PiDuce: a process calculus with native XML datatypes*

Allen L. Brown, Jr.[1], Cosimo Laneve[2], and L. Gregory Meredith[3]

[1] Microsoft Corporation, Redmond, Washington, USA
[2] Department of Computer Science, University of Bologna, Italy
[3] Harvard Medical School, Boston, USA and Djinnisys Corporation, Seattle, USA

Abstract. We develop the static and dynamic semantics of PiDuce, a process calculus with XML values, schemas, and pattern matching. PiDuce values include channel names, therefore the structure of values may not reveal anything about their schemas. This is problematic in the pattern matching algorithm because it requires to verify whether a schema of a channel is a subschema of a pattern. Such a verification has exponential cost, in general. In order to reduce the computational complexity of the pattern matching, channel schemas are constrained to occur in tail positions of sequences and to be labelled-determined.

1 Introduction

The π-calculus has been introduced as a basic formalism for modelling concurrent systems [19]. Its data language is quite simple: only tuples of channel names are admitted. Extensions of the basic model with datatypes, such as integers, booleans, and lists, as well as with polymorphism, have been already explored [18, 21]. In this paper we continue this research by investigating an extension of π-calculus with values and datatypes that closely approximates standard values and datatypes of the web: XML documents and XML schemas, respectively.

Our extension of π-calculus, called PiDuce, has values that may contain channels – a role often played by Uniform Resource Identifiers (URIs) in the web –, as in π-calculus and in XML instances. Correspondingly, datatypes, called schemas, may contain types describing collections of channels that carry messages of similar structure. For instance, the following PiDuce fragment

$$\overline{x}[u] \;\mid\; x(v : \langle \texttt{Int} \rangle).\, \overline{v}[\texttt{5}]$$

consists of two processes in parallel: the process on the left outputs the channel u on x, the process on the right receives a channel carrying integers – this is indicated by the schema $\langle \texttt{Int} \rangle$ – and outputs 5 on it. If the input and output on x communicate, the value 5 will be emitted on u.

* Aspects of this investigation conducted at the University of Bologna were supported in part by a Microsoft initiative in concurrent computing and web services.

PiDuce also retains mechanisms for constructing and deconstructing values. Construction is performed by tagging existing values and putting them in sequence. Deconstruction is achieved by patterns and pattern matching. For instance, the fragment

$$\overline{x}[a[\texttt{5}],b[\texttt{4}]] \;\mid\; x(a[u : \texttt{Int}],b[v : \texttt{Int}]).\,\overline{z}[c[v],d[u]]$$

exchanges the document $a[\texttt{5}],b[\texttt{4}]$ during the communication on x, grasps the values `5` and `4` by means of the pattern $a[u : \texttt{Int}],b[v : \texttt{Int}]$, and constructs a new value $c[\texttt{4}],d[\texttt{5}]$, which is emitted on the channel z.

By combining input choices and patterns it is possible to write sophisticated processes in PiDuce. For instance, the fragment:

$$u(v : S).\,P \;+\; u(v : T).\,Q$$

selects the continuation P or Q according to the received value has schema S or T, respectively. (The choice is nondeterministic if the received value matches with both.) The selection is performed by the pattern matching algorithm that parses the value according to the structure of the pattern. This algorithm has a cost that is proportional to the product of the sizes of the pattern and of the received value, when such a value does not contain channels. When the received value contains a channel, as when it is x, the pattern matching reduces to computing the subschema relation between the schema of x and those of $v : S$ and $v : T$. In turn, this relation calculates a language inclusion and it is decidable because PiDuce schemas define *regular tree languages*.

However the subschema relation has exponential cost in the size of the schemas [17] and this cost may significantly degrade the run-time efficiency of possible implementations since pattern matching is liberally used in PiDuce. To circumvent this problem, channel schemas are restricted to occur in tail position of sequences and to be *labelled-determined*, that is they have a deterministic model as regards labelled transitions (the model is nondeterministic as regards channel transitions). PiDuce labelled-determined schemas admit a subschema relation with a polynomial computational complexity [8].

The subschema relation, which is original to this contribution, uses a "simulation relation" in the style of [3, 20]. In particular, we associate handles to schemas and, in order to derive that S is a subschema of T, we verify that the handles of S are (recursively) contained into the handles of T. The containment goes straight to the structure of the handles when schemas are labelled-determined. Otherwise the structure is not powerful enough. To illustrate the problem, let $S = (a + b)[\texttt{Int} + \texttt{String}],c[\texttt{Int}]$ and $T = a[\texttt{Int}],c[\texttt{Int}] + a[\texttt{String}],c[\texttt{Int}] + b[\texttt{Int} + \texttt{String}],c[\texttt{Int}]$. It turns out that $S <: T$ however, to demonstrate this, one has to pick one addend of T, let it be $T' = a[\texttt{Int}],c[\texttt{Int}]$, compute the difference of S and T', and show that this difference is in T. In this case the difference is $a[\texttt{String}],c[\texttt{Int}] + b[\texttt{Int} + \texttt{String}],c[\texttt{Int}]$ and it is clearly contained in T.

PiDuce possesses a static semantics ensuring that invalid terms can never be produced. This is achieved by a careful control over the interplay between the schemas of the messages and the schemas declared for the channels carrying

them. It is worth noticing that, due to the subschema relation, the static semantics also entails subtyping polymorphism. The static semantics is demonstrated to be sound; this yields a subject reduction property (a well-typed process transits to well typed processes) and a progress property (a well-typed process does not get stuck).

Related Works. `PiDuce` has been strongly influenced by `XDuce`, a functional language for `XML` processing [12]. 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 semistructured data have been studied in recent years. Two similar contributions, that are contemporary and independent to this one, are [10, 2]. The schema language in [10] is the one of [5] plus the channel constructors for input, output, and input-output capability. No apparent restriction to reduce the computational complexity of pattern matching is proposed. The schema language of [2] is simpler than the that of `PiDuce`. In particular recursion is omitted and labeled schemas have singleton labels.

Other contributions integrating semistructured data and processes are discussed in order. `TulaFale` [6], 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$d\pi$ [11] is a language that supports dynamic web page programming. This language is π-calculus with locations plus the 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 similar contribution to [11] is Active `XML` [1] that uses an underlying model consisting of a set of peer locations with data and services.

Structure of the paper. We proceed as follows. We introduce `PiDuce` in Section 2, together with a few examples to elucidate the syntactic constructs, and motivate our design choices. We examine the static semantics of `PiDuce` in Section 3, and we discuss the motivations supporting its design. We introduce the dynamic semantics in Section 4 and we demonstrate a soundness theorem in Appendix A. We conclude with Section 5 where we also discuss few current directions of research.

2 `PiDuce`

The syntax of `PiDuce` includes the categories *labels*, *values*, *schemas*, *patterns*, and *processes* that are defined by the rules in Table 1. Several countably infinite sets are used in the syntax: the set of *tags*, ranged over by $a, b, \cdots$; the set of *variables*, ranged over by $u, x, \cdots$; the set of *schema names*, ranged over by $\mathtt{U}, \mathtt{V}, \cdots$.

$L ::=$		**label**	$S ::=$		**schema**
	a	(tag)		$\texttt{()}$	(void schema)
	$\sim$	(wildcard label)		$\langle S \rangle$	(channel schema)
	$L + L$	(union)		$L[S]$	(labelled schema)
	$L \setminus L$	(difference)		$L[S]\texttt{,}S$	(labelled sequence sch.)
				$S + S$	(union schema)
$V ::=$		**value**		$\texttt{U}$	(schema name)
	$\texttt{()}$	(void)			
	u	(variable)	$P ::=$		**process**
	$a[V]$	(labelled value)		$\mathbf{0}$	(nil)
	$a[V]\texttt{,}V$	(sequence)		$\overline{u}[V]$	(output)
				$\sum_{i \in I} u_i(F_i).P_i$	(input choice)
$F ::=$		**pattern**		$P \mid P$	(parallel)
	S	(schema)		$(u{:}\langle S \rangle)P$	(new)
	$u : F$	(variable pattern)		$!P$	(replication)
	$L[F]$	(labelled pattern)			
	$L[F]\texttt{,}F$	(sequence pattern)			

Table 1. Syntax of `PiDuce`

Labels. Labels specify collections of tags. The semantics of labels is defined by the following function $\widehat{\cdot}$:

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

We write $a \in L$ for $a \in \widehat{L}$.

Values. `PiDuce` values are the set of terms V containing `()`, variables and sequences of elements that are labelled values with the tailing element that may be a labelled value or a variable. For example $a[v]\texttt{,}u$ or $a[u]\texttt{,}b[\texttt{()}]$ are values, whilst $a[u]\texttt{,()}$ or $u\texttt{,}a[\texttt{()}]$ are not values. Variables are either channels or represent placeholders to be instantiated with other values. Channels play an operational role similar to that played by URIs when the latter are used as endpoints for communicating with web services. For example

$$\mathit{message}[\mathit{title}[\texttt{()}], \mathit{chan}[u]]$$

is a value containing an empty title and a channel `u`. This channel could be used by the receiver, for instance, to send back an acknowledgment. In the following $a[\texttt{()}]$ is always shortened to $a[\,]$.

Schema. Schemas describe sets of values that are structurally similar. The schema `()` describes the value `()`; $\langle S \rangle$ describes channels that carry messages of schema S; $L[S]$ describes the values $a[V]$ such that $a \in L$ and V is of schema S; $L[S]\texttt{,}S'$ describes values $a[V]\texttt{,}V'$ where $a[V]$ and V' are of schema $L[S]$ and S', respectively (when S' describes `()`, the schema $L[S]\texttt{,}S'$ also includes the values

described by $L[S]$); $S + S'$ describes values that are in S or in S'. The schema $\mathtt{U}$ describes the least set of values such that $\mathtt{U} = \mathbb{E}(\mathtt{U})$, where $\mathbb{E}$ maps schema names to schemas and fulfils the following finiteness property. Let $names(S)$ be the least set containing the schema names in S and such that if $\mathtt{U} \in names(S)$ then $names(\mathbb{E}(\mathtt{U})) \subseteq names(S)$. A map $\mathbb{E}$ is *finite* if, for every $\mathtt{U} \in \mathtt{dom}(\mathbb{E})$, the set $names(\mathtt{U})$ is finite. We notice that this property implies that `PiDuce` schemas define *tree regular languages* [17].

For example, the schema that collects booleans is $\mathtt{true}[()] + \mathtt{false}[()]$; the schema that collects an emptyset of values is `Empty` defined by $\mathtt{Empty} = \mathtt{Empty}$; the schema that collects any channel, no matter what it can carries, is $\langle \mathtt{Empty} \rangle$; the schema that collects all the values is `Any` defined by $\mathtt{Any} = () + \mathtt{\sim}[\mathtt{Any}], \mathtt{Any} + \langle \mathtt{Empty} \rangle$. We observe that $\langle \mathtt{Empty} \rangle$ and $\langle \mathtt{Any} \rangle$ are very different. $\langle \mathtt{Empty} \rangle$ refers to any channels, $\langle \mathtt{Any} \rangle$ refers only to channels that can carry arbitrary data. For instance $\langle a[()] \rangle$ is an $\langle \mathtt{Empty} \rangle$ but not an $\langle \mathtt{Any} \rangle$. As for values, the schema $()$ in $L[()]$ will be always omitted.

`PiDuce` channel schemas $\langle S \rangle$ are such that S is *labelled-determined*, according to the next definition. Let μ range over internal schema representations $()$, $\langle\rangle(S)$, $L(S\ ;\ T)$ and let $S \downarrow \mu$, read S *has a handle* μ, be the least relation such that:

$() \downarrow ()$	
$\langle S \rangle \downarrow \langle\rangle(S)$	
$L[S] \downarrow L(S\ ;\ ())$	if $\widehat{L} \neq \emptyset$ and there is μ such that $S \downarrow \mu$
$L[S], T \downarrow L(S\ ;\ T)$	if there are μ, μ' such that $L[S] \downarrow \mu$ and $T \downarrow \mu'$
$S + T \downarrow \mu$	if $S \downarrow \mu$ or $T \downarrow \mu$
$\mathtt{U} \downarrow \mu$	if $\mathbb{E}(\mathtt{U}) \downarrow \mu$

We observe that `Empty` has no handle. The schema $a[\,], \mathtt{Empty}$ has no handle as well; the reason is that a sequence has an handle provided that every element of the sequence has an handle. We also remark that a channel $\langle S \rangle$ always retains an handle. A schema S is *not-empty* if and only if S has a handle; it is *empty* otherwise.

Definition 1. *The set* `ldet` *of* labelled-determined schemas *is the least set containing empty schemas and such that:*

1. $() \in \mathtt{ldet}$*;*
2. *if* $S \in \mathtt{ldet}$ *then* $\langle S \rangle \in \mathtt{ldet}$ *and* $L[S] \in \mathtt{ldet}$*;*
3. *if* $S \in \mathtt{ldet}$ *and* $T \in \mathtt{ldet}$ *then* $L[S], T \in \mathtt{ldet}$*;*
4. *if* $S \in \mathtt{ldet}$ *and* $T \in \mathtt{ldet}$ *and, for every* $S \downarrow L(S'\ ;\ S'')$ *and* $T \downarrow L'(T'\ ;\ T'')$, $L \cap L' = \emptyset$ *then* $S + T \in \mathtt{ldet}$*;*
5. *if* $\mathbb{E}(\mathtt{U}) \in \mathtt{ldet}$ *then* $\mathtt{U} \in \mathtt{ldet}$*.*

For example, $a[S] + (\mathtt{\sim} \setminus a)[T]$ and $\mathtt{\sim}[S] + \langle S \rangle + \langle T \rangle$ are labelled-determined schemas. The schemas $a[\,] + (a + b)[\,]$ and $\langle a[\,] + \mathtt{\sim}[\,] \rangle$ are not labelled-determined.

We observe that the labelled-determinedness restriction will be applied to channel schemas only. For instance, $a[\,] + \mathtt{\sim}[\,]$ is a legal `PiDuce` schema. We also recall that labelled-determinedness and the syntactic constraint that channel

schemas always occur in tail positions of sequences entail a polynomial algorithm for language inclusion of channel schemas [8]. As discussed in the introduction, this is fundamental for an efficient pattern matching algorithm (see also Section 3).

Patterns. Patterns allow the deconstruction of values using matching. The pattern S is matched by values of schema S. A variable $u : F$ can be bound in the course of matching to a value described by the schema represented by F (see below). For example, $u : a[\,] + b[\,]$ may bind values such as $a[\,]$ or $b[\,]$; $u : \langle a[\,]\rangle$ may bind any channel value of schema $\langle a[\,]\rangle$, but it does not bind a channel value of schema $\langle a[\,] + b[\,]\rangle$; the pattern $u : a[v : b[\,]]$ binds two variables at the same time: u to values of schema $a[b[\,]]$ and v to values of schema $b[\,]$. The pattern $L[F]$ is a shorthand for $L[F]\texttt{,()}$; $L[F]\texttt{,}F'$ is matched by values of the form $a[V]\texttt{,}V'$, with $a \in L$ and V and V' match with F and F', respectively. The pattern $L[F]\texttt{,}F'$ is also matched by values $a[V]$ if $a[V]$ matches with $L[F]$ and `()` matches with F'.

The *schema represented* by a pattern F, in notation $\texttt{schof}(F)$ is defined inductively by

$$\begin{aligned}
\texttt{schof}(S) &= S \\
\texttt{schof}(u : F) &= \texttt{schof}(F) \\
\texttt{schof}(L[F]) &= L[\texttt{schof}(F)] \\
\texttt{schof}(L[F]\texttt{,}F') &= L[\texttt{schof}(F)]\texttt{,}\texttt{schof}(F')
\end{aligned}$$

`PiDuce` patterns retain the following *well-formedness* properties:

1. *linearity with respect to variables*: variables occurring in a pattern do not clash;
2. *not-emptiness*: patterns F are such that $\texttt{schof}(F)$ is always not-empty. (Patterns such as $u : \texttt{Empty}$ are excluded because they are practically useless and theoretically troublesome (see Proposition 2 and Theorem 1).)

Processes. Processes define the set of computing entities in `PiDuce`. $\mathbf{0}$ is the idle process; $\overline{u}[V]$ outputs the value V on the channel u; $\sum_{i\in I} u_i(F_i).P_i$ inputs on the channel u_i a value that matches with F_i yielding a substitution σ and behaves as $P\sigma$. The family I in the input choice is assumed to be finite. The process $(u{:}\langle S\rangle)P$ defines a new channel u and binds it within the continuation P; $!P$ is the process that always spawns copies of P. The channels u in $\overline{u}[V]$ and u_i in $\sum_{i\in I} u_i(F_i).P_i$ are called *subjects* of output and input, respectively.

As it is usual in concurrent calculi, the choice in the process $\sum_{i\in I} u_i(F_i).P_i$ is unordered. For example, the process

$$\begin{aligned}
&\mathit{print}(u : \texttt{File}).\ \overline{\mathit{printbw}}[u] \\
&+\ \mathit{print}(u : \texttt{Picture}).\ \overline{\mathit{printc}}[u] \\
&+\ \mathit{print}(u : \texttt{Any}).\ \overline{\mathit{error\text{-}handler}}[u]
\end{aligned}$$

takes a value on the channel `print`, and forwards it either to the black and white printer if it matches with `File`, or to the color printer if it matches with `Picture`,

or to an error-handler, otherwise (`File` and `Picture` are two schema names). However it may be the case that the branch $print(u : \texttt{Any}).\overline{error\text{-}handler}[u]$ is picked, even if a file or a picture is received because such values also match with `Any`. Said otherwise, input choices badly express a standard pattern matching mechanism of programming languages, the first-matching semantics. When `PiDuce` is turned into a real concurrent programming language it should be extended with a native first-matching mechanism (indeed, this is the case in [9]).

`PiDuce` processes retain the *output capability* property: every channel received in input may be used in the continuation either as subject of outputs, or within values.

Free and bound variables. The set $\texttt{fv}(\cdot)$ is defined for values, patterns, and processes as follows:

$\texttt{fv}(V)$**:** is the set of variables occurring in V;
$\texttt{fv}(F)$**:** is the set of variables occurring in F;
$\texttt{fv}(P)$**:** is the set of variables occurring in P and, recursively, in the bodies of the constant invocations, that are not *bound*. An occurrence of x in P is *bound* in the input $u(F).P$ if $x \in \texttt{fv}(F)$; an occurrence of u in P is *bound* in $(u\!:\!S)P$. Bound variables are noted $\texttt{bv}(\cdot)$.

The definitions of α-conversion and substitution for bound variables are standard.

Design remarks. The design of the `PiDuce` schema and pattern languages, as well as most of the algorithms regarding these features, has been strongly influenced by the `XDuce` [12] and `CDuce` [5] prototypes. The differences are discussed in order.

`PiDuce` schemas omit primitive schemas, such as `integer` and `string`, but these may be added without any difficulty. For simplicity sake, schemas such as $(S + T)\texttt{,}R$ are also excluded, while they are admitted by the formalism in [13]: such schemas would have entangled the definition of labelled-determinedness. A major departure with respect to the schema in the above languages (and in `XML` DTDs or in `XML` Schema) is entailed by the channel schemas, which are new. These schemas are used in `PiDuce` to verify the correct use of channels.

In `PiDuce` processes, channels that are received in input can be used in the continuations only with *output capability*. Such output capability means that a process receiving a channel cannot redefine that channel by accepting additional inputs meant for that channel (a reasonable constraint in web services). Output capability plays an important role in the static semantics because it entails subtyping polymorphism in schemas that include channel schemas.

3 The static semantics

We begin with the basic notion of subschema; the static semantics of values and of processes are defined afterwards.

Subschema. The semantic definition of subschema in [12] does not adapt well to `PiDuce`. In particular, since values contain channels, in order to verify that a channel u belongs to a schema S, one is reduced to verifying that the schema of u is a subschema of S. To circumvent this circularity we use an "operational" definition – a *simulation* relation – in the style of [3, 20].

The subschema relation uses handles defined in Section 2 to manifest all the branches of the syntax tree of a schema and to get rid of useless `()` elements. In the following definition we use the intersection operator on labels: $L \cap L' \stackrel{\text{def}}{=} \texttt{~} \setminus ((\texttt{~} \setminus L) + (\texttt{~} \setminus L'))$.

Definition 2. *The* subschema relation $\leq:$ *is the largest relation on schemas such that* $S \leq: T$ *implies:*

1. *if* $S \downarrow \texttt{()}$ *then* $T \downarrow \texttt{()}$*;*
2. *if* $S \downarrow \langle\rangle(S')$ *then* $T \downarrow \langle\rangle(T')$ *and* $T' \leq: S'$*;*
3. *if* $S \downarrow L(S' \,;\, S'')$ *then* $T \downarrow L'(T' \,;\, T'')$ *with* $\widehat{L} \cap \widehat{L'} \neq \emptyset$ *and:*
   (a) *either* $\widehat{L} \subseteq \widehat{L'}$*,* $S' \leq: T'$*, and* $S'' \leq: T''$*;*
   (b) *or* $(L \setminus L')[S'], S'' + (L \cap L')[R'], S'' + (L \cap L')[S'], R'' \leq: T$*, for some* R' *and* R'' *such that* $S' \leq: T' + R'$ *and* $S'' \leq: T'' + R''$*.*

We notice that, in the definition of $S \leq: T$, when $S \downarrow L(S' \,;\, S'')$ and T is labelled-determined, case *3.(b)* reduces to $(L \setminus L')[S'], S'' \leq: T$. For example it is easy to verify that $a[\,] \leq: a[\,] + b[\,]$ and $\langle a[\,] + b[\,]\rangle \leq: \langle a[\,]\rangle$. The definition of subschema allows us to derive that $c[a[\,] + b[\,]], (d[\,] + e[\,])$ is a subschema of $T = c[a[\,]], d[\,] + c[b[\,]], (d[\,] + e[\,]) + c[a[\,]], e[\,]$. In particular, since $T \downarrow c(a[\,]] \,;\, d[\,])$, one may reduce the problem to verifying that $c[R'], (d[\,] + e[\,]) + c[a[\,] + b[\,]], R'' \leq: T$ with $R' = b[\,]$ and $R'' = e[\,]$. The relationship $c[b[\,]], (d[\,] + e[\,]) \leq: T$ is immediate because it is the second addend of T. As regards $c[a[\,] + b[\,]], e[\,] \leq: T$ we observe that $T \downarrow c(b[\,]] \,;\, d[\,] + e[\,])$. This reduces to the verification of $c[a[\,]], e[\,] \leq: T$, which is true because $c[a[\,]], e[\,]$ is the third addend of T.

A few properties of $\leq:$ are in order.

Proposition 1. *1.* $\leq:$ *is reflexive and transitive;*
2. (Contravariance of $\langle\cdot\rangle$*)* $S \leq: T$ *if and only if* $\langle T\rangle \leq: \langle S\rangle$*;*
3. For every S*,* $\texttt{Empty} \leq: S \leq: \texttt{Any}$ *and* $\langle\texttt{Any}\rangle \leq: \langle S\rangle \leq: \langle\texttt{Empty}\rangle$*.*

The computational complexity of the subschema relation. Let $\|S\|$ be the *size* of the schema S, namely the size of the syntax tree of S, plus the sizes of the syntax trees of $\mathbb{E}(\texttt{U})$ such that $\texttt{U} \in \mathit{names}(S)$. The computational complexity of $S \leq: T$ is exponential in $\|S\|$ and $\|T\|$ ([17], chapter 1). When S and T are labelled-determined schemas that do not contain channel constructors, there is an algorithm (using tree automata) for computing $S \leq: T$ whose computational complexity is $O(\|S\| \times \|T\|)$ ([17], chapter 1). But `PiDuce` schemas are not labelled-determined with respect to channel constructors. Nevertheless, the syntactic constraints on `PiDuce` channel schemas (channels may only occur in tail positions of sequences) yields a computational complexity for $S \leq: T$ of $O((\|S\| + \|T\|)^3)$. The algorithm uses two boolean tables of size $(\|S\| + \|T\|)^2$: the

table `testing` stores schemas that are being tested, and the table `test_false` stores schemas that have been tested and the algorithm has failed. At each step, which costs $O(\mathtt{max}(\|S\|, \|T\|))$, the algorithm stores `true` either in `testing` or in `test_false`, and the two tables never store `true` in the same position. Therefore there are at most $(\|S\| + \|T\|)^2$ iterations. A detailed discussion of the required algorithm can be found in [8].

Typing rules for values. An *environment* Γ is a finite partial map from variables to schemas. We write $\mathtt{dom}(\Gamma)$ for the set of variables in Γ; we write $\emptyset$ for the *empty environment*, and $u : S$ for the singleton map. We let $\Gamma + \Gamma'$ be $(\Gamma \setminus \mathtt{dom}(\Gamma')) \cup \Gamma'$, where $\Gamma \setminus X$ removes from Γ all the bindings of names in X. The environments we consider in the following are always well-formed: Γ is *well-formed* if, for every u, $\Gamma(u)$ is not-empty.

The judgment $\Gamma \vdash V : S$ – read V has type S in the environment Γ – is the least relation satisfying the following rules:

$$\text{(EMPTY)}\ \Gamma \vdash () : () \qquad \text{(VAR)}\ \frac{\Gamma(u) = S}{\Gamma \vdash u : S} \qquad \text{(LAB)}\ \frac{\Gamma \vdash V : S}{\Gamma \vdash a[V] : a[S]} \qquad \text{(SEQ)}\ \frac{\Gamma \vdash V : S \quad \Gamma \vdash V' : S'}{\Gamma \vdash a[V], V' : a[S], S'}$$

These rules and the well-formedness of environments entail the following properties.

Proposition 2. *1. Let $\Gamma \vdash V : S$ and $S <: \langle T \rangle$, for some T. Then V is a variable and S is a channel schema.*

2. Let Γ be such that, for every $u \in \mathtt{dom}(\Gamma)$, $\Gamma(u)$ is labelled-determined. If $\Gamma \vdash V : S$ then S is labelled-determined.

We observe that Proposition 2.1 would be false if Γ were not well-formed. For instance take $\Gamma = u : \mathtt{Empty}$ and $V = a[u]$; then the hypotheses hold, but the conclusion is false. Proposition 2.2 is relevant because, at run-time, environments map variables to channel schemas. Henceforth communicated values always retain labelled-determined schemas.

Typing rules for processes. Let $\mathtt{Env}(F)$ be the following function taking a pattern and giving an environment:

$$\begin{aligned} \mathtt{Env}(S) &= \emptyset \\ \mathtt{Env}(u : F) &= u : \mathtt{schof}(F) + \mathtt{Env}(F) \\ \mathtt{Env}(L[F]) &= \mathtt{Env}(F) \\ \mathtt{Env}(L[F], F') &= \mathtt{Env}(F) + \mathtt{Env}(F') \end{aligned}$$

It is worth noticing that, in $u : \mathtt{schof}(F) + \mathtt{Env}(F)$ and $\mathtt{Env}(F) + \mathtt{Env}(F')$, the summands have disjoint domains, due to the linearity constraint on patterns.

The judgment $\Gamma \vdash P$ – read P is well typed in the environment Γ – is the least relation satisfying the following rules:

$$\frac{}{\Gamma \vdash \mathbf{0}}\ (\textsc{nil}) \qquad \frac{\Gamma \vdash u : \langle S \rangle \quad \Gamma \vdash V : T \quad T <: S}{\Gamma \vdash \overline{u}[V]}\ (\textsc{out})$$

$$\frac{\left(\begin{array}{c}\Gamma \vdash u_i : \langle S_i \rangle \qquad \Gamma + \mathtt{Env}(F_i) \vdash P_i \\ S_i <: \sum_{j \in I,\ u_j = u_i} \mathtt{schof}(F_j)\end{array}\right)^{i \in I}}{\Gamma \vdash \sum_{i \in I} u_i(F_i).P_i}\ (\textsc{inp})$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P'}{\Gamma \vdash P \mid P'}\ (\textsc{par}) \qquad \frac{\Gamma + u : \langle S \rangle \vdash P}{\Gamma \vdash (u : \langle S \rangle) P}\ (\textsc{new}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash\ !P}\ (\textsc{rep})$$

Rules (NIL), (PAR), (NEW), and (REP) are standard. Rule (OUT) types outputs. The premise $T <: S$ entails that the subject of the input is a channel. It is worth noticing that, if $S = \mathtt{Empty}$, then there is no T such that $\Gamma \vdash V : T$ and $T <: \mathtt{Empty}$. Therefore outputs on channels of schema `Empty` are forbidden. On the contrary, if $S = \mathtt{Any}$, then $\overline{u}[V]$ is always well typed. Rule (INP) types input guarded choices. The second hypothesis is easy to explain: it enforces the typing of the continuation of every summand in the environment Γ plus that defined by the pattern. The third hypothesis may be understood as follows. Assume there are exactly n summands inputting on the same channel u, and let $F_1, \cdots, F_n$ be the corresponding patterns. Let $\langle S \rangle$ be the schema of u. Then the hypothesis establishes that $S <: \mathtt{schof}(F_1) + \cdots + \mathtt{schof}(F_n)$. It is worth recalling that $\mathtt{schof}(F_1) + \cdots + \mathtt{schof}(F_n)$ is in general not labelled-determined. This allows programmers to write processes such as $x(v : a[\mathtt{Int}]).P + x(v : a[\mathtt{String}]).Q$. We also remark that $\Gamma + \mathtt{Env}(F_i)$ is well-formed because of the not-emptiness restriction of patterns.

4 The operational semantics

This section defines the semantics of patterns and processes. In order to cope with values that may carry (channel) variables, both the pattern matching and the transition relations take an associated environment into account.

Patterns. Patterns decompose the structure of values and select parts of them. The algorithm, called *pattern-matching*, yields a substitution in case the decomposition succeeds. Let σ and σ' be two substitutions with disjoint domains. We write $\sigma + \sigma'$ to denote the function that is the union of σ and σ'. Every union in the following rules is always well defined because of the linearity constraint placed on patterns.

The *pattern match* of a value V with respect to a pattern F in an environment Γ, written $\Gamma \vdash V \in F \rightsquigarrow \sigma$, is the least relation satisfying the following rules:

$$\frac{S \downarrow \texttt{()}}{\Gamma \vdash \texttt{()} \in S \rightsquigarrow \emptyset}\ (\textsc{emp}) \qquad \frac{u \in \texttt{dom}(\Gamma) \quad \Gamma(u) <: S}{\Gamma \vdash u \in S \rightsquigarrow \emptyset}\ (\textsc{var})$$

$$\frac{S \downarrow L(S' \,;\, S'') \quad a \in L \quad \Gamma \vdash V \in S' \rightsquigarrow \emptyset \quad \Gamma \vdash \texttt{()} \in S'' \rightsquigarrow \emptyset}{\Gamma \vdash a[V] \in S \rightsquigarrow \emptyset}\ (\textsc{label-void})$$

$$\frac{S \downarrow L(S' \,;\, S'') \quad a \in L \quad \Gamma \vdash V \in S' \rightsquigarrow \emptyset \quad \Gamma \vdash V'' \in S'' \rightsquigarrow \emptyset}{\Gamma \vdash a[V]\texttt{,}V'' \in S \rightsquigarrow \emptyset}\ (\textsc{seq-void})$$

$$\frac{\Gamma \vdash V \in F \rightsquigarrow \sigma}{\Gamma \vdash V \in u : F \rightsquigarrow [u \mapsto V] + \sigma}\ (\textsc{pat-var})$$

$$\frac{\texttt{fv}(L[F]\texttt{,}F') \neq \emptyset \quad a \in L \quad \Gamma \vdash V \in F \rightsquigarrow \sigma \quad \Gamma \vdash \texttt{()} \in F' \rightsquigarrow \sigma'}{\Gamma \vdash a[V] \in L[F]\texttt{,}F' \rightsquigarrow \sigma + \sigma'}\ (\textsc{label}) \qquad \frac{\texttt{fv}(L[F]\texttt{,}F') \neq \emptyset \quad a \in L \quad \Gamma \vdash V \in F \rightsquigarrow \sigma \quad \Gamma \vdash V' \in F' \rightsquigarrow \sigma'}{\Gamma \vdash a[V]\texttt{,}V' \in L[F]\texttt{,}F' \rightsquigarrow \sigma + \sigma'}\ (\textsc{seq})$$

We discuss (VAR), (PAT-VAR), and (SEQ). Rule (VAR) verifies if a variable value matches with a schema S: it reduces to verifying the subschema relation between the schema of the variable and S. Rule (PAT-VAR) defines a substitution of a variable in a pattern. Rule (SEQ) matches a value with a sequence pattern. Since sequence patterns are linear with respect to variables, the substitution $\sigma + \sigma'$ is always well defined. We notice that no rule is defined for the pattern $L[F]$: this pattern is always rewritten into $L[F]\texttt{,()}$.

When the matched value does not contain variables, the pattern matching algorithm has a cost that is proportional to the product of the size of the value and the size of the pattern. This is because the schema of the pattern is not labelled-determined; if such a schema was labelled-determined then the cost should have been linear with respect to the size of the value. The presence of variables in values reduces the pattern matching to the subschema relation, see the right premise of (VAR), which has an exponential cost, in general. However, if every variable in the environment Γ has a channel schema – that is always the case at run-time, see the following transition relation – then the subschema relation reduces to computing the subschema between labelled-determined schemas, which has a polynomial cost. More precisely, in the premise $\Gamma(u) <: S$ of (VAR), $\Gamma(u) = \langle T \rangle$, for some T. In order to match the pattern, one has to verify that T is greater than one schema in $\{S' \mid \langle S \rangle \downarrow \langle\rangle S'\}$. Then (1) every such check is polynomial because the schemas are labelled-determined, and (2) there are at most $\|S\|$ checks because the cardinality of $\{S' \mid S \downarrow \langle\rangle S'\}$ is smaller.

Proposition 3. *Let $\Gamma \vdash V : S$ and $\Gamma \vdash V \in F \rightsquigarrow \sigma$. Then $S <: \texttt{schof}(F)$.*

Processes. Let $\mu, \nu, \cdots$ range over τ, input $u(F)$, and bound output $(\Gamma)\overline{u}[V]$ such that $\mathtt{dom}(\Gamma) \subseteq \mathtt{fv}(V)$. The bound output $(\Gamma)\overline{u}[V]$ is shortened to $\overline{u}[V]$ when $\Gamma = \emptyset$. We use the following auxiliary functions:

fv: $\mathtt{fv}(\tau) = \emptyset$, $\mathtt{fv}(u(F)) = \{u\}$, and $\mathtt{fv}((\Gamma)\overline{u}[V]) = (\{u\} \cup \mathtt{fv}(V)) \setminus \mathtt{dom}(\Gamma)$.
bv: $\mathtt{bv}((\Gamma)\overline{u}[V]) = \mathtt{dom}(\Gamma)$, $\mathtt{bv}(u(F)) = \mathtt{fv}(F)$, and $\mathtt{bv}(\tau) = \emptyset$.

The *transition relation* of PiDuce, $\xrightarrow{\mu}$, is the least relation satisfying the rules:

$$\Gamma \vdash \overline{u}[V] \xrightarrow{\overline{u}[V]} \mathbf{0} \qquad \Gamma \vdash \textstyle\sum_{i\in I} u_i(F_i).P_i \xrightarrow{u_i(F_i)} P_i$$

$$\frac{\Gamma + u\!:\!\langle S\rangle \vdash P \xrightarrow{\mu} Q \quad u \notin \mathtt{fv}(\mu) \cup \mathtt{bv}(\mu)}{\Gamma \vdash (u\!:\!\langle S\rangle)P \xrightarrow{\mu} (u\!:\!\langle S\rangle)Q} \qquad \frac{\Gamma + v\!:\!\langle S\rangle \vdash P \xrightarrow{(\Gamma')\overline{u}[V]} Q \quad v \neq u \quad v \in \mathtt{fv}(V) \setminus \mathtt{dom}(\Gamma')}{\Gamma \vdash (v\!:\!\langle S\rangle)P \xrightarrow{(\Gamma' + v:\langle S\rangle)\overline{u}[V]} Q}$$

$$\frac{\Gamma \vdash P \xrightarrow{\mu} P' \quad \mathtt{bv}(\mu) \cap \mathtt{fv}(Q) = \emptyset}{\Gamma \vdash P \mid Q \xrightarrow{\mu} P' \mid Q}$$

$$\frac{\Gamma \vdash P \xrightarrow{(\Gamma')\overline{u}[V]} P' \quad \Gamma \vdash Q \xrightarrow{u(F)} Q' \quad \mathtt{dom}(\Gamma') \cap \mathtt{fv}(Q) = \emptyset \quad \Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma}{\Gamma \vdash P \mid Q \xrightarrow{\tau} (\Gamma')(P' \mid Q'\sigma)} \qquad \frac{\Gamma \vdash P \xrightarrow{\mu} Q \quad \mathtt{bv}(\mu) \cap \mathtt{fv}(P) = \emptyset}{\Gamma \vdash !P \xrightarrow{\mu} Q \mid !P}$$

plus the symmetric rules for parallel.

This transition relation is similar to that of the π-calculus [19], except for the environment Γ. This environment is partially supplied by enclosing news and partially by the global environment. In particular, bound outputs gather an environment. This means that a communication between a sender and a receiver also carries information about the schema of the variables in the message. In practice this is the case: a web service URI is always shipped with its WSDL document containing, for instance, the protocol that must be used to invoke the service. In case of PiDuce, this WSDL document also contains the schema of the service. However our semantics does not require that the schema information is sent *together with* the message. For example, a service receiving a message on a generic channel and forwarding it to another service does not need to know the schema of the received message. Hence a lazy implementation of schema requests is plausible. Such an implementation downloads schemas only if they are needed by the pattern matching algorithm in the communication and the invocation rules.

In practice, PiDuce processes have free variables that are channels. Such processes may be typed in environments mapping variables to channel schemas, let us call them *channel environments*. A relevant property of the transition relation is that if $\Gamma \vdash P \xrightarrow{\tau} Q$ and Γ is a channel environment, then the pattern matching in the communication rule – the last but one rule – still uses a channel environment. This is critical for computing the transition relation with a polynomial cost.

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

Theorem 1. *(Subject Reduction) Let $\Gamma \vdash P$. Then*

1. *if $\Gamma \vdash P \stackrel{(\Gamma')\overline{u}[V]}{\longrightarrow} Q$ then (a) $\Gamma + \Gamma' \vdash Q$, and (b) $\Gamma \vdash u : \langle S \rangle$ and $\Gamma + \Gamma' \vdash V : T$ with $T <: S$;*
2. *if $\Gamma \vdash P \stackrel{u(F)}{\longrightarrow} Q$ then $\Gamma + \mathtt{Env}(F) \vdash Q$ and $\Gamma \vdash u : \langle S \rangle$;*
3. *if $\Gamma \vdash P \stackrel{\tau}{\longrightarrow} Q$ then $\Gamma \vdash Q$.*

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. In order to guarantee progress, it is necessary to restrict (well-formed) environments. To illustrate the problem, consider the following judgment:

$$u : \langle \mathtt{Int} + \mathtt{String} \rangle, v : \mathtt{Int} + \mathtt{String} \vdash \overline{u}[v] \mid (u(x : \mathtt{Int}).\mathbf{0} + u(x : \mathtt{String}).\mathbf{0})$$

The reader may verify that it is derived by our type system, however no interaction may occur because the schema of the value v is neither a subschema of $\mathtt{schof}(x : \mathtt{Int})$ or of $\mathtt{schof}(x : \mathtt{String})$. In fact this circumstance never occurs in practice: if a value is sent, the unique variables it may contain are channels. Under this constraint, progress is always guaranteed. A similar remark may be made for pattern matching. For generic environments Γ it may be the case that $\Gamma \vdash V : S$ and $S <: \mathtt{schof}(F)$ but there is no σ such that $\Gamma \vdash V \in F \rightsquigarrow \sigma$. Consider for instance $\Gamma = u : a[b[\,]]$, $V = u$, and $F = a[v : b[\,]]$.

Let $P \stackrel{\mu}{\longrightarrow}$ be an abbreviation for "there exists Q such that $P \stackrel{\mu}{\longrightarrow} Q$.

Theorem 2. *(Progress) Let Γ be such that, for every $u \in \mathtt{fv}(V)$, $\Gamma(u)$ is a channel schema.*

1. *If $\Gamma \vdash V : S$ and $S <: T + R$ then either $S <: T$ or $S <: R$.*
2. *If $\Gamma \vdash V : S$ and $S <: \mathtt{schof}(F)$ then there is σ such that $\Gamma \vdash V \in F \rightsquigarrow \sigma$.*
3. *If $\Gamma \vdash P$, $P \stackrel{(\Gamma')\overline{u}[V]}{\longrightarrow}$ and $P \stackrel{u(F)}{\longrightarrow}$ then there are F' and σ such that $P \stackrel{u(F')}{\longrightarrow}$ and $\Gamma + \Gamma' \vdash V \in F' \rightsquigarrow \sigma$.*

5 Conclusions and ongoing research

`PiDuce` is a process language with native `XML` datatypes and operators for constructing and deconstructing `XML` documents. It has been designed for modeling applications that are intrinsically concurrent, such as web services orchestrations and choreographies.

In this paper we have focussed on the theory of `PiDuce`. In order to reduce the computational complexity of the pattern matching algorithm, channel schemas have been constrained to be labelled-determined and to occur in the

tail positions of sequences. These constraints guarantee a polynomial computational complexity of the subschema relation, a mechanism used in the pattern matching.

The `PiDuce` schema language has been designed with a bias towards simplicity. A number of design choices may be changed without affecting the computational complexity of the subschema relation. A thorough analysis of variants of `PiDuce` schemas and their expressive power is left to future investigations.

Most of our current efforts are in prototyping a distributed implementation of `PiDuce`. As a matter of facts, `PiDuce` appears to be a basic computational model of current orchestration and choreography languages of services in the web, such as `BizTalk` [22], `WSFL` [16], `WS-CDL` [14], and `Bpel4ws` [4]. The prototype [9], is intended to serve as a distributed virtual machine for a these languages A significant next step will be the extension of `PiDuce` with a transactional operator, possibly along the lines of [7, 15]. Such an extension will be used to compile technologies such as BizTalk or `Bpel4ws`.

Acknowledgments. The authors thank Samuele Carpineti, David Richter, and Lucian Wischik for the interesting discussions and for having spotted several errors in previous drafts of this paper.

References

1. S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-peer data and Web services integration. In *VLDP 2002: Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases, 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. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
4. T. Andrews and et.al. Business Process Execution Language for Web Services. Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems, 2003.
5. 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. ACM Press, 2003.
6. K. Bhargavan, C. Fournet, A. Gordon, and R. Pucella. Tulafale: A security tool for web services. In *Formal Methods for Components and Objects (FMCO 2003)*, volume 3188 of *LNCS*, pages 197–222. Springer, 2004.
7. R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in join calculus. In *13th International Conference on Concurrency Theory (CONCUR'02)*, volume 2421 of *LNCS*, pages 321–337. Springer, 2002.
8. S. Carpineti and C. Laneve. A rude contract language for web services. Extended Abstract at `www.cs.unibo.it/BoPi`, 2005.
9. S. Carpineti, C. Laneve, and P. Milazzo. BoPi: a distributed machine for experimenting web services technologies. In *5th International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 202–212. IEEE Press, 2005.

10. 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.
11. P. Gardner and S. Maffeis. Modelling dynamic web data. In *9th International Workshop on Database Programming Languages (DBPL'03)*, volume 2921 of *Lecture Notes in Computer Science*, pages 130 – 146. Springer-Verlag, 2003.
12. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.
13. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22. ACM Press, 2000.
14. N. Kavantzas, G. Olsson, J. Mischkinsky, and M. Chapman. Web Services Choreography Description Languages. Oracle Corporation, 2003.
15. C. Laneve and G. Zavattaro. Foundations of web transactions. In *Foundations of Software Science and Computation Structures (FOSSACS'05)*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.
16. F. Leymann. Web Services Flow Language (wsfl 1.0). Technical report, IBM Software Group, 2001.
17. D. Lugiez, F. Jacquemard, H. Comon, M. Tommasi, M. Dauchet, R. Gilleron, and S. Tison. Tree automata techniques and applications. 2002.
18. R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
19. 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.
20. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.
21. D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996. ECS-LFCS-96-345.
22. S. Woodgate, S. Mohr, and B. Loesgen. *Microsoft BizTalk Server 2004 Unleashed*. Sams, 2004.

A Soundness of the static semantics

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

Lemma 1. *(Weakening) If $\Gamma \vdash P$ and $u \notin \mathtt{fv}(P)$ then $\Gamma + u : S \vdash P$. Similarly for $\Gamma \vdash V : S$.*

Lemma 2. *(Substitution)*

1. *Let $\Gamma + u : S \vdash V : T$ and $\Gamma \vdash V' : S'$ such that $S' <: S$. Then $\Gamma \vdash V\{^{V'}/_{u}\} : T'$ with $T' <: T$.*
2. *Let $\Gamma + u : S \vdash P$ and the free occurrences of u in P are not subjects of inputs. If $\Gamma \vdash V : T$, and $T <: S$ then $\Gamma \vdash P\{^{V}/_{u}\}$.*

Proof. The demonstration is by induction on the structures of the proofs of $\Gamma + u : S \vdash V : T$ and $\Gamma + u : S \vdash P$. We only discuss the case when the last

rule is (OUT). Then $P = \overline{w}[V']$ and the premises of the rule are the judgments $\Gamma + u : S \vdash w : \langle R \rangle$ and $\Gamma + u : S \vdash V' : S'$, and the predicate

$$S' <: R \tag{1}$$

We must prove $\Gamma \vdash \overline{w}[V']\{^V/_u\}$. By $\Gamma + u : S \vdash V' : S'$, the hypothesis $\Gamma \vdash V : T$, $T <: S$, and the substitution lemma for values, we obtain

$$\Gamma \vdash V'\{^V/_u\} : T' \tag{2}$$
$$T' <: S' \tag{3}$$

As regards the subject of the output, there are two subcases: (a) $u \neq w$ and (b) $u = w$. Case (a) follows by (1), (3) and transitivity of $<:$. Case (b) implies $S = \langle R \rangle$ and, by Proposition 2, V is a variable and T is a channel schema. Let V be u'. By the substitution lemma for values

$$\Gamma \vdash u' : \langle T'' \rangle \tag{4}$$

and $\langle T'' \rangle <: \langle R \rangle$. Then $\Gamma \vdash \overline{w}[V']\{^V/_u\}$ follows by (2), (4), the relations $\langle T'' \rangle <: \langle R \rangle$, (1), (3), the controvariance of $\langle \cdot \rangle$ and the transitivity of $<:$. □

The soundness of pattern matching is established by the next lemma.

Lemma 3. *(Pattern Matching)*

1. *If $\Gamma \vdash V \in F \rightsquigarrow \sigma$ and $u \notin \mathtt{fv}(V)$ then $\Gamma + u : T \vdash V \in F \rightsquigarrow \sigma$.*
2. *If $\Gamma \vdash V : S$ and $\Gamma \vdash V \in F \rightsquigarrow \sigma$ then, for every $u \in \mathtt{fv}(F)$, $\Gamma \vdash \sigma(u) : T$ and $T <: \mathtt{Env}(F)(u)$.*

The preliminaries are in place for the subject reduction theorem.

Theorem 1. *(Subject Reduction) Let $\Gamma \vdash P$. Then*

1. *if $\Gamma \vdash P \overset{(\Gamma')\overline{u}[V]}{\longrightarrow} Q$ then $\Gamma + \Gamma' \vdash Q$, $\Gamma \vdash u : \langle S \rangle$ and $\Gamma + \Gamma' \vdash V : T$ with $T <: S$;*
2. *if $\Gamma \vdash P \overset{u(F)}{\longrightarrow} Q$ then $\Gamma + \mathtt{Env}(F) \vdash Q$ and $\Gamma \vdash u : \langle S \rangle$;*
3. *if $\Gamma \vdash P \overset{\tau}{\longrightarrow} Q$ then $\Gamma \vdash Q$.*

Proof. The demonstration proceeds by induction on the structure of the proof of $\Gamma \vdash P \overset{\mu}{\longrightarrow} Q$ and by cases on the last rule that has been applied. We only detail the case of the communication rule

$$\frac{\Gamma \vdash P \overset{(\Gamma')\overline{u}[V]}{\longrightarrow} P' \quad \Gamma \vdash Q \overset{u(F)}{\longrightarrow} Q' \quad \mathtt{dom}(\Gamma') \cap \mathtt{fv}(Q) = \emptyset \quad \Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma}{\Gamma \vdash P \mid Q \overset{\tau}{\longrightarrow} (\Gamma')(P' \mid Q'\sigma)}$$

Since $\Gamma \vdash P \mid Q$, the premises of (PAR) give $\Gamma \vdash P$ and $\Gamma \vdash Q$. By $\mathtt{dom}(\Gamma') \cap \mathtt{fv}(Q) = \emptyset$ and Lemma 1, $\Gamma + \Gamma' \vdash Q'$. By inductive hypotheses on $\Gamma \vdash P \overset{(\Gamma')\overline{u}[V]}{\longrightarrow} P'$ and $\Gamma \vdash Q \overset{u(F)}{\longrightarrow} Q'$ we obtain:

$$\Gamma + \Gamma' \vdash P' \tag{5}$$
$$\Gamma + \Gamma' \vdash V : T \tag{6}$$
$$\Gamma \vdash u : S \tag{7}$$
$$\Gamma + \Gamma' + \mathtt{Env}(F) \vdash Q' \tag{8}$$

By Lemma 3(2) applied to $\Gamma + \Gamma' \vdash V \in F \rightsquigarrow \sigma$ and (6) we obtain that, for every $v \in \mathtt{fv}(F)$, $\Gamma + \Gamma' \vdash \sigma(v) : T'$ and $T' <: \mathtt{Env}(F)(v)$. By the substitution lemma applied to this last judgment and (8), we derive $\Gamma + \Gamma' \vdash Q'\sigma$. We conclude with (NEW): $\Gamma \vdash (\Gamma')(P' \mid Q'\sigma)$.

The theorem about progress is discussed below.

Theorem 2. *(Progress) Let Γ be such that, for every $u \in \mathtt{fv}(V)$, $\Gamma(u)$ is a channel schema.*

1. *If $\Gamma \vdash V : S$ and $S <: T + R$ then either $S <: T$ or $S <: R$.*
2. *If $\Gamma \vdash V : S$ and $S <: \mathtt{schof}(F)$ then there is σ such that $\Gamma \vdash V \in F \rightsquigarrow \sigma$.*
3. *If $\Gamma \vdash P$, $P \overset{(\Gamma')\overline{u}[V]}{\longrightarrow}$ and $P \overset{u(F)}{\longrightarrow}$ then there are F' and σ such that $P \overset{u(F')}{\longrightarrow}$ and $\Gamma + \Gamma' \vdash V \in F' \rightsquigarrow \sigma$.*

Proof. The proof of items 1 and 2 are simple and therefore omitted. As regards item 3, we consider the proof of $P \overset{u(F)}{\longrightarrow}$. By Theorem 1(2) applied to $\Gamma \vdash P$ and $P \overset{u(F)}{\longrightarrow}$ we obtain $\Gamma \vdash u : \langle S \rangle$. Then, we consider the proof tree of $P \overset{u(F)}{\longrightarrow}$. It must have an axiom (the leaf of the proof tree) whose shape is

$$\sum_{i \in I} u_i(F_i).P_i \overset{u(F)}{\longrightarrow}$$

Correspondingly, in the proof tree of $\Gamma \vdash P$ there is a judgment $\Gamma + \Gamma'' \vdash \sum_{i \in I} u_i(F_i).P_i$, for some Γ''. This judgment must have been proved with an instance of (INP) that yields $S <: \sum_{j \in I, u_j = u} \mathtt{schof}(F_i)$. By a similar argument applied to $P \overset{(\Gamma')\overline{u}[V]}{\longrightarrow}$, using (OUT), there are Γ''' and T such that $\Gamma + \Gamma''' + \Gamma' \vdash V : T$ and $T <: S$. Therefore, by transitivity of $<:$, $T <: \sum_{j \in I, u_j = u} \mathtt{schof}(F_i)$. By item 1(a), there exists F_k such that $T <: \mathtt{schof}(F_k)$. Additionally, by Lemmas 1 and 2, $\Gamma + \Gamma''' + \Gamma' \vdash V : T$ may be simplified into $\Gamma + \Gamma' \vdash V : T$. We conclude by item 1(b). □