \tag{B.23}$$

The judgment (B.19) follows from (B.22) and (B.23) with the rule (OUT).

When the last rule is an instance of (DTR2) we have:

(DTR2)
$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(u@\mathtt{l}':\langle S\rangle^{\mathtt{IO}})} Q \quad u \notin \mathtt{dom}(\Gamma') \cup \mathtt{dom}(\Gamma)}{\Gamma \vdash_{\mathtt{l}} P \parallel \Gamma' \vdash_{\mathtt{l}'} R \quad \longrightarrow \quad \Gamma + u : \langle S\rangle^{\mathtt{IO}} \vdash_{\mathtt{l}} Q \parallel \Gamma' + u : \langle S\rangle^{\mathtt{IO}} \vdash_{\mathtt{l}'} R}$$

We verify the well-typedness of the two runtime environments; machine consistency is immediate. By the inductive hypothesis on $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(u@\mathtt{l}':\langle S\rangle^{\mathtt{IO}})} Q$ we obtain $\Gamma\,;\,[\Gamma_{\mathtt{l}}]_{\mathtt{IO}} + u : \langle S\rangle^{\mathtt{IO}} \vdash Q$. This is sufficient for the correctness of location $\mathtt{l}$ because $\Gamma\,;\,[\Gamma_{\mathtt{l}}]_{\mathtt{IO}} + u{:}\langle S\rangle^{\mathtt{IO}} = (\Gamma + u{:}\langle S\rangle^{\mathtt{IO}})\,;\,[\Gamma_{\mathtt{l}} + u{:}\langle S\rangle^{\mathtt{IO}}]_{\mathtt{IO}}$. The judgment $(\Gamma' + u : \langle S\rangle^{\mathtt{IO}})\,;\,[\Gamma' + u : \langle S\rangle^{\mathtt{IO}}]^{\mathtt{IO}}_{\mathtt{l}'} \vdash R$ follows by Lemma 8 applied to $\Gamma'\,;\,[\Gamma']^{\mathtt{IO}}_{\mathtt{l}'} \vdash R$.

When the last rule is an instance of (DTR3) we have

(DTR3)
$$\frac{\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma'')u \multimap v} Q \quad u@\mathtt{l}' \quad \Gamma' \vdash u : \langle S\rangle^{\kappa} \quad \mathtt{dom}(\Gamma'') \cap \mathtt{dom}(\Gamma') = \emptyset \quad \Gamma''' = \Gamma|_{\{v\}} + \Gamma''}{\Gamma \vdash_{\mathtt{l}} P \parallel \Gamma' \vdash_{\mathtt{l}'} R \quad \longrightarrow \quad \Gamma + \Gamma'' \vdash_{\mathtt{l}} Q \parallel \Gamma' + \Gamma''' \vdash_{\mathtt{l}'} \mathtt{spawn}\,\{u?(x : S)\ v\,!\,(x)\}\ R}$$

We focus on the well-typedness of the two runtime environments. By inductive hypothesis on $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma'')u \multimap v} Q$ we obtain

$$\Gamma + \Gamma''\,;\,[\Gamma + \Gamma'']^{\mathtt{IO}}_{\mathtt{l}} \vdash Q \tag{B.24}$$
$$\Gamma + \Gamma'' \vdash \mathtt{u}{:}T \tag{B.25}$$
$$\Gamma + \Gamma'' \vdash v{:}\langle T'\rangle^{\mathtt{O}} \tag{B.26}$$
$$T <: \langle T'\rangle^{\mathtt{I}} \tag{B.27}$$

By (B.24) we immediately derive that the left runtime environment is well-typed. Therefore we focus on the right runtime environment. To demonstrate the correctness of its process we will eventually use (SPAWN). Therefore we reduce to prove: (1) $\Gamma' + \Gamma'''; [\Gamma' + \Gamma''']^{\mathtt{IO}}_{\mathtt{l}'} \vdash R$ and (2) $\Gamma' + \Gamma'''; [\Gamma' + \Gamma''']^{\mathtt{IO}}_{\mathtt{l}'} \vdash u?(x : S)v\,!(x)$. The judgment (1) follows by the hypothesis $\Gamma'; [\Gamma']^{\mathtt{IO}}_{\mathtt{l}'} \vdash R$, $\mathtt{dom}(\Gamma'') \cap \mathtt{dom}(\Gamma') = \emptyset$, by Lemma 8 and (in case $v \in \mathtt{dom}(\Gamma')$) Proposition 12. As regards (2), the well-typedness of M entails $\langle S\rangle^{\kappa}$ <: T. By transitivity of <:, $\langle S\rangle^{\kappa}$ <: $\langle T'\rangle^{\mathtt{I}}$. Therefore κ is either $\mathtt{I}$ or $\mathtt{IO}$ and S <: T'. Without loss of generality, let x be fresh. Since $\Gamma + \Gamma'' = \Gamma + \Gamma'''$, (B.26) and Lemma 8 give $\Gamma' + \Gamma''' + x : S \vdash v : \langle T'\rangle^{\mathtt{O}}$. Then, by rule (OUT), we obtain $\Gamma' + \Gamma''' + x : S; [\Gamma' + \Gamma''' + x : S]^{\mathtt{IO}}_{\mathtt{l}} \vdash v\,!(x)$. Finally, it is easy to derive $\Gamma' + \Gamma''' + [\Gamma' + \Gamma''']^{\mathtt{IO}}_{\mathtt{l}} \vdash u : \langle S\rangle^{\kappa}$ from the hypothesis $\Gamma' \vdash u : \langle S\rangle^{\kappa}$. We conclude with (SELECT).

When the last rule is an instance of (DTR5) we have:

$$\text{(DTR5)}\quad \frac{\mathsf{M} \xrightarrow{\Delta} \mathsf{N} \quad (\mathtt{dom}(\mathsf{N}) \setminus \mathtt{dom}(\mathsf{M})) \cap \mathtt{dom}(\Gamma) = \emptyset \quad \Delta@\mathtt{l} \subseteq \Gamma}{\mathsf{M} \parallel \Gamma \vdash_{\mathtt{l}} P \xrightarrow{\Delta\setminus\mathtt{l}} \mathsf{N} \parallel \Gamma \vdash_{\mathtt{l}} P}$$

Since $\vdash (\mathsf{M} \parallel \Gamma \vdash_{\mathtt{l}} P)$ then both (1) $\vdash \mathsf{M}$ and (2) $\vdash (\Gamma \vdash_{\mathtt{l}} P)$. By inductive hypotheses applied to (1) and $\mathsf{M} \xrightarrow{\Delta} \mathsf{N}$ we derive $\vdash \mathsf{N}$. The machine consistency of the composite machine follows from that of $\vdash (\mathsf{M} \parallel \Gamma \vdash_{\mathtt{l}} P)$ and the constraint $\Delta@\mathtt{l} \subseteq \Gamma$. □

The proof of the Progress Theorem follows.

Theorem 7 (Progress) *Let Γ be channeled.*

(1) If $\Gamma \vdash V : S$ and S <: $\mathtt{schof}(F)$, then there is σ such that $\Gamma \vdash V \in F \rightsquigarrow \sigma$;

(2) If $\Gamma; [\Gamma]^{\mathtt{IO}}_{\mathtt{l}} \vdash P$ and $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u\,!(V)} Q'$ and $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{u?(F)} Q''$, then there is Q such that $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{\tau} Q$;

(3) If $\vdash (\Gamma \vdash_{\mathtt{l}} P \parallel \mathsf{M})$, $\Gamma \vdash_{\mathtt{l}} P \xrightarrow{(\Gamma')u\,!(V)} Q$, and u is located at a location of M, then $\Gamma \vdash_{\mathtt{l}} P \parallel \mathsf{M} \xrightarrow{\Delta} \Gamma \vdash_{\mathtt{l}} Q \parallel \mathsf{N}$, for some N. Similarly when the label is $(\Gamma')u \multimap v$.

Proof: As regards item (1), let the size of a pattern F, written $h(F)$, be defined as follows:

$$\begin{aligned} h(\mathtt{()}) = h(\mathtt{B}) = h(\langle S\rangle^{\kappa}) = h(L[F]) &= 1 \\ h(S^*) &= 1 + h(S) \\ h(x : F) &= 2 + h(F) \\ h(F_1, F_2) = h(F_1 + F_2) &= 1 + h(F_1) + h(F_2) \\ h(\mathtt{Y}) &= 1 + h(\mathcal{F}(\mathtt{Y})) \end{aligned}$$

Notice that $h(F)$ is well-defined when F is a well-formed pattern because a pattern name $\mathtt{Y}$ cannot occur unguarded in $\mathcal{F}(\mathtt{Y})$ and $L[F]$ has size 1 regardless of F's size. We generalize the h function to markers and to sequences of patterns and markers, where the size of a marker is 1 and the size of a sequence $\Phi = F_1 :: F_2 :: \cdots :: F_n$ is defined as the sum of the sizes of all of its elements.

The proof is by induction on the pair $(V, h(\Phi))$, the idea being that at each induction step either we reduce to pattern matching a value that is structurally smaller than V or the size of the pattern sequence decreases. Recall that, since S is the schema of a value, it does not contain $+$'s, starred schemas, and schema names, except possibly within channel constructors.

We only show the most relevant cases. In the base case we have $h(\Phi) = 0$ and $V = \texttt{()}$. We conclude immediately by (PM1). Assume $h(\Phi) > 0$, meaning that $\Phi = F :: \Phi'$ for some F and Φ'. We reason by cases on the structure of F.

Assume $F = \texttt{()}$. We notice that $\texttt{schof}(\Phi)$ <: $\texttt{schof}(\Phi')$ and that $h(\Phi') < h(\Phi)$. By induction hypothesis we obtain $\Gamma \vdash V \in \Phi' \leadsto \sigma$ from which we conclude by (PM2).

Assume $F = L[F']$. Then $V = a[V']@V''$ where $a \in L$, $\Gamma \vdash V' : S'$, $\Gamma \vdash V'' : S''$, S' <: $\texttt{schof}(F')$, and S'' <: $\texttt{schof}(\Phi')$. By induction hypothesis we obtain $\Gamma \vdash V' \in F' \leadsto \sigma$ and $\Gamma \vdash V'' \in \Phi' \leadsto \sigma'$ and we conclude by (PM6).

Assume $F = F_1 + F_2$. Notice that $\texttt{schof}(\Phi)$ <: $\texttt{schof}(F_1 :: \Phi') + \texttt{schof}(F_2 :: \Phi')$. By Proposition 10(3) we have that either S <: $\texttt{schof}(F_1 :: \Phi')$ or S <: $\texttt{schof}(F_2 :: \Phi')$. If S <: $\texttt{schof}(F_1 :: \Phi')$ then by induction hypothesis $\Gamma \vdash V \in F_1 :: \Phi' \leadsto \sigma$ and we conclude by (PM8). If $S \not<: \texttt{schof}(F_1 :: \Phi')$ then by Lemma 14(1) we have $\Gamma \vdash V \notin F_1 :: \Phi'$. From S <: $\texttt{schof}(F_2 :: \Phi')$ and the induction hypothesis we obtain $\Gamma \vdash V \in F_2 :: \Phi' \leadsto \sigma$ from which we conclude by (PM9).

Assume $F = T^*$. Let $V = V_1@V_2$ so that $\Gamma \vdash V_1 : S_1$ and $\Gamma \vdash V_2 : S_2$ and S_1 <: T^* and S_2 <: $\texttt{schof}(\Phi')$. We take V_1 to be the longest prefix of V with these properties. The existence of V_1 and V_2 is guaranteed by Proposition 10(4). By induction hypothesis we obtain that $\Gamma \vdash V_2 \in \Phi' \leadsto \sigma$.

Now we reason on the structure of V_1 to show that there exists $n \geq 0$ such that $\Gamma \vdash V_1 \in T^n \leadsto \emptyset$. Assume $V_1 = \texttt{()}$. Then it is sufficient to take $n = 0$. Assume $V_1 \neq \texttt{()}$. By Proposition 10(5) there exist V_1' and V_1'' such that $V_1' \neq \texttt{()}$ and $\Gamma \vdash V_1' : S_1'$ and $\Gamma \vdash V_1'' : S_1''$ and S_1' <: T and S_1'' <: T^*. By induction hypothesis we obtain that $\Gamma \vdash V_1' \in T \leadsto \emptyset$ and furthermore there exists $m \geq 0$ such that $\Gamma \vdash V_1'' \in T^m \leadsto \emptyset$. Now it is sufficient to take $n = m + 1$ and we conclude by noticing that if $\Gamma \vdash V_1' \in T \leadsto \emptyset$ and $\Gamma \vdash V_1'' \in T^m \leadsto \emptyset$, then $\Gamma \vdash V_1'@V_1'' \in T\texttt{,}T^m \leadsto \emptyset$.

Because V_1 was chosen as the longest prefix of V such that S_1 <: T^* and S_2 <: $\texttt{schof}(\Phi')$, by soundness of pattern matching (Lemma 14(1)) we conclude that any extension of V_1 with a nonempty suffix W such that $V_2 = W@V_2'$ will lead us to conclude either $\Gamma \vdash V_1@W \notin T^*$ or $\Gamma \vdash V_2' \notin \Phi'$. Hence we conclude by (PM12).

As regards item (2), by Theorem 6(1) applied to $\Gamma; [\Gamma]_1^{\mathtt{IO}} \vdash P$ and $\Gamma; [\Gamma]_1^{\mathtt{IO}} \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q'$, we derive $\Gamma + [\Gamma]_1^{\mathtt{IO}} \vdash u : S$, $\Gamma + \Gamma' \vdash V : T$ and S <: $\langle T \rangle^{\mathtt{O}}$. By Theorem 6(2) applied to $\Gamma; [\Gamma]_1^{\mathtt{IO}} \vdash P$ and $\Gamma; [\Gamma]_1^{\mathtt{IO}} \vdash_1 P \xrightarrow{u?(F)} Q''$, we also derive $\Gamma + [\Gamma]_1^{\mathtt{IO}} \vdash u : S$ and S <: $\langle \mathtt{schof}(F) \rangle^{\mathtt{I}}$. Since Γ is channeled, $S = \langle S' \rangle^{\kappa}$, for some S', κ, and by Proposition 2, T <: $\mathtt{schof}(F)$. Therefore, by item 1, there is σ such that $\Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma$. The proof now requires a close inspection of the proof trees of $\Gamma \vdash_1 P \xrightarrow{(\Gamma')u!(V)} Q'$ and $\Gamma \vdash_1 P \xrightarrow{u?(F)} Q''$. By definition of the transition relation, these trees must have common subtrees beginning at the root and terminating in correspondence of a subterm $\mathtt{spawn}\ \{P'\}\ \ P''$ of P. At this point, the two subtrees continue with premises $\Gamma + \Gamma''' \vdash_1 P' \xrightarrow{(\Gamma'')u!(V)} Q_1'$ and $\Gamma + \Gamma''' \vdash_1 P'' \xrightarrow{u?(F)} Q_2''$ (or conversely). Progress holds because rule (TR8) may be applied (the constraint $\mathtt{dom}(\Gamma'') \cap \mathtt{fv}(P'') = \emptyset$ may be easily enforced by alpha-conversion) to $\mathtt{spawn}\ \{P'\}\ \ P''$ and the resulting τ-transition may be lifted to P by means of rules (TR3), (TR6), (TR7).

Item (3) is straightforward. □

C The subschema relation and the type system

The definition of <: in Section 4 is given coinductively and it is hard to implement directly. In this section we illustrate the algorithm used in PiDuce for <: and we demonstrate its soundness and completeness. The algorithm follows the style of Hosoya, Pierce and Vouillon [22] has an exponential computational cost (in the sizes of the argument schemas). In order to alleviate this cost we define a subclass of schemas and demonstrate the existence of a polynomial algorithm for them.

Let $\mathsf{handles}(S) = \{R \mid S \downarrow R\}$ and let $\wp(1..n)$ be the set of subsets of numbers in $1..n$. Table C.1 contains the inference rules that define a relation $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$, which we are going to relate with <:. The set $\mathtt{A}$, called *assumptions*, is a set of pairs (S, T) representing relations that have been proved or that are being proved. The set $\mathtt{A}'$, following Brand and Henglein [9], is used for recording already computed or being computed relations. The rules parse the structure of handles of the left schema. Rule (EMPTY) accounts for left schemas with no handle (empty schemas). Rules (VOID), (BASE), (CHAN), (SPLIT), and (LSEQ) deal with schemas that are handles (void, sequences with an initial schema that is either basic or channel or labelled). They closely correspond to the items 1,

Table C.1
The algorithmic subschema (arguments of shape `B` are always replaced by `B,()`. Similarly for $\langle S\rangle^{\kappa}$, $L[S]$, $S+S'$, `U`, and S^*. Arguments `()`,S are always replaced by S).

(EMPTY)
$$\frac{\mathsf{handles}(S)=\emptyset}{S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}}$$

(VOID)
$$\frac{T \downarrow \mathtt{()}}{\mathtt{()} \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}}$$

(BASE)
$$\frac{(T \downarrow \mathtt{B}_i, T_i \quad \mathtt{B} \sqsubseteq \mathtt{B}_i)^{i\in 1..n} \quad S \preceq_{\mathtt{A}} \sum_{i\in\{1,\dots,n\}} T_i \Rightarrow \mathtt{A}'}{\mathtt{B}, S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'}$$

(CHAN)
$$\frac{\begin{array}{c} (T \downarrow \langle T_i\rangle^{\kappa_i}\mathtt{,}T'_i)^{i\in 1..n} \quad \kappa \leq \kappa_i \\ \left(\begin{array}{rl} \kappa_i = \mathtt{O} & \text{implies} \;\; T_i \preceq_{\mathtt{A}_{i-1}} S \Rightarrow \mathtt{A}_i \\ \kappa_i = \mathtt{I} & \text{implies} \;\; S \preceq_{\mathtt{A}_{i-1}} T_i \Rightarrow \mathtt{A}_i \\ \kappa_i = \mathtt{IO} & \text{implies} \;\; S \preceq_{\mathtt{A}_{i-1}} T_i \Rightarrow \mathtt{A}'_i \text{ and } T_i \preceq_{\mathtt{A}'_i} S \Rightarrow \mathtt{A}_i \end{array}\right)^{i\in 1..n} \\ S' \preceq_{\mathtt{A}_n} \sum_{i\in 1..n} T'_i \Rightarrow \mathtt{A}_{n+1} \end{array}}{\langle S\rangle^{\kappa}\mathtt{,}S' \preceq_{\mathtt{A}_0} T \Rightarrow \mathtt{A}_{n+1}}$$

(SPLIT)
$$\frac{\begin{array}{c} T \downarrow L'[T']\mathtt{,}T'' \quad \widehat{L} \not\subseteq \widehat{L'} \quad \widehat{L} \cap \widehat{L'} \neq \emptyset \\ (L \setminus L')[S]\mathtt{,}S' \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}' \quad (L \cap L')[S]\mathtt{,}S' \preceq_{\mathtt{A}'} T \Rightarrow \mathtt{A}'' \end{array}}{L[S]\mathtt{,}S' \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}''}$$

(LSEQ)
$$\frac{\begin{array}{c} (T \downarrow L_i[T_i]\mathtt{,}T'_i)^{i\in 1..n} \quad \widehat{L} \subseteq \bigcap_{i\in 1..n} \widehat{L_i} \quad J_1, \cdots, J_{2^n} = \wp(1..n) \\ \left(S \preceq_{\mathtt{A}_{k-1}} \sum_{i\in J_k} T_i \Rightarrow \mathtt{A}_k \quad \mathtt{or} \quad S' \preceq_{\mathtt{A}_{k-1}} \sum_{i\in 1..n\setminus J_k} T'_i \Rightarrow \mathtt{A}_k\right)^{k\in 1..2^n} \end{array}}{L[S]\mathtt{,}S' \preceq_{\mathtt{A}_0} T \Rightarrow \mathtt{A}_{2^n}}$$

(UNION)
$$\frac{S\mathtt{,}S'' \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}' \quad S'\mathtt{,}S'' \preceq_{\mathtt{A}'} T \Rightarrow \mathtt{A}''}{(S+S')\mathtt{,}S'' \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}''}$$

(NAME)
$$\frac{\begin{array}{c} \mathtt{A}' = \mathtt{A} \cup \{(\mathtt{U}\mathtt{,}S, T)\} \\ \mathcal{E}(\mathtt{U})\mathtt{,}S \preceq_{\mathtt{A}'} T \Rightarrow \mathtt{A}'' \end{array}}{\mathtt{U}\mathtt{,}S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}''}$$

(STAR)
$$\frac{\begin{array}{c} \mathtt{A}' = \mathtt{A} \cup \{(S^*\mathtt{,}S',\ T)\} \\ (\mathtt{()} + S\mathtt{,}S^*)\mathtt{,}S' \preceq_{\mathtt{A}'} T \Rightarrow \mathtt{A}'' \end{array}}{S^*\mathtt{,}S' \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}''}$$

(ASMP)
$$\frac{(S,T) \in \mathtt{A}}{S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}}$$

2, 3, 4.a and 4.b of `<:`, respectively. The remaining rules are used for reducing the computation to such rules. Rule (UNION) applies to schemas $S+S'\mathtt{,}R$ and verifies that both $S\mathtt{,}R$ and $S'\mathtt{,}R$ are subschemas of T. Rule (NAME) accounts for left schemas of shape `U`,S. In this case the name `U` is replaced by its definition $\mathcal{E}(\mathtt{U})$, the set of assumptions is extended with the pair $(\mathtt{U}\mathtt{,}S,\ T)$ and the relation $\preceq$ is computed on these new arguments. Rule (STAR) is similar to (NAME) but for starred schemas. Rule (ASMP) terminates proofs when the arguments are already in the set of assumptions.

The relation $\preceq$ is sound with respect to `<:`.

Lemma 15 (Soundness) *If $S \preceq_{\emptyset} T \Rightarrow \mathtt{A}$, then S `<:` T.*

Proof: Let $\mathcal{R}$ be the relation containing

(1) pairs (S', T') such that a subtree $S' \preceq_{\mathtt{A}'} T' \Rightarrow \mathtt{A}''$ exists in the tree $S \preceq_{\emptyset} T \Rightarrow \mathtt{A}$;
(2) if $(\mathtt{B}, T') \in \mathcal{R}$, then $(\mathtt{B,()}, T') \in \mathcal{R}$, too. Similarly for pairs $(\langle S'\rangle^{\kappa}, T')$, $(\langle S'\rangle^{\kappa}, T')$, $(L[S'], T')$, $(S' + S'', T')$, $(\mathtt{U}, T')$, and (S^*, T').

To check that $\mathcal{R}$ is a subschema relation, let $(S', T') \in \mathcal{R}$ and $S' \downarrow R$. By induction on the structure of the proof $S' \downarrow R$ it is easy to show that $(R, T) \in \mathcal{R}$, too. $\square$

We note that the rules in Table C.1 define a program, which we call the $\preceq$-*program*, that takes a triple $(S, T, \mathtt{A})$ and attempts to build the proof tree by recursively analyzing the structure of S and the set $\mathtt{A}$. The program returns a set $\mathtt{A}'$ if the attempt succeeds, returns a failure otherwise. The $\preceq$-program terminates. To demonstrate this property we introduce some notation:

- $\mathfrak{t}(S)$, called the *set of subterms* of S, is the smallest set satisfying the equations:

$$
\begin{aligned}
\mathfrak{t}(\mathtt{()}) &= \{\mathtt{()}\}\\
\mathfrak{t}(\mathtt{B}) &= \{\mathtt{B}\} \cup \{\mathtt{B,()}\}\\
\mathfrak{t}(\mathtt{U}) &= \{\mathtt{U}\} \cup \{\mathtt{U,()}\} \cup \{\mathfrak{t}(\mathcal{E}(\mathtt{U}))\}\\
\mathfrak{t}(\langle S\rangle^{\kappa}) &= \{\langle S\rangle^{\kappa}\} \cup \{\langle S\rangle^{\kappa}\mathtt{,()}\} \cup \mathfrak{t}(S)\\
\mathfrak{t}(L[S]) &= \{L[S]\} \cup \{L[S]\mathtt{,()}\} \cup \mathfrak{t}(S)\\
\mathfrak{t}(S\mathtt{,}S') &= \{T\mathtt{,}S' \mid T \in \mathfrak{t}(S)\} \cup \{\mathfrak{t}(S')\}\\
\mathfrak{t}(S^*) &= \mathfrak{t}(S) \cup \{S^*\} \cup \{S\mathtt{,}S^*\} \cup \{\mathtt{()}\}\\
\mathfrak{t}(S+T) &= \{S+T\} \cup \mathfrak{t}(S) \cup \mathfrak{t}(T)
\end{aligned}
$$

  It is easy to demonstrate that $\mathfrak{t}(S)$ is always finite.
- $\mathtt{anames}(S)$ is the set $\{\mathtt{U,}T : \mathtt{U,}T \in \mathfrak{t}(S)\} \cup \{T^*\mathtt{,}T' : T^*\mathtt{,}T' \in \mathfrak{t}(S)\}$
- $\mathtt{lsubt}(S, T)$ is the smallest set containing $\mathfrak{t}(S)$, $\mathfrak{t}(T)$ and closed under the following properties:
  - if $L[Q]\mathtt{,}Q' \in \mathtt{lsubt}(S, T)$ and $L'[Q'']\mathtt{,}Q''' \in \mathtt{lsubt}(S, T)$ and $\widehat{L} \not\subseteq \widehat{L'}$ then $(L \setminus L')[Q]\mathtt{,}Q' \in \mathtt{lsubt}(S, T)$ and $(L \cap L')[Q]\mathtt{,}Q' \in \mathtt{lsubt}(S, T)$
  - if $S\mathtt{,}S' \in \mathfrak{t}(S)$ and $T\mathtt{,}T' \in \mathfrak{t}(S)$ then $S' + T' \in \mathfrak{t}(S)$;

  Since $\mathfrak{t}(S)$ and $\mathfrak{t}(T)$ are finite then $\mathtt{lsubt}(S, T)$ is finite as well.
- $\|S\|_{\mathcal{X}}$, called the *size* of S with names in $\mathcal{X}$, is the function inductively defined as:

$$
\|S\|_{\mathcal{X}} = \begin{cases}
0 & \text{if } S = \mathtt{U} \in \mathcal{X}\\
1 & \text{if } S = \mathtt{()}\\
\|\mathcal{E}(\mathtt{U})\|_{\mathcal{X}\cup\{\mathtt{U}\}} & \text{if } S = \mathtt{U} \notin \mathcal{X}\\
1 + \|T\|_{\mathcal{X}} & \text{if } S = \langle T\rangle^{\kappa} \text{ or } S = L[T] \text{ or } S = T^*\\
1 + \|T\|_{\mathcal{X}} + \|T'\|_{\mathcal{X}} & \text{if } S = T\mathtt{,}T' \text{ or } S = T + T'
\end{cases}
$$

  The number $\|S\|_{\emptyset}$ is shortened into $\|S\|$.

We note that $\|S\|$ and $|\mathsf{t}(S)|$ are finite (because $\mathcal{E}$ is a finite map). They are also different values in general. For instance $\|S + S\| = 2 \times \|S\| + 1$ whilst $|\mathsf{t}(S + S)| = |\mathsf{t}(S)| + 1$.

Lemma 16 *(1) The set* $\mathsf{handles}(S)$ *is always finite.*
(2) The $\preceq$*-program always terminates.*

Proof: As regards (1), let $h(S)$ be the function defined as

$$h(S) = \begin{cases} 0 & \text{if } S \textit{ is empty} \\ 1 & \text{if } S = \texttt{()} \text{ or } S = \langle T \rangle^{\kappa} \\ 1 & \text{if } S = L[T] \text{ and } S \textit{ is not-empty} \\ h(T) + h(T') & \text{if } (S = T + T' \text{ or } S = T \,\texttt{,}\, T') \text{ and } S \textit{ is not-empty} \\ 1 + h(T) & \text{if } S = T^{*} \\ 1 + h(\mathcal{E}(\texttt{U})) & \text{if } S \textit{ is not-empty} \text{ and } S = \texttt{U} \end{cases}$$

Since $\mathcal{E}$ is well-formed, $h(S)$ is finite for every schema. The proof proceeds by induction on $h(S)$. The base case is obvious. The inductive case is by cases on the structure of S. We discuss the subcase $S = \texttt{U}$. We observe that, by definition, $\mathsf{handles}(\texttt{U}) = \mathsf{handles}(\mathcal{E}(\texttt{U}))$. By inductive hypothesis $\mathsf{handles}(\mathcal{E}(\texttt{U}))$ is finite; therefore $\mathsf{handles}(\texttt{U})$ is finite as well.

As regards (2), let $n_{S,T,\texttt{A}} = |(\texttt{anames}(S) \cup \texttt{anames}(T)) \times \texttt{lsubt}(S, T) \setminus \texttt{A}|$ (the subtrees of T are considered because of the contravariance of $\langle \cdot \rangle^{\texttt{O}}$). We note that $\texttt{A}$ is contained into $(\texttt{anames}(S) \cup \texttt{anames}(T)) \times \texttt{lsubt}(S, T)$. We demonstrate that every invocation of $S \preceq_{\texttt{A}} T \Rightarrow \texttt{A}'$ in the premises of the rules of Table C.1 decreases the value $(n_{S,T,\texttt{A}}, \|S\| + \|T\|)$ (the order is lexicographic) of the conclusion. There is one problematic case: when the $\preceq$-program tries to apply (SPLIT). In this case, the value $(n_{S,T,\texttt{A}}, \|S\| + \|T\|)$ for the two premises is equal to that of the conclusion. However, after a finite number of application of (SPLIT) – corresponding (in the worst case) to the number of labelled handles of T, which are finite by (1) – (SPLIT) reduces to (LSEQ). In (LSEQ) the value $(n_{S,T,\texttt{A}}, \|S\| + \|T\|)$ decreases, thus guaranteeing termination. $\square$

Completeness of $\preceq$ with respect to <: is demonstrated below.

Definition 17 *A triple* $(S, T, \texttt{A})$ *is* correct *if and only if: (1)* S <: T *and (2)* $(S', T') \in \texttt{A}$ *implies* S' <: T'.

Proposition 18 *If* $(S, T, \texttt{A})$ *is correct, then one of the rules in Table C.1 is applied by the* $\preceq$*-program and every judgment used in the premise of the rule is correct as well.*

Proof: Together with the statement of the Proposition, we also demonstrate that if the $\preceq$-program returns a set $\texttt{A}'$, then $\texttt{A}'$ is correct: $(S', T') \in \texttt{A}'$ implies S' <: T'. We focus on not empty schemas S and the argument is by induction on the structure of S. The case of empty schemas is immediate. The case $S = \mathtt{()}$ is immediate as well. As inductive cases, we omit those where S is a sequence of length 1 because they may be reduced to the following ones by Proposition 2(6). If $S = \mathtt{B},S'$, then, by S <: T, there exist $(T \downarrow \mathtt{B}_i, T_i)^{i \in 1..n}$ such that, for every i, $\mathtt{B} \subseteq \mathtt{B}_i$ and S' <: $\sum_{i\in 1..n} T_i$. Therefore, the $\preceq$-program may apply (BASE) reducing to the triple $(S', \sum_{i \in 1..n} T_i, \mathtt{A})$. The correctness of this triple follows by the hypotheses. The output set of the program is correct by inductive hypothesis. The case when $S = \langle S' \rangle^{\kappa}, S''$ is similar to the previous one. When $S = L[S'], S''$ the $\preceq$-program may apply (SPLIT) or (LSEQ) according to condition 4.a or 4.b of Definition 1 is used in <:. Again, the correctness of every triple used in the premises follows by the hypotheses; the output set of every invocation of the program is correct by inductive hypothesis. If $S = S' + S'', R$ then, by Proposition 2(8), both S', R <: T and S'', R <: T. Then the $\preceq$-program may apply (UNION), thus reducing to two triples that are still correct. Similarly, the set that are returned by every invocation of the program are correct by inductive hypothesis. If either $S = \mathtt{U}, S'$ or $S = S'^{*}, S''$ then the $\preceq$-program may apply either (NAME) or (STAR) or (ASMP). In the first two cases, the correctness of the new triple follows by the correctness of the current triple. In the third case no new triple is generated. □

Completeness is an immediate consequence of Proposition 18 and Lemma 16.

Lemma 19 (Completeness) *If S <: T then there exists* $\mathtt{A}$ *such that* $S \preceq_{\emptyset} T \Rightarrow \mathtt{A}$.

Rule (LSEQ) in Table C.1 retains a number of subtrees which is exponential in the size of the right schema. This causes an exponential cost for the $\preceq$-program. Such a cost is an issue in Web-services, where data coming from untrusted parties, such as WSDL documents (containing the schema of a service), might be validated at run-time before processing. Since Web-services documents carry references, validation has to verify that the schema of the reference conforms with some expected schema, thus reducing itself to the subschema relation. It is worth to notice that in XDuce run-time subschema checks are avoided because programs are strictly coupled and typechecking guarantees that invalid values cannot be produced. In CDuce there is the possibility of using pattern matching on function values, thus invoking the subschema relation at run-time. However this feature is never used in CDuce programs.

In [11] a schema language restriction has been studied so that the corresponding subschema relation has a polynomial cost. Specifically, following XML-Schema, schemas are constrained in order to retain a deterministic model as regards tag-labelled transitions. The model is still nondeterministic with respect to channel-labelled transitions.

Definition 20 *The set* ldet *of* label-determined schemas *is the greatest set of schemas such that:*

$\texttt{()} \in \mathsf{ldet}$	
$\texttt{B} \in \mathsf{ldet}$	
$\langle S\rangle^{\kappa} \in \mathsf{ldet}$	if $S \in \mathsf{ldet}$
$L[S] \in \mathsf{ldet}$	if $S \in \mathsf{ldet}$
$S\texttt{,}T \in \mathsf{ldet}$	if $S \in \mathsf{ldet}$ and $T \in \mathsf{ldet}$
$S^{*} \in \mathsf{ldet}$	*if* $S \in \mathsf{ldet}$
$S + T \in \mathsf{ldet}$	if $S \downarrow L[S']\texttt{,}S''$ and $T \downarrow L'[T']\texttt{,}T''$implies $\widehat{L} \cap \widehat{L'} = \emptyset$ and $S \in \mathsf{ldet}, T \in \mathsf{ldet}$
$\texttt{U} \in \mathsf{ldet}$	if $\mathcal{E}(\texttt{U}) \in \mathsf{ldet}$

By the definition $a[S] + (\tilde{\ } \setminus a)[T]$ and $\tilde{\ }[S] + \langle S\rangle^{\kappa} + \langle T\rangle^{\kappa'}$ are label-determined schemas whilst $a[\ ] + (a+b)[\ ]$ and $\langle a[\ ] + \tilde{\ }[\ ]\rangle^{\kappa}$ are not label-determined. It is worth to remark that every empty schema – the schema that does not retain any handle – is in ldet and that schemas like $a[\ ] + a[\texttt{Empty}]$ are also label-determined.

We observe that, if S and T are label-determined then the proof of $S \preceq_{\emptyset} T \Rightarrow \texttt{A}$ never requires the rule (LSEQ), which was problematic for its computational cost. Actually, in [11], the $\preceq$-program has been proved to have a polynomial cost when invoked on label-determined schemas.