Logical Methods in Computer Science
Vol. 4 (1:7) 2008, pp. 1–41
www.lmcs-online.org

Submitted Jan. 25, 2007
Published Mar. 21, 2008

ALGEBRAIC PATTERN MATCHING IN JOIN CALCULUS *

QIN MA [a] AND LUC MARANGET [b]

[a] OFFIS, Escherweg 2, 26121 Oldenburg, Germany
e-mail address: Qin.Ma@offis.de

[b] INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
e-mail address: Luc.Maranget@inria.fr

Abstract. We propose an extension of the join calculus with pattern matching on algebraic data types. Our initial motivation is twofold: to provide an intuitive semantics of the interaction between concurrency and pattern matching; to define a practical compilation scheme from extended join definitions into ordinary ones plus ML pattern matching. To assess the correctness of our compilation scheme, we develop a theory of the applied join calculus, a calculus with value passing and value matching. We implement this calculus as an extension of the current JoCaml system.

1. Introduction

The join calculus [15, 16] is a process calculus in the tradition of the π-calculus of Milner *et al.* [33]. One distinctive feature of join calculus is the simultaneous definition of all receptors on several channels through *join definitions*. A join definition is structured as a list of *reaction rules*, with each reaction rule being a pair of one *join pattern* and one *guarded process*. A join pattern is in turn a list of channel names (with formal arguments), specifying the synchronization among those channels: namely, a join pattern is matched only if there are messages present on all its channels. Finally, the reaction rules of one join definition define competing behaviors with a non-deterministic choice of which guarded process to trigger when several join patterns are satisfied.

In this paper, we extend the matching mechanism of join patterns, such that *message* contents are also taken into account. As an example, let us consider the following list-based implementation of a concurrent stack:[1]

$$\begin{array}{l}\textbf{def}\ \mathit{pop}(r)\ \&\ \mathit{State}(x::\mathit{xs}) \triangleright r(x)\ \&\ \mathit{State}(\mathit{xs})\\ \ \textbf{or}\ \mathit{push}(v)\ \&\ \mathit{State}(\mathit{ls}) \triangleright \mathit{State}\ (v::\mathit{ls})\\ \ \textbf{in}\ \mathit{State}(\texttt{[]})\ \&\ \ldots\end{array}$$

1998 ACM Subject Classification: D.1.3, D.3.3, F.3.2.
Key words and phrases: join-calculus, pattern-matching, process calculus, concurrency.
* Extended version of [28].
[1] We use the OCaml syntax for lists, with *Nil* being `[]` and *Cons* being the infix `::`.

The second join pattern $push(v)$ & $State(ls)$ is an *ordinary* one: it is matched whenever there are messages on both *State* and *push*. By contrast, the first join pattern is an *extended* one, where the formal argument of channel *State* is an *algebraic pattern*, matched only by messages that are cons cells. Thus, when the stack is empty (*i.e.*, when message `[]` is pending on channel *State*), pop requests are delayed. Note that we follow the convention that capitalized channels are private: only *push* and *pop* will be visible outside.

A similar stack can be implemented without using extended join patterns, but instead, using an extra private channel and ML pattern matching in guarded processes:

$$
\begin{array}{l}
\textbf{def}\ \mathit{pop}(r)\ \&\ \mathit{Some}(\mathit{ls})\ \triangleright \textbf{match}\ \mathit{ls}\ \textbf{with} \\
\qquad\qquad\qquad\qquad\qquad \mid\ \mathtt{[}x\mathtt{]} \rightarrow r(x)\ \&\ \mathit{Empty}() \\
\qquad\qquad\qquad\qquad\qquad \mid\ y::x::\mathit{xs} \rightarrow r(y)\ \&\ \mathit{Some}(x::\mathit{xs}) \\
\ \textbf{or}\ \mathit{push}(v)\ \&\ \mathit{Empty}() \triangleright \mathit{Some}\ (\mathtt{[}v\mathtt{]}) \\
\ \textbf{or}\ \mathit{push}(v)\ \&\ \mathit{Some}(\mathit{ls}) \triangleright \mathit{Some}\ (v::\mathit{ls}) \\
\ \textbf{in}\ \mathit{Empty}()\ \&\ \ldots
\end{array}
$$

This second definition encodes the empty/non-empty status of the stack as a message on channels *Empty* and *Some* respectively. Pop requests on an empty stack are still delayed, since there is no rule for the join pattern $pop(r)$ & $Empty()$. The second definition obviously requires more programming effort. Moreover, it is not immediately apparent that messages on *Some* are non-empty lists, and that the partial ML pattern matching thus never fails.

Join definitions with (constant) pattern arguments appear informally in functional nets [36]. Here we generalize this idea to full algebraic patterns. A similar attempt has also been scheduled by Benton *et al.* as an interesting future work for Cω [7].

The new semantics is a smooth extension, since both join pattern matching and pattern matching rest upon classical substitution (or semi-unification). However, an efficient implementation is more involved. Our idea is to address this issue by transforming programs whose definitions contain extended join patterns into equivalent programs whose definitions use ordinary join patterns and whose guarded processes use ML pattern matching. Doing so, we leave most of the burden of pattern matching compilation to an ordinary ML pattern matching compiler. However, such a transformation is far from obvious. More specifically, there is a gap between (extended) join pattern matching, which is non-deterministic, and ML pattern matching, which is deterministic (following the "first match policy"). For example, in our definition of a concurrent stack with extended join patterns, $State(ls)$ is still matched by any message on *State*, regardless of the presence of the more precise $State(x::xs)$ in the competing reaction rule that precedes it. Our solution to this problem relies on partitioning matching values into non-intersecting sets. In the case of our concurrent stack, those sets simply are the singleton $\{\mathtt{[]}\}$ and the set of non-empty lists. Then, pattern $State(ls)$ is matched by values from both sets, while pattern $State(x::xs)$ is matched only by values of the second set.

The rest of the paper is organized as follows: Section 2 first gives a brief review of algebraic patterns and ML pattern matching. Section 3 presents the applied join calculus — an extension of join with algebraic pattern matching. We introduce the semantics and the appropriate equivalence relations. Section 4 informally explains the key ideas to transform the extension to the ordinary join calculus, and especially how we deal with the nondeterminism problem. Section 5 formalizes the transformation as a compilation scheme and presents the algorithm which essentially works by building a meet semi-lattice of patterns. We go through a complete example in Section 6, and finally, we deal with the correctness of the compilation scheme in Section 7. Implementation has been carried out as an extension of the JoCaml system. We discuss the issues that have arisen during the implementation work in Section 8.

An earlier version of this paper (lacking the detailed proofs and the discussion of the implementation) appeared as [28].

2. Algebraic data types and ML pattern matching

This section serves as a brief introduction to algebraic data types and ML pattern matching. Interested readers are referred to [30, 26] for further details.

2.1. **Algebraic data types.** In functional languages, new types can be introduced by using *data type definitions* and such types are algebraic data types. For example, using OCaml syntax, binary trees can be defined as follows:

$$\textbf{type}\ \mathit{tree} = \mathit{Empty} \mid \mathit{Leaf}\ \textbf{of int} \mid \mathit{Node}\ \textbf{of}\ \mathit{tree} * \mathit{tree}$$

The *complete signature* of type *tree* has three *constructors*: *Empty*, *Leaf*, and *Node*, which are used to build the values of this type. Every constructor has an arity, *i.e.* the number of arguments it requires and meanwhile specifies the corresponding types of each argument. In this definition, *Empty* is of arity zero, *Leaf* is of arity one (and accepts integer arguments), and *Node* is of arity two (both its arguments being themselves of type *tree*). A constructor of zero arity is sometimes called a *constant constructor*.

Most native ML data types can be seen as particular instances of algebraic data types. For example, lists are defined by two constructors: constant *Nil* (written `[]`) for empty lists and *Cons* (written as the infix `::`) for nonempty ones; pairs are defined by one constructor with arity two, (written as the infix "`,`"); and integers are defined by infinitely many (or 2^{31}) constant constructors.

Formally, the algebraic values (for short values) of type t are well-typed terms built from the constructors of t. "Well-typed" here means correct with respect to constructor arity and argument types. Assuming a countable set of identifiers for constructors, ranged over by κ, we give the formal definition of values as follows:

$$\begin{array}{lll} v ::= & & \textbf{Algebraic values} \\ & \kappa(v_1, v_2, \ldots, v_n) & \kappa \text{ of arity } n \geq 0 \end{array}$$

Type correctness is left implicit: we shall consider well typed terms only.

Algebraic patterns (for short patterns) of type t are also well-typed terms built from the constructors of t, but with variables.[2] The formal definition of patterns is given as follows.

$$\begin{array}{lll} \pi ::= & & \textbf{Algebraic patterns} \\ & x & \text{variable} \\ & \kappa(\pi_1, \pi_2, \ldots, \pi_n) & \kappa \text{ of arity } n \geq 0 \end{array}$$

We further require all variables in a pattern to be pairwise distinct, that is, we only consider *linear* patterns.

Again, we assume a typed context. More precisely, we rely on the ML type system to guarantee that values and patterns are well-typed. Moreover, we rely on a ML type inferer to enrich syntax with explicit types (which we leave implicit), and consider that the type of any syntactic structure is available whenever needed. Doing so, we focus on our main issue and avoid complications that would be of little explanatory value.

[2] We freely replace variables whose names are of no importance by wildcards "`_`".

Patterns are used to discriminate values according to their structures. More specifically, a pattern denotes a set of values that have a common prefix specified by the pattern. We say a value v (of type t) is an *instance* of pattern π (of type t), or that v matches π, when π describes the prefix of v, in other words, when there exists a substitution σ, such that $\pi\sigma = v$. For linear patterns, the instance relation can be defined inductively as follows:

Definition 2.1 (Instance)**.** Let π be a pattern and v be a value, such that π and v have the same type, the instance relation $\pi \preceq v$ is defined as:

$$\begin{array}{rcll} \_ & \preceq & v & \\ \kappa(\pi_1,\ldots,\pi_n) & \preceq & \kappa(v_1,\ldots,v_n) & \text{iff } \pi_i \preceq v_i \text{ for all } 1 \leq i \leq n \end{array}$$

We write $\mathsf{Ins}(\pi)$ for the set of the instances of pattern π. The instance relation induces the following relations among patterns. These relations apply to patterns π_1 and π_2 that have the same type.

Definition 2.2 (Pattern relations)**.**

- Patterns π_1 and π_2 are **compatible** when they share at least one instance. Otherwise π_1 and π_2 are **incompatible** written $\pi_1 \# \pi_2$. Two compatible patterns admit a least upper bound written $\pi_1 \uparrow \pi_2$, whose instance set is $\mathsf{Ins}(\pi_1) \cap \mathsf{Ins}(\pi_2)$.
- Pattern π_1 is **less precise** than pattern π_2 written $\pi_1 \preceq \pi_2$ when $\mathsf{Ins}(\pi_2) \subseteq \mathsf{Ins}(\pi_1)$.
- Patterns π_1 and π_2 are **equivalent** written $\pi_1 \equiv \pi_2$ when $\mathsf{Ins}(\pi_1) = \mathsf{Ins}(\pi_2)$. If so, their least upper bound is their representative, written $\pi_i \updownarrow \pi_2$.

Note that we use the same notation $\preceq$ for both relations: "being an instance of" (which is between a pattern and a value) and "being less precise" (which is between two patterns). Indeed, values are in fact a special case of patterns (with no variables), and in that case, both relations collapse.

The least upper bound of two patterns can be computed at the same time when compatibility is checked by the following rules:

$$\begin{array}{rcl} \_ \uparrow \pi & = & \pi \\ \pi \uparrow \_ & = & \pi \\ \kappa(\pi_1,\ldots,\pi_n) \uparrow \kappa(\omega_1,\ldots,\omega_n) & = & \kappa(\pi_1 \uparrow \omega_1,\ldots,\pi_n \uparrow \omega_n) \end{array}$$

Deciding the relation "being less precise" is more involved. Because of typing, there exists nontrivial such relations, for instance $(\_,\_) \preceq \_$. The JoCaml compiler relies on an efficient algorithm for this task, called the $\mathcal{U}$ algorithm, with $\mathcal{U}$ standing for "Usefulness" [30]. Algorithm $\mathcal{U}$ takes two parameters: a list of patterns Π and a pattern π, and returns a boolean. Roughly speaking, it checks the usefulness of π with respect to Π. More specifically, algorithm $\mathcal{U}$ tests the existence of at least one value v such that π admits v as an instance, and none of the patterns in Π does.

From the point of view of algorithm $\mathcal{U}$, deciding the relation $\pi_1 \preceq \pi_2$ amounts to compute the *negation* of $\mathcal{U}([\pi_1], \pi_2)$. Namely, π_1 is less precise then π_2, if and only if all the instances of π_2 are instances of π_1.

$$\pi_1 \preceq \pi_2 \iff \mathcal{U}([\pi_1], \pi_2) = \textbf{useless}$$

We now give a simplified definition of algorithm $\mathcal{U}$. The simplified definition suffices for our needs and also conveys the basic idea behind the algorithm.

Consider $\mathcal{U}([\pi_1], \pi_2)$, where π_1 and π_2 are patterns of a common type t. The following two cases are distinguished.

Case $\pi_2 = \kappa(\omega_1, \ldots, \omega_n)$
- If $\pi_1 = \kappa(\gamma_1, \ldots, \gamma_n)$, then check if $\exists i, 1 \le i \le n$, s.t. $\mathcal{U}([\gamma_i], \omega_i)$.
- If $\pi_1 = \kappa'(\gamma_1, \ldots, \gamma_n)$ and $\kappa \ne \kappa'$, then **useful** (*i.e.* **false** for $\pi_1 \preceq \pi_2$).
- If $\pi_1 = \_$, then **useless** (*i.e.* **true** for $\pi_1 \preceq \pi_2$).

Case $\pi_2 = \_$
- If $\pi_1 = \_$, then **useless** (*i.e.* **true** for $\pi_1 \preceq \pi_2$).
- If $\pi_1 = \kappa(\gamma_1, \ldots, \gamma_n)$,
  - if κ is the unique constructor of type t, then check if $\exists i, 1 \le i \le n$, s.t. $\mathcal{U}([\gamma_i], \_)$.
  - otherwise **useful** (*i.e.* **false** for $\pi_1 \preceq \pi_2$).

Once we can decide relation "$\preceq$", we can easily decide pattern equivalence, since, by definition, $\pi_1 \equiv \pi_2$ means $\pi_1 \preceq \pi_2$ and $\pi_2 \preceq \pi_1$.

2.2. **ML pattern matching.** In ML, operating on algebraic data types is performed by the use of the following **match** *construct* that we extend to processes (Q_1, Q_2 etc. below are processes of the join calculus).

$$\textbf{match } v \textbf{ with } \pi_1 \to Q_1 \mid \pi_2 \to Q_2 \mid \ldots \mid \pi_n \to Q_n$$

Above, we attempt a matching of value v against a sequence of patterns $\pi_1, \ldots, \pi_n$ of the same type.

ML pattern matching is deterministic. It follows the "first match policy". That is, when value v is an instance of more than one of the patterns π_i, the **match** construct chooses the one with the smallest index i. This can be seen as checking patterns π_1, π_2, $\ldots$, π_n for admitting v as an instance sequentially, stopping as soon as a match is found. As a consequence, pattern π_i is matched only by the values in set $\mathsf{Ins}(\pi_i) \setminus (\bigcup_{1 \le j < i} \mathsf{Ins}(\pi_j))$. Moreover, patterns in ML pattern matching also act as binding constructs. Once a successful match is found, say $\pi_k \preceq v$, the variables in π_k are all bound to the corresponding subterms of v in the guarded process Q_k.

Additionally, we say a **match** construct is *exhaustive* when $\cup_{1 \le i \le n} \mathsf{Ins}(\pi_j)$ is the whole set of values of the considered type. We accept non-exhaustive **match** constructs.

3. The applied join calculus

We define the applied join calculus by analogy with the applied π-calculus [1]. The applied join calculus inherits its capabilities of communication and concurrency from pure join. Moreover it supports algebraic value passing and algebraic pattern matching in both join patterns and processes.

3.1. **Syntax and scopes.** The syntax of the applied join calculus is given in Figure 1. As it is customary in process calculi definitions, we assume an infinite set of identifiers for variables, ranged over by x, y, z.

With respect to pure join calculus, two new syntactic categories are introduced: expressions and patterns. At first glance, both expressions e and patterns π are terms constructed from variables and constructors, where n stands for the arity of constructor κ. We make them different syntactic categories for clarity, and also because we require patterns to be linear. We also formalize the ML pattern matching in processes, as the new **match** construct. Moreover, in contrast to ordinary name passing join calculus, there are two other,

P	$::=$		**Processes**
		0	null process
		$x(e)$	message sending
		$P \mathbin{\&} P$	parallel
		$\mathbf{def}\ D\ \mathbf{in}\ P$	definition
		$\mathbf{match}\ e\ \mathbf{with}\ \pi_1 \to P_1 \mid \ldots \mid \pi_m \to P_m$	pattern matching
D	$::=$		**Join definitions**
		$\top$	empty definition
		$J \triangleright P$	reaction
		$D\ \mathbf{or}\ D$	disjunction
J	$::=$		**Join patterns**
		$x(\pi)$	message pattern
		$J \mathbin{\&} J$	synchronization
π	$::=$		**Algebraic patterns**
		x	variable
		$\kappa(\pi_1, \pi_2, \ldots, \pi_n)$	constructor pattern
e	$::=$		**Expressions**
		x	variable
		$\kappa(e_1, e_2, \ldots, e_n)$	constructor expression

Figure 1: Syntax of the applied join calculus

more radical, extensions: first, in message sending, message contents become expressions as $x(e)$, that is, we have value passing; second, when a channel name is defined in a join pattern, in addition to the synchronization requirement, we also specify what pattern the message content should satisfy by $x(\pi)$.

There are two kinds of bindings: the definition process **def** D **in** P binds all the channel names defined in D (written $\mathsf{dv}[D]$) with scope P; and the reaction rule $J \triangleright P$ or the ML pattern matching **match** e **with** $\pi_1 \to P_1 \mid \ldots \mid \pi_m \to P_m$ bind all the local variables (written $\mathsf{rv}[J]$ or $\mathsf{rv}[\pi_i]$) with scope P or P_i, $i \in \{1, \ldots, m\}$. The definition of the sets of defined channel names $\mathsf{dv}[\cdot]$ is the same as in pure join. By contrast, the definition of sets $\mathsf{rv}[\cdot]$ has to be extended, so as to take pattern arguments into account. Meanwhile, the definition of sets $\mathsf{fv}[\cdot]$ should also be extended, to cater for the new **match** process and expressions. We present the formal definitions of $\mathsf{dv}[\cdot]$, $\mathsf{rv}[\cdot]$, and $\mathsf{fv}[\cdot]$ in Figure 2. In these rules, $\uplus$ is the disjoint union, which expresses linearity constraints on both algebraic and join patterns.

In applied join, values become of two kinds: channel names or algebraic values. We assume a type discipline in the style of the type system of the join-calculus [18], extended with algebraic data types and the rule for ML pattern matching. Without making the type discipline more explicit, we consider only well-typed terms (whose type we know), and assume that substitutions preserve types. It should be observed that tuples are now represented as a kind of constructed expressions and the arity checking of polyadic join calculus is now replaced by a well-typing assumption in applied join, which is thus monadic. One important consequence of typing is that any (free) variable in a term possesses a type and that we know this type. Hence, we can discriminate between those variables that are of a type of constructed values and those that are of channel type. Following the semantics

For algebraic patterns :

$$\begin{aligned}
\mathsf{rv}[x] &\stackrel{\text{def}}{=} \{x\} \\
\mathsf{rv}[\kappa(\pi_1, \pi_2, \ldots, \pi_n)] &\stackrel{\text{def}}{=} \mathsf{rv}[\pi_1] \uplus \mathsf{rv}[\pi_2] \uplus \ldots \uplus \mathsf{rv}[\pi_n]
\end{aligned}$$

For expressions :

$$\begin{aligned}
\mathsf{fv}[x] &\stackrel{\text{def}}{=} \{x\} \\
\mathsf{fv}[\kappa(e_1, e_2, \ldots, e_n)] &\stackrel{\text{def}}{=} \mathsf{fv}[e_1] \cup \mathsf{fv}[e_2] \cup \ldots \cup \mathsf{fv}[e_n]
\end{aligned}$$

For join patterns :

$$\begin{aligned}
\mathsf{rv}[x(\pi)] &\stackrel{\text{def}}{=} \mathsf{rv}[\pi] \\
\mathsf{rv}[J_1 \,\&\, J_2] &\stackrel{\text{def}}{=} \mathsf{rv}[J_1] \uplus \mathsf{rv}[J_2] \\
\\
\mathsf{dv}[x(\pi)] &\stackrel{\text{def}}{=} \{x\} \\
\mathsf{dv}[J_1 \,\&\, J_2] &\stackrel{\text{def}}{=} \mathsf{dv}[J_1] \uplus \mathsf{dv}[J_2]
\end{aligned}$$

For join definitions :

$$\begin{aligned}
\mathsf{dv}[\top] &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{dv}[J \triangleright P] &\stackrel{\text{def}}{=} \mathsf{dv}[J] \\
\mathsf{dv}[D_1 \textbf{ or } D_2] &\stackrel{\text{def}}{=} \mathsf{dv}[D_1] \cup \mathsf{dv}[D_2] \\
\\
\mathsf{fv}[\top] &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{fv}[J \triangleright P] &\stackrel{\text{def}}{=} \mathsf{dv}[J] \cup (\mathsf{fv}[P] \setminus \mathsf{rv}[J]) \\
\mathsf{fv}[D_1 \textbf{ or } D_2] &\stackrel{\text{def}}{=} \mathsf{fv}[D_1] \cup \mathsf{fv}[D_2]
\end{aligned}$$

For processes :

$$\begin{aligned}
\mathsf{fv}[0] &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{fv}[x(e)] &\stackrel{\text{def}}{=} \{x\} \cup \mathsf{fv}[e] \\
\mathsf{fv}[P_1 \,\&\, P_2] &\stackrel{\text{def}}{=} \mathsf{fv}[P_1] \cup \mathsf{fv}[P_2] \\
\mathsf{fv}[\textbf{def } D \textbf{ in } P] &\stackrel{\text{def}}{=} (\mathsf{fv}[D] \cup \mathsf{fv}[P]) \setminus \mathsf{dv}[D] \\
\mathsf{fv}[\textbf{match } e \textbf{ with}|^{i \in I}\ \pi_i \to P_i] &\stackrel{\text{def}}{=} \mathsf{fv}[e] \cup (\textstyle\bigcup_{i \in I} \mathsf{fv}[P_i] \setminus \mathsf{rv}[\pi_i])
\end{aligned}$$

For solutions :

$$\begin{aligned}
\mathsf{dv}[\mathcal{D}] &\stackrel{\text{def}}{=} \textstyle\bigcup_{D \in \mathcal{D}} \mathsf{dv}[D] \\
\mathsf{fv}[\mathcal{D}] &\stackrel{\text{def}}{=} \textstyle\bigcup_{D \in \mathcal{D}} \mathsf{fv}[D] \\
\mathsf{fv}[\mathcal{P}] &\stackrel{\text{def}}{=} \textstyle\bigcup_{P \in \mathcal{P}} \mathsf{fv}[P]
\end{aligned}$$

Figure 2: Bindings and scopes in the applied join calculus

$$
\begin{array}{llcl}
\textsc{Str-Null} & \vdash 0 & \rightleftharpoons & \vdash \\
\textsc{Str-Par} & \vdash P_1 \,\&\, P_2 & \rightleftharpoons & \vdash P_1, P_2 \\
\textsc{Str-Top} & \top \vdash & \rightleftharpoons & \vdash \\
\textsc{Str-Def} & \vdash \mathbf{def}\ D\ \mathbf{in}\ P & \rightleftharpoons & D \vdash P \\
\textsc{React} & J \triangleright P\ \mathbf{or}\ D \vdash J\sigma & \longrightarrow & J \triangleright P\ \mathbf{or}\ D \vdash P\sigma \\
\textsc{Match} & \multicolumn{3}{l}{\vdash \mathbf{match}\ \pi_i\eta\ \mathbf{with}\ \pi_1 \to P_1 \mid \ldots \mid \pi_m \to P_m} \\
 & & \longrightarrow & \vdash P_i\eta
\end{array}
$$

Side conditions:

Str-Def	$\mathsf{dv}[D]$ is fresh
React	σ substitutes closed expressions for $\mathsf{rv}[J]$
Match	η substitutes closed expressions for $\mathsf{rv}[\pi_i]$ and $\forall j < i, \pi_j \not\preccurlyeq \pi_i\eta$

Figure 3: RCHAM of the applied join calculus

of name passing calculi such as join, we treat the latter kind of variables as channel names, that is, values. While, in any reasonable semantics, the former kind of variables cannot be treated so. We call a term *variable-closed* (*closed* for short) when its free variables are all of channel type, and otherwise *open*.

3.2. **Chemical semantics.** We establish the semantics following the reflexive chemical abstract machine (RCHAM) style — the reflexive variant of CHAM [8], whose states are chemical solutions. A *chemical solution* is a pair $\mathcal{D} \vdash \mathcal{P}$, where $\mathcal{D}$ is a multiset of (active) join definitions, and $\mathcal{P}$ is a multiset of (running) processes. Extending the notion of closeness to solutions in the member-wise manner, we say a solution is closed when all its active join definitions and running processes are closed, namely, free variables are all of channel type. We define semantics only on closed solutions. The chemical rewriting rules are given in Figure 3, consisting of two kinds as in join: structural rules $\rightharpoonup$ or $\rightharpoondown$ represent the syntactical rearrangement of the terms, and reduction rules $\longrightarrow$ represent the computation steps. We follow the convention to omit the part of the solution that remains unchanged during rewrite. This can also be expressed by the following context rule:

$$
\textsc{Context}\ \frac{\mathcal{D}_0 \vdash \mathcal{P}_0 \ \overset{\rightharpoonup}{\underset{\rightharpoondown}{\longrightarrow}}\ \mathcal{D}_1 \vdash \mathcal{P}_1}{\mathcal{D}, \mathcal{D}_0 \vdash \mathcal{P}_0, \mathcal{P} \ \overset{\rightharpoonup}{\underset{\rightharpoondown}{\longrightarrow}}\ \mathcal{D}, \mathcal{D}_1 \vdash \mathcal{P}_1, \mathcal{P}}
$$

where $\overset{\rightharpoonup}{\underset{\rightharpoondown}{\longrightarrow}}$ stands for either $\rightleftharpoons$ or $\longrightarrow$, and $\mathcal{D}$ and $\mathcal{P}$ are the independent context of the considered subsolution. Rule Str-Def is a bit of exception because its side condition actually requires the following relationship hold between the rewriting part and its context: $\mathsf{dv}[D] \cap (\mathsf{fv}[\mathcal{D}] \cup \mathsf{fv}[\mathcal{P}]) = \emptyset$.

Finally, it is perhaps to be noticed that, amongst the various, slightly different, semantics of join-machines, we extend the one of [18], which is adapted to static typing. This means that we need to state explicitly that **or** is an associative-commutative operator. As a consequence, the notation $J \triangleright P$ **or** D in rule React stands for a definition that possesses a reaction rule whose pattern is J.

Matching of message contents against formal pattern arguments is integrated in the substitution σ in rule React. As a consequence this rule does not formally change with respect to ordinary join calculus. However its semantical power has much increased. The Match rule is new and expresses ML pattern matching. Its side condition enforces the first match policy.

According to the convention of processes as solutions, namely P as $\vdash P$, the semantics is also defined on closed processes in the following sense.

Definition 3.1. Let $\rightleftharpoons^*$ denote the transitive closure of $\rightharpoonup \cup \rightharpoondown$,
(1) $P \equiv Q$ iff $\vdash P \rightleftharpoons^* \vdash Q$
(2) $P \longrightarrow Q$ iff $\vdash P \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \vdash Q$

Subsequently, we have the following structural rule:

Lemma 3.2. *If $P \longrightarrow Q$, $P \equiv P'$, and $Q \equiv Q'$, then $P' \longrightarrow Q'$.*

Proof. Trivially follow the definitions of $\equiv$ and $\longrightarrow$, and the transitivity of $\rightleftharpoons^*$. □

3.3. **Equivalence relation.** In this section, we equip the applied join calculus with equivalence relations to allow reasoning over processes. The classical notion of *barbed congruence* is a sensible behavioral equivalence based on a reduction semantics and barb predicates. It was initially proposed by Milner and Sangiorgi for CCS [34], and adapted to many other process calculi [22, 3], including the join calculus. We take *weak barbed congruence* [34] as our basic notion of behavioral equivalence for closed processes.

3.3.1. *Observational equivalence for closed processes.*

Definition 3.3 (Barb predicates)**.** Let P be a closed process, and x be a free channel name in P,
(1) P has a strong barb on channel x: $P{\downarrow_x}$, iff $P \equiv \textbf{def } D \textbf{ in } Q \,\&\, x(e)$, for some D, Q and e, where $x \notin \mathsf{dv}[D]$.
(2) P has a weak barb on channel x: $P{\Downarrow_x}$, iff $P \longrightarrow^* P'$, such that $P'{\downarrow_x}$.
where $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

Following the definition, it is easy to check that two structurally congruent processes maintain the same barbs, *i.e.* the lemma below.

Lemma 3.4. *For two closed processes P and Q, whenever $P \equiv Q$, we have $P{\downarrow_x}$ iff $Q{\downarrow_x}$, and $P{\Downarrow_x}$ iff $Q{\Downarrow_x}$.*

Proof. The part for strong barb holds following the transitivity of $\equiv$, and the part for weak barb holds following Lemma 3.2. □

Definition 3.5 (Weak barbed bisimulation)**.** A binary relation $\mathcal{R}$ on closed processes is a weak barbed bisimulation, iff whenever $P \mathrel{\mathcal{R}} Q$, we have:
(1) If $P \longrightarrow^* P'$, then $\exists Q'$, such that $Q \longrightarrow^* Q'$ and $P' \mathrel{\mathcal{R}} Q'$, and vice versa. ($\mathcal{R}$ is a reduction bisimulation.)
(2) $P \Downarrow_x$ implies $Q \Downarrow_x$ for any channel x, and vice versa. ($\mathcal{R}$ preserves barbs.)

To make the definition easier to work with, we prove the following lemma where $P \longrightarrow^* P'$ is replaced by $P \longrightarrow P'$ in the first clause, and $P \Downarrow_x$ is replaced by $P \downarrow_x$ in the second clause.

Lemma 3.6. *Let $\mathcal{R}$ be a binary relation on closed processes that satisfies the following two conditions for any processes P and Q such that $P \mathrel{\mathcal{R}} Q$:*
(1) If $P \longrightarrow P'$, then $\exists Q'$, such that $Q \longrightarrow^* Q'$ and $P' \mathrel{\mathcal{R}} Q'$, and vice versa.
(2) $P \downarrow_x$ implies $Q \Downarrow_x$ for any channel x, and vice versa.
Then $\mathcal{R}$ is a weak barbed bisimulation.

Proof. We check against the two clauses of Definition 3.5 for one direction. The proof of the other direction is symmetric.
(1) $\mathcal{R}$ is a reduction bisimulation, that is
$$P \longrightarrow^* P' \implies \exists Q',\ \text{s.t. } Q \longrightarrow^* Q' \text{ and } P' \mathrel{\mathcal{R}} Q'$$
We reason on the length of the derivation $P \longrightarrow^* P'$, written n.

Base case. $n = 0, 1$, trivial.

Induction case. As illustrated in the following diagram chase,

$$\begin{tikzcd}
P \arrow[d, "\text{n-1}"] & \mathcal{R} & Q \arrow[d, "*"] \\
P_1 \arrow[d] & \mathcal{R} & Q_1 \arrow[d, "*"] \\
P' & \mathcal{R} & Q'
\end{tikzcd}$$

we have $P \longrightarrow^{n-1} P_1 \longrightarrow P'$. By induction hypothesis, we have $\exists Q_1$, s.t. $Q \longrightarrow^* Q_1$ and $P_1 \mathrel{\mathcal{R}} Q_1$. By applying hypothesis (1) to to P_1 and Q_1, we also have $\exists Q'$, s.t. $Q_1 \longrightarrow^* Q'$ and $P' \mathrel{\mathcal{R}} Q'$. And we conclude.
(2) $\mathcal{R}$ preserves barbs, that is $P \Downarrow_x \implies Q \Downarrow_x$. We thus assume $P \Downarrow_x$. That is,
$$\exists P', P \longrightarrow^* P' \text{ and } P' \downarrow_x$$
By (1) above,
$$\exists Q', Q \longrightarrow^* Q' \text{ and } P' \mathrel{\mathcal{R}} Q'$$
Then by applying hypothesis (2) to P' and Q', we get $Q' \Downarrow_x$. Hence we have $Q \Downarrow_x$.

□

In later discussion, we sometimes directly check against the two conditions of Lemma 3.6 instead of the ones of Definition 3.5 for weak barbed bisimulation.

We define *a context* as a term built by the grammar of process with a single process placeholder $[\cdot]$. *An evaluation context* $E[\cdot]$ is a context in which the placeholder is not guarded. Namely:

$$E[\cdot] \stackrel{\text{def}}{=} [\cdot] \mid E[\cdot] \,\&\, P \mid P \,\&\, E[\cdot] \mid \textbf{def } D \textbf{ in } E[\cdot]$$

In addition to evaluation contexts, there are two kinds of *guarded contexts*, referred to as *definition contexts* (*i.e.* **def** $J \triangleright [\cdot]$ **or** D **in** P) and *pattern matching contexts* (*i.e.* **match** e **with** $\ldots \mid \pi_k \to [\cdot] \mid \ldots$). We say that a context is closed if all the free variables in it are of channel types.

Definition 3.7 (Weak barbed congruence)**.** A binary relation on closed processes is a weak barbed congruence, iff it is a weak barbed bisimulation and closed by application of any closed evaluation context. We denote the largest weak barbed congruence as $\approx$.

The weak barbed congruence $\approx$ is defined on the closed subset of the applied join calculus. Although the definition itself only requires the closure of evaluation contexts, it can be proved that the full congruence does not provide more discriminative power. Similarly to what Fournet has established for the pure join calculus in his thesis [15], we first have the property that $\approx$ is closed by substitution because, roughly, name substitutions may be mimicked by evaluation contexts with "forwarders".

Lemma 3.8. *Given two closed processes P and Q, if $P \approx Q$, then for any substitution σ, $P\sigma \approx Q\sigma$. (Note that "closed" stands for "variable-closed".)*

Proof. The main idea is to build an evaluation context $E[\cdot]$ whose task is to forward messages from names to names according to the substitution σ, and to prove the equivalences $P\sigma \approx E[P]$ and $Q\sigma \approx E[Q]$. Because $\approx$ is closed by evaluation contexts, we also have $P \approx Q \implies E[P] \approx E[Q]$. Then we conclude by the transitivity of $\approx$. Refer to the proof of Fournet in [15, Lemma 4.14 of Chapter 4] for details. □

Then based on this property, the full congruence is also guaranteed considering the fact that the essence of a guarded context is substitution.

Theorem 3.9. *Weak barbed congruence $\approx$ is closed by application of any closed context.*

Proof. Corollary of Theorem 3.19 that we prove later on. □

Up to now, we have defined the weak barbed congruence to express the equivalence of two closed processes. However, our purpose is to study the correctness of a static transformation. Since static transformations apply perfectly well to processes with free variables of non-channel type, restricting ourselves to the world of closed processes is not an option. In the next section, we will derive an equivalence relation for open processes. But before getting into the definition, let us first establish some up-to techniques on the closed sub-set of the calculus. Such up-to techniques will be used during the courses of proving upcoming lemmas and theorems.

Definition 3.10 (Weak barbed congruence up to $\equiv$)**.** A binary relation $\mathcal{R}$ on closed processes is a weak barbed congruence up to $\equiv$, iff $P \mathrel{\mathcal{R}} Q$ implies:

(1) for any closed evaluation context $E[\cdot]$, $E[P] \equiv\mathcal{R}\equiv E[Q]$ ($\mathcal{R}$ is closed under evaluation contexts up to $\equiv$);

(2) whenever $P \longrightarrow^* P'$, $\exists Q'$, such that $Q \longrightarrow^* Q'$ and $P' \equiv\mathcal{R}\equiv Q'$, and vice versa ($\mathcal{R}$ is a reduction bisimulation up to $\equiv$);
(3) $P\Downarrow_x$ implies $Q\Downarrow_x$ for any channel x, and vice versa. ($\mathcal{R}$ preserves barbs.)

As we did for plain weak barbed bisimulation (Definition 3.5) in Lemma 3.6, we introduce the following weakened conditions for checking weak barbed congruence up to $\equiv$.

Lemma 3.11. *Let $\mathcal{R}$ be a binary relation on closed processes and $\mathcal{R}$ that satisfies the following three conditions for any processes P and Q such that $P \mathrel{\mathcal{R}} Q$:*
(1) *for any closed evaluation context $E[\cdot]$, $E[P] \equiv\mathcal{R}\equiv E[Q]$;*
(2) *If $P \longrightarrow P'$, then $\exists Q'$, such that $Q \longrightarrow^* Q'$ and $P' \equiv\mathcal{R}\equiv Q'$, and vice versa.*
(3) *$P\downarrow_x$ implies $Q\Downarrow_x$ for any channel x, and vice versa.*
Then $\mathcal{R}$ is a weak barbed congruence up to $\equiv$.

Proof. We check against the three clauses of Definition 3.10.
(1) The first clause is the same as clause (1) of Definition 3.10.
(2) We show:
$$P \longrightarrow^* P' \implies \exists Q',\ \text{s.t. } Q \longrightarrow^* Q' \text{ and } P' \equiv\mathcal{R}\equiv Q'$$
We reason on the length of the derivation $P \longrightarrow^* P'$, written n.

Base case. $n = 0, 1$, trivial.

Induction case. As illustrated in the following diagram chase,

$$\begin{tikzcd}
P \arrow[d, "n-1"] & & & \mathcal{R} & & & Q \arrow[d, "*"] \\
P_1 \arrow[d] & \equiv & P_2 \arrow[dll] & \mathcal{R} & Q_2 \arrow[d, "*"] & \equiv & Q_1 \arrow[dll, "*"'] \\
P' & \equiv & \mathcal{R} & \equiv & Q' & &
\end{tikzcd}$$

we have $P \longrightarrow^{n-1} P_1 \longrightarrow P'$. By induction hypothesis, we have $\exists Q_1, Q_2, P_2$, s.t. $Q \longrightarrow^* Q_1$ and $P_1 \equiv P_2 \mathrel{\mathcal{R}} Q_2 \equiv Q_1$. Following Lemma 3.2, we have $P_2 \longrightarrow P'$, too. By applying hypothesis (2) to P_2 and Q_2, we also have $\exists Q'$, s.t. $Q_2 \longrightarrow^* Q'$ and $P' \equiv\mathcal{R}\equiv Q'$. Then by Lemma 3.2 again, we have $Q_1 \longrightarrow^* Q'$, too. To conclude, we have $\exists Q'$, s.t. $Q \longrightarrow^* Q'$ and $P' \equiv\mathcal{R}\equiv Q'$.

The proof of the other direction is symmetric.
(3) We show:
$$P\Downarrow_x \implies Q\Downarrow_x$$
We thus assume $P\Downarrow_x$:
$$\exists P_1,\ \text{s.t. } P \longrightarrow^* P_1 \text{ and } P_1\downarrow_x$$
By (2) above, we get:
$$\exists Q_1, Q_2, P_2,\ \text{s.t. } Q \longrightarrow^* Q_1 \text{ and } P_1 \equiv P_2 \mathrel{\mathcal{R}} Q_2 \equiv Q_1$$
By Lemma 3.4, we have $P_2\downarrow_x$. Applying hypothesis (3) to P_2 and Q_2, we get $Q_2\Downarrow_x$. Then by Lemma 3.4 again, we have $Q_1\Downarrow_x$. To conclude, we have $Q \longrightarrow^* Q_1$ and $Q_1\Downarrow_x$, *i.e.* $Q\Downarrow_x$. The proof of the other direction is symmetric.

□

Lemma 3.12. *If $\mathcal{R}$ is a weak barbed congruence up to $\equiv$, then $\mathcal{R} \subseteq \approx$.*

Proof. We first show $\equiv\mathcal{R}\equiv\ \subseteq\ \approx$, *i.e.* $\equiv\mathcal{R}\equiv$ is a weak barbed congruence.

(1) $\equiv\mathcal{R}\equiv$ is closed under evaluation contexts. Given $P \equiv\mathcal{R}\equiv Q$, there exist P_1 and Q_1 such that $P \equiv P_1 \mathrel{\mathcal{R}} Q_1 \equiv Q$. Let us name two properties:
 (a) $\equiv$ is closed under evaluation contexts;
 (b) clause (1) of Definition 3.10.

 Then, for any closed evaluation context $E[\cdot]$, we have:

$$E[P] \overset{(a)}{\equiv} E[P_1] \overbrace{\equiv\mathcal{R}\equiv}^{(b)} E[Q_1] \overset{(a)}{\equiv} E[Q]$$

 By transitivity of $\equiv$, we conclude:

$$E[P] \equiv\mathcal{R}\equiv E[Q]$$

(2) $\equiv\mathcal{R}\equiv$ is a reduction bisimulation. We use clause (2) of Definition 3.10 and then Lemma 3.2 to reason by diagram chase as follows:

$$\begin{tikzcd}[column sep=small]
P \arrow[d] & \equiv & P_1 \arrow[dll] & \mathcal{R} & Q_1 \arrow[d, "*"] & \equiv & Q \arrow[dll, "*"] \\
P' & \equiv & \mathcal{R} & \equiv & Q' & &
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
P \arrow[drr, "*"'] & \equiv & P_1 \arrow[d, "*"] & \mathcal{R} & Q_1 \arrow[drr] & \equiv & Q \arrow[d] \\
& & P' & \equiv & \mathcal{R} & \equiv & Q'
\end{tikzcd}$$

(3) $\equiv\mathcal{R}\equiv$ preserves barbs. Given $P \equiv\mathcal{R}\equiv Q$, we have $P \equiv P_1 \mathrel{\mathcal{R}} Q_1 \equiv Q$, and the following statement,

$$P{\downarrow_x} \overset{\underline{Lemma}\ 3.4}{\Longrightarrow} P_1{\downarrow_x} \overset{Def\ \underline{3.10}.(3)}{\Longrightarrow} Q_1{\Downarrow_x} \overset{\underline{Lemma}\ 3.4}{\Longrightarrow} Q{\Downarrow_x}\text{ , and vice versa.}$$

Then because $\mathcal{R} \subseteq\ \equiv\mathcal{R}\equiv\ \subseteq\ \approx$, we conclude that $\mathcal{R} \subseteq \approx$. □

A standard proof technique is then to consider weak barbed congruence up to $\approx$. However, as demonstrated in [41], such a technique does not work in general in weak settings. Thus, we instead define another relation, where up to $\approx$ is performed on one side only. This new relation is sound, as shown by the forthcoming Lemma 3.15.

Definition 3.13 (Weak barbed congruence up to Id [3])**.** A binary relation $\mathcal{R}$ on closed processes is a weak barbed congruence up to Id, iff $P \mathrel{\mathcal{R}} Q$ implies:

(1) for any closed evaluation context $E[\cdot]$, $E[P] \equiv\mathcal{R}\equiv E[Q]$ ($\mathcal{R}$ is closed under evaluation contexts up to $\equiv$);
(2) whenever $P \longrightarrow^* P'$, $\exists Q'$, such that $Q \longrightarrow^* Q'$ and $P' \mathrel{\mathcal{R}\approx} Q'$;
(3) whenever $Q \longrightarrow^* Q'$, $\exists P'$, such that $P \longrightarrow^* P'$ and $P' \mathrel{\approx\mathcal{R}} Q'$;
 (The two clause above say that R is a reduction bisimulation up to Id.)
(4) $P{\Downarrow_x}$ implies $Q{\Downarrow_x}$ for any channel x, and vice versa. ($\mathcal{R}$ preserves barbs.)

Again, we first derive the following alternative conditions for checking weak barbed congruence up to Id.

[3] Id stands for the identity relation on closed processes. Note that this relation is derived from "bisimulation up to almost-weak bisimulation" in [41], because Id is included in almost-weak bisimulation, with some adjustments to the barbed setting.

Lemma 3.14. *$\mathcal{R}$ is a binary relation on closed processes and $\mathcal{R}$ satisfies the following conditions for any processes P and Q such that $P \mathrel{\mathcal{R}} Q$:*
(1) for any closed evaluation context $E[\cdot]$, $E[P] \equiv\mathcal{R}\equiv E[Q]$;
(2) whenever $P \longrightarrow P'$, $\exists Q'$, such that $Q \longrightarrow^* Q'$ and $P' \mathrel{\mathcal{R}\approx} Q'$;
(3) whenever $Q \longrightarrow Q'$, $\exists P'$, such that $P \longrightarrow^* P'$ and $P' \mathrel{\approx\mathcal{R}} Q'$;
(4) $P{\downarrow_x}$ implies $Q{\Downarrow_x}$ for any channel x, and vice versa.
Then $\mathcal{R}$ is a weak barbed congruence up to Id.

Proof. We check against the clauses of Definition 3.13.

(1) The first clause is the same.
(2) We show:

$$P \longrightarrow^* P' \implies \exists Q',\ \text{s.t. } Q \longrightarrow^* Q' \text{ and } P' \mathrel{\mathcal{R}\approx} Q'$$

We reason on the length of the derivation $P \longrightarrow^* P'$, written n.

Base case. $n = 0, 1$, trivial.

Induction case. As illustrated in the following diagram chase,

$$\begin{tikzcd}
P \arrow[d, "n-1"] & & \mathcal{R} & & Q \arrow[d, "*"] \\
P_1 \arrow[d] & \mathcal{R} & Q_2 \arrow[d, "*"] & \approx & Q_1 \arrow[d, "*"] \\
P' & \mathcal{R}\ \approx & Q_3 & \approx & Q'
\end{tikzcd}$$

we have $P \longrightarrow^{n-1} P_1 \longrightarrow P'$. By induction hypothesis, we get $\exists Q_1$, such that $Q \longrightarrow^* Q_1$ and $P_1 \mathrel{\mathcal{R}\approx} Q_1$. That is, $\exists Q_2$, such that $P_1 \mathrel{\mathcal{R}} Q_2 \approx Q_1$. By applying hypothesis (2) to P_1 and Q_2, we have $\exists Q_3$ such that $Q_2 \longrightarrow^* Q_3$ and $P' \mathrel{\mathcal{R}\approx} Q_3$. Because $Q_2 \approx Q_1$, we also have $\exists Q'$ such that $Q_1 \longrightarrow^* Q'$ and $Q_3 \approx Q'$ — remember that $\approx$ is the largest weak barbed congruence and thus a reduction bisimulation. We conclude by transitivity of $\approx$.
(3) Symmetric of (2) above.
(4) We show:

$$P{\Downarrow_x} \implies Q{\Downarrow_x}$$

We thus assume $P{\Downarrow_x}$. That is, we have:

$$\exists P_1 \text{ s.t } P \longrightarrow^* P_1 \text{ and } P_1{\downarrow_x}$$

By (2) above, we get:

$$\exists Q_1, Q_2,\ \text{s.t. } Q \longrightarrow^* Q_1 \text{ and } P_1 \mathrel{\mathcal{R}} Q_2 \approx Q_1$$

Applying hypothesis (4) to P_1 and Q_2, we get $Q_2{\Downarrow_x}$. Applying clause (2) of Definition 3.5 to Q_2 and Q_1, we then get $Q_1{\Downarrow_x}$. To conclude, we have $Q \longrightarrow^* Q_1$ and $Q_1{\Downarrow_x}$, *i.e.* $Q{\Downarrow_x}$. The proof of the other direction is symmetric.

∎

Lemma 3.15. *If* $\mathcal{R}$ *is a weak barbed congruence up to* Id*, then* $\mathcal{R} \subseteq \approx$*.*

Proof. We first show $\approx\mathcal{R}\approx \subseteq \approx$, *i.e.* $\approx\mathcal{R}\approx$ is a weak barbed congruence.

(1) $\approx\mathcal{R}\approx$ is closed under evaluation contexts. Given $P \approx\mathcal{R}\approx Q$, there exist P_1 and Q_1 such that $P \approx P_1 \mathrel{\mathcal{R}} Q_1 \approx Q$. Let us name two properties:
   (a) $\approx$ is closed under evaluation contexts;
   (b) clause (1) of Definition 3.13;
   Then, for any closed evaluation context $E[\cdot]$, we have:
$$E[P] \overset{(a)}{\approx} E[P_1] \overbrace{\equiv\mathcal{R}\equiv}^{(b)} E[Q_1] \overset{(a)}{\approx} E[Q]$$
   Because $\equiv \subseteq \approx$, we have $\equiv\mathcal{R}\equiv \subseteq \approx\mathcal{R}\approx$. Hence we have:
$$E[P] \approx E[P_1] \approx\mathcal{R}\approx E[Q_1] \approx E[Q]$$
   And we conclude, by transitivity of $\approx$.

(2) $\approx\mathcal{R}\approx$ is a reduction bisimulation. We use clause (1) of Definition 3.5, clause (2) of Definition 3.13, clause (1) of Definition 3.5, and the transitivity of $\approx$, in the proof sketched by the following diagram:
$$\begin{tikzcd}[column sep=small]
P \arrow[d] & \approx & P_1 \arrow[d, "*"] & \mathcal{R} & Q_1 \arrow[d, "*"] & \approx & Q \arrow[d, "*"] \\
P' & \approx & P'_1 & \mathcal{R}\approx & Q'_1 & \approx & Q' \\
P' & \approx & & \mathcal{R} & & \approx & Q'
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
P \arrow[d, "*"'] & \approx & P_1 \arrow[d, "*"'] & \mathcal{R} & Q_1 \arrow[d, "*"'] & \approx & Q \arrow[d] \\
P' & \approx & P'_1 & \approx\mathcal{R} & Q'_1 & \approx & Q' \\
P' & \approx & & \mathcal{R} & & \approx & Q'
\end{tikzcd}$$

(3) $\approx\mathcal{R}\approx$ preserves barbs. Given $P \approx\mathcal{R}\approx Q$, we have $P \approx P_1 \mathrel{\mathcal{R}} Q_1 \approx Q$, and the following statement,
$$P{\downarrow_x} \overset{\mathit{Def}\ 3.5.(2)}{\Longrightarrow} P_1{\Downarrow_x} \overset{\mathit{Def}\ 3.13.(3)}{\Longrightarrow} Q_1{\Downarrow_x} \overset{\mathit{Def}\ 3.5.(2)}{\Longrightarrow} Q{\Downarrow_x}\ \text{, and vice versa.}$$

Then because $\mathcal{R} \subseteq \approx\mathcal{R}\approx \subseteq \approx$, we conclude that $\mathcal{R} \subseteq \approx$. ☐

3.3.2. *Observational equivalence for open processes.* The approach we follow here is to lift the equivalence relation of closed processes to open processes by closing up by all substitutions, and we call the resulting relation *open equivalence.*

Although both are "open", our open equivalence is unrelated to the open bisimilarity of Sangiorgi in [40]. We use "open" to name our equivalence relation because it relates open terms. By contrast, "open" in open bisimilarity emphasizes a characteristic of the bisimulation definition, namely free names are open to equality throughout the bisimulation game. From the perspective of where and when to apply name substitutions, for open equivalence, we instantiate free names (and variables) only at the beginning before we test the resulting (closed) processes for weak barbed congruence. On the contrary, in the case of open bisimilarity, such instantiation happens at every co-inductive step.

Another way to define equivalence relations on open terms could be to adapt the semantics to symbolic transition system and to define a symbolic barbed congruence like in [5]. Although the symbolic method is claimed to be easier for analysis and verification, we found open equivalence to be lighter and more intuitive. As a matter of fact, it is not uncommon to define functions extensionally, *i.e.* by considering application to all possible arguments. Moreover, as can be seen in Section 7, our proofs remain tractable.

Definition 3.16 (Open equivalence $\Bumpeq$)**.** Two processes P and Q are open equivalent, written $P \Bumpeq Q$, iff for any substitution σ such that $P\sigma$ and $Q\sigma$ are closed, we have $P\sigma \approx Q\sigma$.

As a corollary, $\Bumpeq$ is closed by any substitution.

Lemma 3.17. $P \Bumpeq Q \implies \forall \sigma . P\sigma \Bumpeq Q\sigma$

Proof. We assume $P \Bumpeq Q$ and let σ be a substitution. We need to prove that $P\sigma \Bumpeq Q\sigma$. That is, we need to prove that, for all closing substitution ρ, we have:

$$(P\sigma)\rho \approx (Q\sigma)\rho$$

Thus, we need to prove that, for all closing substitution ρ, we have:

$$P(\rho \circ \sigma) \approx Q(\rho \circ \sigma) \tag{3.1}$$

where $\circ$ stands for substitution composition, *i.e.* $P(\rho \circ \sigma) \stackrel{\text{def}}{=} (P\sigma)\rho$. It remains to observe that $\rho \circ \sigma$ closes both processes P and Q, and to apply the definition of $P \Bumpeq Q$, before concluding that statement (3.1) above holds. □

We aim at proving that $\Bumpeq$ is closed by any contexts (Theorem 3.19 below). To prove the theorem, we need the following rather unusual lemma, to state the fact that although we have introduced "deterministic" reduction into the process calculus by extending it with the **match** construct, this kind of determinism does not impact process equivalence.

Lemma 3.18. *We say a closed process P* deterministically reduces *to P', iff for all P'' such that $P \longrightarrow P''$, we have $P' \equiv P''$. For any such pair of closed processes P and P', we have $P \approx P'$.*

Proof. Let $\mathcal{R}$ be the relation $\{(\textbf{def } D \textbf{ in } P \,\&\, Q,\ \textbf{def } D \textbf{ in } P' \,\&\, Q), (S, S)\}$ for all closed definitions D, closed processes Q and S, and all (P, P') pairs such that P deterministically reduces to P'. We prove that $\mathcal{R}$ is a weak barbed congruence up to $\equiv$.

- By definition, $\mathcal{R}$ is closed by evaluation contexts up to $\equiv$ (*i.e.* Lemma 3.11.(1)).
- We show that $\mathcal{R}$ preserves barbs (*i.e.* Lemma 3.11.(3)). We omit the (trivial) discussion of pairs of identical processes (S, S) in $\mathcal{R}$. We show that $(\textbf{def } D \textbf{ in } P \,\&\, Q){\downarrow_x} \implies (\textbf{def } D \textbf{ in } P' \,\&\, Q){\Downarrow_x}$. We distinguish the cases that make $(\textbf{def } D \textbf{ in } P \,\&\, Q){\downarrow_x}$ hold.
  - $P{\downarrow_x}$. Obviously reduction cannot erase a barb ($x \notin \mathsf{dv}[D]$), *i.e.* we have $P'{\downarrow_x}$. Hence, we have $(\textbf{def } D \textbf{ in } P' \,\&\, Q){\downarrow_x}$.
  - $Q{\downarrow_x}$. Trivial.

  As to the opposite direction *i.e.* $(\textbf{def } D \textbf{ in } P' \,\&\, Q){\downarrow_x} \implies (\textbf{def } D \textbf{ in } P \,\&\, Q){\Downarrow_x}$, it holds trivially because $\textbf{def } D \textbf{ in } P \,\&\, Q \longrightarrow \textbf{def } D \textbf{ in } P' \,\&\, Q$.
- We show $\mathcal{R}$ to be a reduction bisimulation up to $\equiv$ (*i.e.* Lemma 3.11.(2)). We omit the trivial case of pairs of identical processes in $\mathcal{R}$, that is, we only consider process pairs of form: $(\textbf{def } D \textbf{ in } P \,\&\, Q,\ \textbf{def } D \textbf{ in } P' \,\&\, Q)$.
  - If the reduction of the left part is caused by a reduction on Q alone or by the interaction between D and Q, yielding $\textbf{def } D \textbf{ in } P \,\&\, Q'$, then the right part can perform the same reduction step, yielding $\textbf{def } D \textbf{ in } P' \,\&\, Q'$. The resulting two processes are still in relation $\mathcal{R}$ with Q being Q'. Vice versa.
  - If the reduction of the left part is caused by a reduction on P alone, then, because P deterministically reduces to P', the resulting process is $\textbf{def } D \textbf{ in } P' \,\&\, Q$ (up to $\equiv$). Thus, the right part simulates with no reduction and $\textbf{def } D \textbf{ in } P' \,\&\, Q$ satisfies relation $\mathcal{R}$ with itself.

– If the reduction of the left part is caused by the interaction between D and P, then we must have $P \equiv P_0 \,\&\, J\sigma$ where $J \triangleright G$ is a reaction rule in D and the resulting process is **def** D **in** $P_0 \,\&\, Q \,\&\, G\sigma$. Because $J\sigma$ does not reduce by itself and P deterministically reduces to P', we have $P' \equiv P'_0 \,\&\, J\sigma$ and P_0 deterministically reduces to P'_0. Therefore, the right part simulates by an identical reduction and gives **def** D **in** $P'_0 \,\&\, Q \,\&\, G\sigma$. The resulting two processes are still in relation $\mathcal{R}$ with Q being $Q \,\&\, G\sigma$, P being P_0, and P' being P'_0.
– If the reduction of the right part is caused by a reduction on P' itself or by the interaction between D and P', then the left part can always simulate the reduction by first reducing **def** D **in** $P \,\&\, Q$ to **def** D **in** $P' \,\&\, Q$.

Following the analysis above, $\mathcal{R}$ is a weak barbed congruence up to $\equiv$. Besides we have

$$P \equiv (\mathbf{def}\ \top\ \mathbf{in}\ P \,\&\, 0)\ \mathcal{R}\ (\mathbf{def}\ \top\ \mathbf{in}\ P' \,\&\, 0) \equiv P'$$

Moreover, by the proof of Lemma 3.12, relation $\equiv\mathcal{R}\equiv$ is a weak barbed congruence. Hence we conclude $P \approx P'$. □

Theorem 3.19. *The open equivalence* $\diamondsuit$ *is a full congruence.*

Proof. We demonstrate $\diamondsuit$ is closed by 1. evaluation contexts, 2. definition contexts, and 3. pattern matching contexts. In the proof, we locally use A, B, R, S, T, V, W, X, Y, Z to denote various processes.

1. Closed by evaluation contexts: $E[\cdot]$**.** We show:

$$P \diamondsuit Q \implies E[P] \diamondsuit E[Q]$$

For any substitution σ such that $(E[P])\sigma$ and $(E[Q])\sigma$ are closed, we need to prove $(E[P])\sigma \approx (E[Q])\sigma$. We write $(E[P])\sigma$ as $E\sigma[P\sigma_1]$ and $(E[Q])\sigma$ as $E\sigma[Q\sigma_1]$, where $E\sigma[\cdot]$, $P\sigma_1$, $Q\sigma_1$ are closed and σ_1 is σ minus the (possible) bindings for the channel names bound by E in $[\cdot]$. By hypothesis $P \diamondsuit Q$, we have $P\sigma_1 \approx Q\sigma_1$. Then, $E\sigma[\cdot]$ being a closed evaluation context, we conclude, by definition of $\approx$.

2. Closed by definition contexts: def $J \triangleright [\cdot]$ **or** D **in** R**.** We show:

$$P \diamondsuit Q \implies (\mathbf{def}\ J \triangleright P\ \mathbf{or}\ D\ \mathbf{in}\ R) \diamondsuit (\mathbf{def}\ J \triangleright Q\ \mathbf{or}\ D\ \mathbf{in}\ R)$$

For any substitution σ such that $(\mathbf{def}\ J \triangleright P\ \mathbf{or}\ D\ \mathbf{in}\ R)\sigma$ and $(\mathbf{def}\ J \triangleright Q\ \mathbf{or}\ D\ \mathbf{in}\ R)\sigma$ are closed, we need to prove:

$$(\mathbf{def}\ J \triangleright P\ \mathbf{or}\ D\ \mathbf{in}\ R)\sigma \approx (\mathbf{def}\ J \triangleright Q\ \mathbf{or}\ D\ \mathbf{in}\ R)\sigma$$

namely,

$$\mathbf{def}\ J \triangleright P\sigma_1\ \mathbf{or}\ D\sigma_2\ \mathbf{in}\ R\sigma_2 \approx \mathbf{def}\ J \triangleright Q\sigma_1\ \mathbf{or}\ D\sigma_2\ \mathbf{in}\ R\sigma_2 \tag{3.1}$$

where σ_2 is σ minus the (possible) bindings for the channel names defined in $J \triangleright P$ **or** D (*i.e.* $\mathsf{dv}[J \triangleright P\ \mathbf{or}\ D]$), and σ_1 is σ_2 minus the (possible) bindings for the variables of $\mathsf{rv}[J]$. Notice that, by contrast with the subcomponents $D\sigma_2$ and $R\sigma_2$ that are closed, the processes $P\sigma_1$ and $Q\sigma_1$ may *not* be closed, since some of the variables in $\mathsf{rv}[J]$ may be of an algebraic type. Nevertheless, by hypothesis $P \diamondsuit Q$ and Lemma 3.17, we have $P\sigma_1 \diamondsuit Q\sigma_1$.

Then, we build the following relation $\mathcal{R}$ on closed processes:

$$\mathcal{R} = \{(\mathbf{def}\ J \triangleright S\ \mathbf{or}\ D\ \mathbf{in}\ A, \mathbf{def}\ J \triangleright T\ \mathbf{or}\ D\ \mathbf{in}\ B) \mid S \diamondsuit T \text{ and } A \approx B\}.$$

We analyze the following three aspects of $\mathcal{R}$: closure by closed evaluation contexts; preserving barbs; and reduction bisimulation.

- $\mathcal{R}$ is closed by closed evaluation contexts up to $\equiv$ (*i.e.* Lemma 3.14.(1)). For any closed $E[\cdot]$, with necessary α-conversions left implicit, we have:

$$E[\textbf{def } J \triangleright S \textbf{ or } D \textbf{ in } A] \equiv \textbf{def } J \triangleright S \textbf{ or } (D \textbf{ or } D') \textbf{ in } (A \,\&\, K)$$
$$E[\textbf{def } J \triangleright T \textbf{ or } D \textbf{ in } B] \equiv \textbf{def } J \triangleright T \textbf{ or } (D \textbf{ or } D') \textbf{ in } (B \,\&\, K)$$

  where $A \,\&\, K \approx B \,\&\, K$, because $\approx$ is preserved by the closed evaluation context $[\cdot] \,\&\, K$.
- $\mathcal{R}$ preserves barbs (*i.e.* Lemma 3.14.(4)). We write $\mathcal{D}[X, Y]$ for the closed process **def** $J \triangleright X$ **or** D **in** Y. Since $\mathcal{R}$ is a symmetric relation, we only need to prove:

$$\mathcal{D}[S, A]\downarrow_x \Longrightarrow \mathcal{D}[T, B]\Downarrow_x$$

  Because $\mathcal{D}[S, A]\downarrow_x$ implies $A\downarrow_x$ and $x \notin (\mathsf{dv}[J] \cup \mathsf{dv}[D])$, we also have $\mathcal{D}[T, A]\downarrow_x$. Moreover, because $\mathcal{D}[T, \cdot]$ is a closed evaluation context, and by hypothesis $A \approx B$, we have

$$\mathcal{D}[T, A] \approx \mathcal{D}[T, B] \tag{3.2}$$

  By clause (2) of Definition 3.5, we finally get $\mathcal{D}[T, B]\Downarrow_x$.
- $\mathcal{R}$ is a reduction bisimulation up to *Id* (*i.e.* Lemma 3.14.(2) and (3)). We first prove the following statement. For any two $\mathcal{D}[S, A]$ and $\mathcal{D}[T, A]$, we have:

$$\text{If } \mathcal{D}[S, A] \longrightarrow W, \text{ then } \mathcal{D}[T, A] \longrightarrow V, \text{ and } W \mathrel{\mathcal{R}} V. \tag{3.3}$$

  There are three subcases, depending on the nature of the reduction to W.
  (1) $A \longrightarrow A'$ and $W = \mathcal{D}[S, A']$. Then $\mathcal{D}[T, A] \longrightarrow \mathcal{D}[T, A']$, with obviously $\mathcal{D}[S, A'] \mathrel{\mathcal{R}} \mathcal{D}[T, A']$, since $A' \approx A'$.
  (2) $A \equiv A_0 \,\&\, J\eta$ and $W = \mathcal{D}[S, A_0 \,\&\, S\eta]$. Then $\mathcal{D}[T, A] \longrightarrow \mathcal{D}[T, A_0 \,\&\, T\eta]$. Notice that $S\eta$ and $T\eta$ are closed. Then, from $S \mathrel{\dot{\approx}} T$, we get $S\eta \approx T\eta$, and thus $A_0 \,\&\, S\eta \approx A_0 \,\&\, T\eta$. That is, we get $\mathcal{D}[S, A_0 \,\&\, S\eta] \mathrel{\mathcal{R}} \mathcal{D}[T, A_0 \,\&\, T\eta]$.
  (3) $A \equiv A_0 \,\&\, J_i\eta_i$, D has form $\ldots$ **or** $J_i \triangleright P_i$ **or** $\ldots$, and $W = \mathcal{D}[S, A_0 \,\&\, P_i\eta_i]$. Then $\mathcal{D}[T, A] \longrightarrow \mathcal{D}[T, A_0 \,\&\, P_i\eta_i]$. And we conclude, as we did in case 1 above.

  Moreover, from equivalence (3.2) and since $\approx$ is a bisimulation, we have:

$$\text{If } \mathcal{D}[T, A] \longrightarrow V, \text{ then } \exists V' \text{ s.t. } \mathcal{D}[T, B] \longrightarrow^* V', \text{ and } V \approx V'. \tag{3.4}$$

  Combining both statements (3.3) and (3.4), we get:

$$\text{If } \mathcal{D}[S, A] \longrightarrow W, \text{ then } \exists V' \text{ s.t. } \mathcal{D}[T, B] \longrightarrow^* V', \text{ and } W \mathrel{\mathcal{R}\approx} V'. \tag{3.5}$$

  The proof of the other direction is by symmetry.

Following the analysis above, $\mathcal{R}$ is a weak barbed congruence up to *Id*, hence by Lemma 3.15, $\mathcal{R} \subseteq \approx$. Obviously, the two processes of statement (3.1) are related by $\mathcal{R}$. Therefore, (3.1) holds. In other words, $\dot{\approx}$ is closed by any definition context.

3. Closed by pattern matching contexts: match e **with** $\ldots \mid \pi_k \to [\cdot] \mid \ldots$**.** We show:

$$P \mathrel{\dot{\approx}} Q \implies$$
$$(\textbf{match } e \textbf{ with} \ldots \mid \pi_k \to P \mid \ldots) \mathrel{\dot{\approx}} (\textbf{match } e \textbf{ with} \ldots \mid \pi_k \to Q \mid \ldots) \quad (3.6)$$

To establish the right part, we need to show:

$$(\textbf{match } e \textbf{ with} \ldots \mid \pi_k \to P \mid \ldots)\sigma \approx (\textbf{match } e \textbf{ with} \ldots \mid \pi_k \to Q \mid \ldots)\sigma$$

for all σ, s.t. $(\textbf{match } e \textbf{ with} \ldots \mid \pi_k \to P \mid \ldots)\sigma$ and $(\textbf{match } e \textbf{ with} \ldots \mid \pi_k \to Q \mid \ldots)\sigma$ are closed. Namely,

$$\textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to P\sigma_k \mid \ldots \approx \textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to Q\sigma_k \mid \ldots \quad (3.7)$$

where σ_k is σ minus the (possible) bindings for the variables of $\mathsf{rv}[\pi_k]$. Notice that $e\sigma$ is closed, while $P\sigma_k$ and $Q\sigma_k$ may not be.

By the semantics of ML pattern matching, $\textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to P\sigma_k \mid \ldots$ deterministically reduces to either $P(\eta_k \circ \sigma_k)$ or $R_i\eta_i$, depending on the value of $e\sigma$. Process R_i is the ith guarded process ($i \neq k$) in this pattern matching, η_k and η_i stand for the substitutions that originate from algebraic matching. Notice that $P(\eta_k \circ \sigma_k)$ and $R_i\eta_i$ now are closed processes. We have the similar statement for $\textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to Q\sigma_k \mid \ldots$. Therefore, by Lemma 3.18, we have either:

$$\textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to P\sigma_k \mid \ldots \approx P(\eta_k \circ \sigma_k) \quad (3.8)$$
$$\textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to Q\sigma_k \mid \ldots \approx Q(\eta_k \circ \sigma_k) \quad (3.9)$$

or we have:

$$\textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to P\sigma_k \mid \ldots \approx R_i\eta_i \quad (3.10)$$
$$\textbf{match } e\sigma \textbf{ with} \ldots \mid \pi_k \to Q\sigma_k \mid \ldots \approx R_i\eta_i \quad (3.11)$$

Obviously we have $R_i\eta_i \approx R_i\eta_i$. Moreover, since $P \mathrel{\dot{\approx}} Q$, we get $P(\eta_k \circ \sigma_k) \approx Q(\eta_k \circ \sigma_k)$. Then, by the transitivity of $\approx$ and, either by (3.8)–(3.9), or by (3.10)–(3.11), we conclude that the statement (3.7) holds.

Additionally, in the case where $e\sigma$ matches none of the patterns in (3.7), both processes are blocked and are $\approx$ to the null process 0. □

There is still a good property worth noticing: for the closed subset of the applied join-calculus, the equivalences $\dot{\approx}$ and $\approx$ coincide. This is straightforward by the definition of $\dot{\approx}$ and by Lemma 3.8. Then, Theorem 3.9 follows as a corollary.

4. Transforming pattern arguments into ML pattern matching

The extension of the join calculus that we have presented up to now remains quite simple, in particular as regards chemical semantics. However, an efficient implementation is more involved. Our approach is to first transform the extended join definitions into ordinary ones plus ML pattern matching, then reuse the existing implementation of join. In this section, we explain informally the key ideas of the transformation.

The extended join-pattern matching in applied join requires to test message contents against pattern arguments, while the ordinary join-pattern matching in join is only capable of testing message presence. Our idea is to separate algebraic pattern testing from join-pattern synchronization, and to perform the former operation by using ML pattern matching. To avoid inappropriate message consumption, message contents are tested first. Let us consider the following join definition where channel x has two pattern arguments:

$$\begin{array}{l} \textbf{def } x(\pi_1) \ \&\ y_1(\ldots) \triangleright P_1 \\ \ \ \textbf{or } x(\pi_2) \ \&\ y_2(\ldots) \triangleright P_2 \end{array}$$

We refine channel x into more precise ones, each of which carries the instances of patterns π_1 or π_2:

$$\begin{array}{l} \textbf{def } x_{\pi_1}(\ldots) \ \&\ y_1(\ldots) \triangleright P_1 \\ \ \ \textbf{or } x_{\pi_2}(\ldots) \ \&\ y_2(\ldots) \triangleright P_2 \end{array}$$

Then, we add a new reaction rule to dispatch the messages on channel x to either x_{π_1} or x_{π_2}:

$$\begin{array}{l} \textbf{or } x(z) \triangleright \textbf{match } z \textbf{ with} \\ \qquad\qquad \mid \pi_1 \rightarrow x_{\pi_1}(...) \\ \qquad\qquad \mid \pi_2 \rightarrow x_{\pi_2}(...) \\ \qquad\qquad \mid \_ \rightarrow 0 \end{array}$$

Note that the null process is used in the last matching rule to discard messages that match neither π_1 nor π_2.

The simple compilation above works perfectly, as long as π_1 and π_2 are incompatible. Unfortunately, it falls short when π_1 and π_2 have common instances. Consider the situation where there is a message pending on channel y_2, none on y_1, and also a message v on x where v is a common instance of patterns π_1 and π_2. Then, following the first match policy, the deterministic ML pattern matching can only dispatch $x(v)$ to the refined channel x_{π_1}. As a result, the guarded process P_2 is not triggered, whereas it could have been.[4] To tackle this problem, further refinements are called for according to the following cases.

- If $\pi_1 \preceq \pi_2$, (but $\pi_2 \not\preceq \pi_1$), that is if all instances of π_2 are instances of π_1, then, to get a chance of meeting its instances, pattern π_2 must come first:

  $$\begin{array}{l} \textbf{or } x(z) \triangleright \textbf{match } z \textbf{ with} \\ \qquad\qquad \mid \pi_2 \rightarrow x_{\pi_2}(...) \\ \qquad\qquad \mid \pi_1 \rightarrow x_{\pi_1}(...) \\ \qquad\qquad \mid \_ \rightarrow 0 \end{array}$$

  But now, channel x_{π_1} does not carry all the possible instances of pattern π_1 any more, instances shared by pattern π_2 are dispatched to x_{π_2}. As a consequence, the actual transformation of the initial reaction rules is as follows:

  $$\begin{array}{l} \textbf{def } x_{\pi_1}(\ldots) \ \&\ y_1(\ldots) \triangleright P_1 \\ \ \ \textbf{or } x_{\pi_2}(\ldots) \ \&\ y_1(\ldots) \triangleright P_1 \\ \ \ \textbf{or } x_{\pi_2}(\ldots) \ \&\ y_2(\ldots) \triangleright P_2 \end{array}$$

  Observe that nondeterminism is now more explicit: an instance of π_2 sent on channel x can be consumed by either the second or the third reaction rule to trigger either P_1 or P_2. We can shorten the new definition a little by using **or** in join patterns:

  $$\begin{array}{l} \textbf{def } (x_{\pi_1}(\ldots) \textbf{ or } x_{\pi_2}(\ldots)) \ \&\ y_1(\ldots) \triangleright P_1 \\ \ \ \textbf{or } x_{\pi_2}(\ldots) \ \&\ y_2(\ldots) \triangleright P_2 \end{array}$$

  Here the disjunctive composition (J_1 **or** J_2) in join patterns works as syntactic sugar, in the following sense:

  $$J \,\&(J_1 \textbf{ or } J_2) \triangleright P \ \stackrel{\text{def}}{=}\ (J \,\&\, J_1 \triangleright P) \textbf{ or } (J \,\&\, J_2 \triangleright P)$$

- If $\pi_1 \equiv \pi_2$, then matching by their representative is enough:

[4]Given our implementation "limited fairness guarantee", it can be argued that P_2 should be triggered.

$$
\begin{array}{l}
\textbf{def } x_{\pi_1 \updownarrow \pi_2}(\ldots) \ \&\ y_1(\ldots) \triangleright P_1 \\
\ \ \textbf{or } x_{\pi_1 \updownarrow \pi_2}(\ldots) \ \&\ y_2(\ldots) \triangleright P_2 \\
\ \ \textbf{or } x(z) \triangleright \textbf{match } z \textbf{ with} \\
\qquad\qquad\quad \mid \pi_1 \updownarrow \pi_2 \rightarrow \ x_{\pi_1 \updownarrow \pi_2}(\ldots) \\
\qquad\qquad\quad \mid \_ \rightarrow 0
\end{array}
$$

- Finally, if neither $\pi_1 \preceq \pi_2$ nor $\pi_2 \preceq \pi_1$ holds, with π_1 and π_2 being nevertheless compatible, then an extra matching by pattern $\pi_1 \uparrow \pi_2$ is needed:

$$
\begin{array}{l}
\textbf{def } (x_{\pi_1}(\ldots) \textbf{ or } x_{\pi_1 \uparrow \pi_2}(\ldots)) \ \&\ y_1(\ldots) \triangleright P_1 \\
\ \ \textbf{or } (x_{\pi_2}(\ldots) \textbf{ or } x_{\pi_1 \uparrow \pi_2}(\ldots)) \ \&\ y_2(\ldots) \triangleright P_2 \\
\ \ \textbf{or } x(z) \triangleright \textbf{match } z \textbf{ with} \\
\qquad\qquad\quad \mid \pi_1 \uparrow \pi_2 \rightarrow x_{\pi_1 \uparrow \pi_2}(\ldots) \\
\qquad\qquad\quad \mid \pi_1 \rightarrow x_{\pi_1}(\ldots) \mid \pi_2 \rightarrow x_{\pi_2}(\ldots) \\
\qquad\qquad\quad \mid \_ \rightarrow 0
\end{array}
$$

Note that the relative order of π_1 and π_2 is irrelevant here.

In the transformation rules above, we paid little attention to variables in patterns, by writing $x_\pi(\ldots)$. We now show variable management by means of the concurrent stack example. Here, the relevant patterns are $\pi_1 = \mathit{ls}$ and $\pi_2 = x::\mathit{xs}$ and we are in the case where $\pi_1 \preceq \pi_2$ (and $\pi_2 \not\preceq \pi_1$ because of instance empty list `[]`). Our idea is to let dispatching focus on instance checking, and to perform variable binding after synchronization:

$$
\begin{array}{l}
\textbf{def } \mathit{pop}(r) \ \&\ \mathit{State}_{x::xs}(z) \triangleright \textbf{match } z \textbf{ with } x::\mathit{xs} \rightarrow r(x) \ \&\ \mathit{State}(\mathit{xs}) \\
\ \ \textbf{or } \mathit{push}(v) \ \&\ (\mathit{State}_{x::xs}(z) \textbf{ or } \mathit{State}_{ls}(z)) \triangleright \textbf{match } z \textbf{ with } \mathit{ls} \rightarrow \mathit{State}(v::\mathit{ls}) \\
\ \ \textbf{or } \mathit{State}(z) \triangleright \textbf{match } z \textbf{ with} \\
\qquad\qquad\qquad \mid \_::\_ \rightarrow \mathit{State}_{x::xs}(z) \\
\qquad\qquad\qquad \mid \_ \rightarrow \mathit{State}_{ls}(z)
\end{array}
$$

One may believe that the matching of the pattern $x::\mathit{xs}$ needs to be performed twice (once in the dispatcher, once in the first reaction rule), but it is not necessary. The compiler should know that the matching of z against $x::\mathit{xs}$ in the first reaction rule cannot fail, and as a consequence, no test needs to be performed here, only the binding of the pattern variables. See Section 8.2 for details.

5. The compilation $[\![\cdot]\!]$

We formalize the intuitive idea described in Section 4 as a transformer Y_x, which transforms a join definition D with respect to channel x. The algorithm essentially works by constructing the meet semi-lattice of the formal pattern arguments of channel x in D, modulo pattern equivalence $\equiv$, with the less precise relation $\preceq$ being the partial order. Moreover, we visualize the lattice as a Directed Acyclic Graph (DAG), namely, vertices as patterns, and edges representing the partial order. If we reason more on instance sets than on patterns, this structure is quite close to the "subset graph" of [38].

Algorithm Y_x: Given D, a join definition, where x is a channel defined by D.

Step 0: Preprocess.

(1) Collect all the pattern arguments of channel x into the sequence:

$$\Pi_x = \pi_1^x; \pi_2^x; \ldots; \pi_n^x$$

(2) Let Π'_x be formed from Π_x by replacing all variables by wildcards "_" and taking the $\updownarrow$ of all equivalent patterns; thus Π'_x is a sequence of pairwise nonequivalent patterns.

(3) Perform exhaustiveness check on Π'_x, if not exhaustive, issue a warning.
(4) **IF:** There is only one pattern in Π'_x, and that Π'_x is exhaustive
THEN: goto Step 5. (In that case, no dispatching is needed.)

Step 1: Closure by least upper bound.
For any pattern π and pattern sequence $\Pi = \pi_1; \pi_2; \ldots; \pi_n$, we define $\pi \uparrow \Pi$ as the sequence $\pi \uparrow \pi_{i_1}; \pi \uparrow \pi_{i_2}; \ldots; \pi \uparrow \pi_{i_m}$, where the π_{i_k} s are the patterns from Π that are compatible with π.

We also define function F, which takes a pattern sequence Π as argument and returns a pattern sequence.
IF: Π is empty
THEN: $F(\Pi) = \Pi$
ELSE: Decompose Π as $\pi; \Pi'$ and state $F(\Pi) = \pi; F(\Pi'); \pi \uparrow F(\Pi')$

Compute the sequence $\Omega' = F(\Pi'_x)$. It is worth noticing that Ω' is the sequence of all valid patterns $(\pi^{x'}_{i_1} \uparrow \ldots (\pi^{x'}_{i_{k-1}} \uparrow \pi^{x'}_{i_k}) \ldots)$, with $1 \le i_1 < i_2 < \ldots < i_k \le n$, and $1 \le k \le n$, where we decompose Π'_x as $\pi^{x'}_1; \pi^{x'}_2; \ldots; \pi^{x'}_n$.

Step 2: Up to equivalence.
As in Step 0.2, build Ω by taking the $\updownarrow$ of all equivalent patterns in Ω'.

Step 3: Build DAG:
Corresponding to the semi-lattice $(\Omega, \preceq)$, build a directed acyclic graph $G(\mathcal{V}, \mathcal{E})$.
(1) $\mathcal{V} = \emptyset, \mathcal{E} = \emptyset$.
(2) For each pattern ω in Ω, add a new vertex v into $\mathcal{V}$ and annotate the vertex with ω.
(3) $\forall (v, v') \in \mathcal{V} \times \mathcal{V}, v \neq v'$, with annotations ω and ω' respectively, if $\omega \preceq \omega'$, then add an edge from v' to v into $\mathcal{E}$.

Step 4: Add dispatcher.
Following one topological order, the vertices of G are indexed as $v_1, \ldots, v_m$, correspondingly with annotations $\omega_1, \ldots, \omega_m$. We extend the join definition D with a dispatcher on channel x of the form: $x(z) \triangleright$ **match** z **with** Λ, where z is a fresh variable and Λ is built as follows:
(1) Let j ranges over $\{1, \ldots, m\}$. Following the topological order above, for all vertices v_j in $\mathcal{V}$ append a rule "$\mid \omega_j \to x_{\omega_j}(z)$" to Λ, where x_{ω_j} is a fresh channel name assigned to vertex v_j whose annotation is ω_j. Such fresh channels are here for the purpose of carrying messages originally sent to x then forwarded by the dispatcher, hence are also referred to as *forwarding channels*.
(2) If Π_x is not exhaustive, then add a rule "$\mid \_ \to 0$" at the end.

Step 5: Rewrite reaction rules.
For each reaction rule defining channel x in D: $J_i \,\&\, x(\pi^x_i) \triangleright Q_i$, we rewrite it according to the following policy. Let $Q'_i =$ **match** z_i **with** $\pi^x_i \to Q_i$, where z_i is a fresh variable.
IF: coming from Step 0
THEN: rewrite to $J_i \,\&\, x(z_i) \triangleright Q'_i$
ELSE:
(1) Let v_{j_i} be the unique vertex in $\mathcal{V}$, s.t. its annotation $\omega_{j_i} \equiv \pi^x_i$.
(2) We collect all the predecessors of v_{j_i} in G, and we record the indices of them, together with j_i, into a set that we note $I(\pi^x_i)$.
(3) Rewrite to $J_i \,\&(\bigvee_{j \in I(\pi^x_i)} x_{\omega_j}(z_i)) \triangleright Q'_i$, where $\bigvee$ is the generalized **or** construct of join patterns.

Given a join definition D, we note $\mathsf{dv}[D] = \{x_1, \ldots, x_n\}$ $(n \geq 0)$, that is we order the channel names arbitrarily. To transform D, we apply $Y_{x_n} \ldots Y_{x_1}(D)$. And the compilation of processes $[\![\cdot]\!]$ is inductively defined as follows:

$$
\begin{array}{rcl}
[\![0]\!] & \stackrel{\text{def}}{=} & 0 \\
[\![x(e)]\!] & \stackrel{\text{def}}{=} & x(e) \\
[\![P_1 \,\&\, P_2]\!] & \stackrel{\text{def}}{=} & [\![P_1]\!] \,\&\, [\![P_2]\!] \\
[\![\mathbf{def}\ D\ \mathbf{in}\ P]\!] & \stackrel{\text{def}}{=} & \mathbf{def}\ Y_{x_n} \ldots Y_{x_1}([\![D]\!])\ \mathbf{in}\ [\![P]\!] \\
[\![\mathbf{match}\ e\ \mathbf{with}|^{i \in I}\ \pi_i \to P_i]\!] & \stackrel{\text{def}}{=} & \mathbf{match}\ e\ \mathbf{with}|^{i \in I}\ \pi_i \to [\![P_i]\!] \\
\\
[\![\top]\!] & \stackrel{\text{def}}{=} & \top \\
[\![J \triangleright P]\!] & \stackrel{\text{def}}{=} & J \triangleright [\![P]\!] \\
[\![D_1\ \mathbf{or}\ D_2]\!] & \stackrel{\text{def}}{=} & [\![D_1]\!]\ \mathbf{or}\ [\![D_2]\!]
\end{array}
$$

Observe that the compilation preserves the interface of join definitions. Namely, it only affects the join definitions, never suppressing a channel, while message sending remains the same.

6. Example of compilation

Given the following join definition for an enriched integer stack:

$$
\begin{array}{l}
\mathbf{def}\ \mathit{push}(v)\ \&\ \mathit{State}(\mathit{ls}) \triangleright \mathit{State}\ (v::\mathit{ls}) \\
\ \ \mathbf{or}\ \mathit{pop}(r)\ \&\ \mathit{State}(x::\mathit{xs}) \triangleright r(x)\ \&\ \mathit{State}(\mathit{xs}) \\
\ \ \mathbf{or}\ \mathit{insert}(n)\ \&\ \mathit{State}(0::\mathit{xs}) \triangleright \mathit{State}(0::n::\mathit{xs}) \\
\ \ \mathbf{or}\ \mathit{last}(r)\ \&\ \mathit{State}(\texttt{[}x\texttt{]}) \triangleright r(x)\ \&\ \mathit{State}(\texttt{[}x\texttt{]}) \\
\ \ \mathbf{or}\ \mathit{swap}()\ \&\ \mathit{State}(x_1::x_2::\mathit{xs}) \triangleright \mathit{State}(x_2::x_1::\mathit{xs}) \\
\ \ \mathbf{or}\ \mathit{pause}(r)\ \&\ \mathit{State}(\texttt{[]}) \triangleright r() \\
\ \ \mathbf{or}\ \mathit{resume}(r) \triangleright \mathit{State}(\texttt{[]})\ \&\ r()
\end{array}
$$

The *insert* channel inserts an integer as the second topmost element, but only when the topmost element is 0. The *last* channel gives back the last element in the stack, keeping the stack unchanged. The *swap* channel exchange the topmost two elements in the stack. The *pause* channel temporarily freezes the stack when it is empty, while the *resume* channel brings the stack back into work. We now demonstrate our transformation with respect to channel *State*.

Step 0: We collect the pattern arguments of channel *State* into Π_{State}:

$$\Pi_{\mathit{State}} = \mathit{ls};\ x::\mathit{xs};\ 0::\mathit{xs};\ \texttt{[}x\texttt{]};\ x_1::x_2::\mathit{xs};\ \texttt{[]};\ \texttt{[]}$$

We drop the last equivalent `[]` pattern during the up to equivalence substep 0.2, and we get:

$$\Pi'_{\mathit{State}} = \mathit{ls};\ x::\mathit{xs};\ 0::\mathit{xs};\ \texttt{[}x\texttt{]};\ x_1::x_2::\mathit{xs};\ \texttt{[]}$$

Additionally, Π'_{State} is exhaustive (pattern *ls* alone covers all possibilities). Note that in the demonstration of this example, we sometimes keep variable names in patterns for readers' convenience. They are not necessary and are actually all replaced by "_" in the implementation.

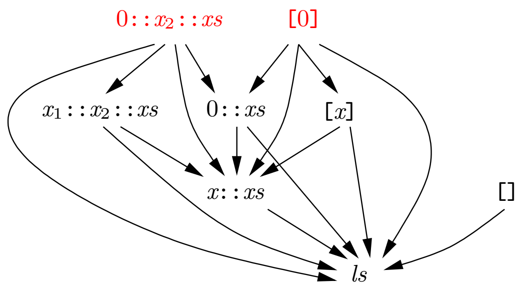
Step 1,2: Ω' extends Π'_{State} with all possible least upper bounds. Then we form Ω from Ω' by taking the $\updownarrow$ of all equivalent patterns.

$$\Omega = ls;\ x\mathtt{::}xs;\ 0\mathtt{::}xs;\ \mathtt{[}x\mathtt{]};\ x_1\mathtt{::}x_2\mathtt{::}xs;\ \mathtt{[]};\ 0\mathtt{::}x_2\mathtt{::}xs;\ \mathtt{[0]}$$

Note that the last two patterns are new, where:

$$\begin{array}{rcl} 0\mathtt{::}x_2\mathtt{::}xs & = & 0\mathtt{::}xs \uparrow x_1\mathtt{::}x_2\mathtt{::}xs \\ \mathtt{[0]} & = & 0\mathtt{::}xs \uparrow \mathtt{[}x\mathtt{]} \end{array}$$

Step 3: We build the semi-lattice $(\Omega, \preceq)$, see Figure 4.



1. $0\mathtt{::}x_2\mathtt{::}xs$
2. $\mathtt{[0]}$
3. $x_1\mathtt{::}x_2\mathtt{::}xs$
4. $0\mathtt{::}xs$
5. $\mathtt{[}x\mathtt{]}$
6. $x\mathtt{::}xs$
7. $\mathtt{[]}$
8. ls

Figure 4: The semi-lattice of patterns and the topological order

Step 4: One possible topological order of the vertices is also given at the right of Figure 4. Following that order, we build the dispatcher on channel *State*.

$$\begin{array}{l} \textbf{or } State(z) \triangleright \textbf{match } z \textbf{ with} \\ \quad \mid\ 0\mathtt{::\_::\_} \rightarrow State_1(z) \\ \quad \mid\ \mathtt{[0]} \rightarrow State_2(z) \\ \quad \mid\ \mathtt{\_::\_::\_} \rightarrow State_3(z) \\ \quad \mid\ 0\mathtt{::\_} \rightarrow State_4(z) \\ \quad \mid\ \mathtt{[\_]} \rightarrow State_5(z) \\ \quad \mid\ \mathtt{\_::\_} \rightarrow State_6(z) \\ \quad \mid\ \mathtt{[]} \rightarrow State_7(z) \\ \quad \mid\ \mathtt{\_} \rightarrow State_8(z) \end{array}$$

where $State_1$, ..., $State_8$ are the fresh forwarding channels.

Step 5: We rewrite the original reaction rules. As an example, consider the third reaction rule for the *insert* behavior: the pattern in $State(0\mathtt{::}xs)$ corresponds to vertex 4 with annotation $0\mathtt{::}xs$ in the graph, which has two predecessors: vertex 1 with annotation $0\mathtt{::}x_2\mathtt{::}xs$ and vertex 2 with annotation $\mathtt{[0]}$. Therefore, the reaction rule is rewritten to:

$$\begin{array}{l} insert(n)\ \&\ (State_1(z_3) \textbf{ or } State_2(z_3) \textbf{ or } State_4(z_3)) \\ \qquad \triangleright \textbf{match } z_3 \textbf{ with } 0\mathtt{::}xs \rightarrow State(0\mathtt{::}n\mathtt{::}xs) \end{array}$$

where z_3 is a fresh variable.

As a final result of our transformation, we get the disjunction of the following rules and of the dispatcher built in Step 4.

$$\begin{array}{l} \textbf{def } push(v)\ \&\ (State_1(z_1) \textbf{ or } \ldots \textbf{ or } State_8(z_1)) \\ \qquad \triangleright \textbf{match } z_1 \textbf{ with } ls \rightarrow State\ (v\mathtt{::}ls) \\ \textbf{or } pop(r)\ \&\ (State_1(z_2) \textbf{ or } \ldots \textbf{ or } State_6(z_2)) \\ \qquad \triangleright \textbf{match } z_2 \textbf{ with } x\mathtt{::}xs \rightarrow r(x)\ \&\ State(xs) \\ \textbf{or } insert(n)\ \&\ (State_1(z_3) \textbf{ or } State_2(z_3) \textbf{ or } State_4(z_3)) \end{array}$$

$$\begin{array}{l}
\qquad\qquad\qquad \triangleright\, \textbf{match}\ z_3\ \textbf{with}\ 0::xs \to \mathit{State}(0::n::xs) \\
\textbf{or}\ \mathit{last}(r)\ \&\ (\mathit{State}_2(z_4)\ \textbf{or}\ \mathit{State}_5(z_4)) \\
\qquad\qquad\qquad \triangleright\, \textbf{match}\ z_4\ \textbf{with}\ [x] \to r(x)\ \&\ \mathit{State}([x]) \\
\textbf{or}\ \mathit{swap}()\ \&\ (\mathit{State}_1(z_5)\ \textbf{or}\ \mathit{State}_3(z_5)) \\
\qquad\qquad\qquad \triangleright\, \textbf{match}\ z_5\ \textbf{with}\ x_1::x_2::xs \to \mathit{State}(x_2::x_1::xs) \\
\textbf{or}\ \mathit{pause}(r)\ \&\ \mathit{State}_7(z_6) \triangleright \textbf{match}\ z_6\ \textbf{with}\ [] \to r() \\
\textbf{or}\ \mathit{resume}(r) \triangleright \mathit{State}([])\ \&\ r()
\end{array}$$

7. Correctness

A program written in the applied join calculus of Section 3 is a process P. The compilation $[\![P]\!]$ replaces all the join definitions D in P by $Y_{x_n} \dots Y_{x_1}(D)$, where $\mathsf{dv}[D] = \{x_1, \dots, x_n\}$. To guarantee the correctness, we require the programs before and after the compilation be open equivalent. Namely, the following theorem should hold.

Theorem 7.1. *For any process P, $[\![P]\!] \approx P$.*

Proof. By structural induction on processes. Because $\approx$ is a full congruence and a transitive relation, it suffices to prove one step of the compilation, that is, Y_x is correct (see Lemma 7.2 below). □

Lemma 7.2. *For any join definition D, channel name $x \in \mathsf{dv}[D]$, and process P, we have:*

$$\textbf{def}\ D\ \textbf{in}\ P \approx \textbf{def}\ Y_x(D)\ \textbf{in}\ P$$

This lemma is crucial to the correctness of the compilation. We elaborate the proof in the coming sections. First, we recall the notations of algorithm Y_x in Section 7.1. Then, we discuss the properties of the dispatcher built by Y_x in Section 7.2. Finally, we prove Lemma 7.2 in Section 7.3.

7.1. Summary of notations.

We summarize the connection between the input and the output of Y_x. For simplicity, we omit the x superscripts everywhere. According to the algorithm given in Section 5, there are two cases during the procedure of Y_x, chosen at the end of Step 0:

Case "jump". The case where Steps 1 to 4 are skipped. Then, for any reaction rule of the form $J_i \,\&\, x(\pi_i) \triangleright Q_i$ of D, $i = 1 \dots n$, the pattern π_i is irrefutable, namely, $\pi_i \equiv \_$. And in $Y_x(D)$, we have the corresponding reaction rule $J_i \,\&\, x(z_i) \triangleright \textbf{match}\ z_i\ \textbf{with}\ \pi_i \to Q_i$, where z_i is fresh.

Case "go through". The general case. We recall the notations of the DAG $G(\mathcal{V}, \mathcal{E})$ built by the algorithm. G has m vertices, and following the topological order, the vertices are indexed as $v_1, \dots, v_m$ with pattern annotations $\omega_1, \dots, \omega_m$. Each vertex is also assigned a fresh forwarding channel, written x_{ω_j}.

For any reaction rule of the form $J_i \,\&\, x(\pi_i) \triangleright Q_i$ of D, $i = 1 \dots n$, there exists a unique vertex in G called v_{j_i}, such that its annotation $\omega_{j_i} \equiv \pi_i$. We use $I(\pi_i)$ to record the indices of the predecessors of v_{j_i} as well as j_i. Note that we have $\pi_i \preceq \omega_j$ iff $j \in I(\pi_i)$. In $Y_x(D)$, we have a corresponding reaction rule as $J_i \,\&(\bigvee_{j \in I(\pi_i)} x_{\omega_j}(z_i)) \triangleright \textbf{match}\ z_i\ \textbf{with}\ \pi_i \to Q_i$, where the variable z_i is fresh. Moreover, we add a dispatcher on channel x into $Y_x(D)$ as:

$$
\begin{array}{ll}
x(z) \triangleright \textbf{match } z \textbf{ with} & \\
\quad \mid \omega_1 \to x_{\omega_1}(z) & \\
\quad \mid \ldots & \\
\quad \mid \omega_m \to x_{\omega_m}(z) & \\
\quad \mid \_ \to 0 & (* \quad \mathit{if} \;\; \mathit{non-exhaustive} *)
\end{array}
$$

where z is a fresh variable.

7.2. **Property of the dispatcher.** We go on to discuss the property of the dispatcher built during the transformation on channel x. Let u range over closed expressions, that is over values. Modulo pattern equivalence $\equiv$, the patterns of the dispatcher $(\omega_j, j = 1, \ldots, m)$ are all the least upper bounds of the pattern arguments of channel x in the original D $(\pi_i, i = 1, \ldots, n)$. Thus, the π_i s and the ω_j s admit the same instances: $\bigcup_{1\leq i\leq n} \mathsf{Ins}(\pi_i) = \bigcup_{1\leq j\leq m} \mathsf{Ins}(\omega_j)$. As an immediate consequence, on one hand, for the set of values that do not match any of the original π_i s, written $\aleph = \{u \mid \forall i, u \notin \mathsf{Ins}(\pi_i)\}$, the values of $\aleph$ do not match any ω_j either, and those values are silently eaten by the dispatcher. On the other hand, given any value u such that there exists at least one π_i with $u \in \mathsf{Ins}(\pi_i)$, then the dispatcher must forward u onto one of the forwarding channels. More precisely, the following lemma holds.

Lemma 7.3. *For any value u that is an instance of some original pattern argument π_i, the dispatcher forwards u to the forwarding channel assigned to a vertex in G, whose index belongs to $I(\pi_i)$.*

Proof. We thus assume $u \in \mathsf{Ins}(\pi_i)$. Let K be the set of indices $\{k \mid u \in \mathsf{Ins}(\pi_k)\}$ and $\Pi_K = \{\pi_k \mid k \in K\}$. Let ω be the least upper bound of the patterns in Π_K, written $\uparrow\Pi_K$ (ω exists, since Π_K is non-empty). By steps 1–3 of the compilation algorithm Y_x, there must exist some vertex denoted by v_{j_K} in G with annotation $\omega_{j_K} \equiv \omega$. The dispatcher forwards message u onto the forwarding channel $x_{\omega_{j_K}}$, for the following two reasons.

(1) Value u is an instance of ω_{j_K}.
(2) No pattern of the dispatcher that appears before ω_{j_K} admits u as an instance. Namely, any pattern of the dispatcher $\omega_j, 1 \leq j \leq m$, such that $u \in \mathsf{Ins}(\omega_j)$ must be the least upper bound of a subset of Π_K. Then, since the patterns of the dispatchers are ordered topologically (with precision order $\preceq$), ω_{j_K} must be the foremost pattern in the dispatcher which has u as an instance. Namely, precision order $\preceq$ applied to least upper bounds is reverse set inclusion applied to instance sets.

Moreover, because $\pi_i \in \Pi_K$ and $\omega_{j_K} \equiv \omega = \uparrow\Pi_K$, we have $\pi_i \preceq \omega_{j_K}$. Thus, by definition of $I(\pi_i)$, we have $j_K \in I(\pi_i)$. $\square$

In the following, given some value u, we write x_u for the forwarding channel to which u is sent by the dispatcher. Using the new notation, Lemma 7.3 is reformulated as follows: if $u \in \mathsf{Ins}(\pi_i)$, then x_u exists and we have $x_u \in \{x_{\omega_j} \mid j \in I(\pi_i)\}$.

7.3. **Proof of Lemma 7.2.**

Proof. Following the definition of $\eqsim$, we should prove $(\textbf{def } D \textbf{ in } P)\sigma \approx (\textbf{def } Y_x(D) \textbf{ in } P)\sigma$, for any closing substitution σ. In other words, since $Y_x(D)\sigma = Y_x(D\sigma)$, we should prove:

$$\textbf{def } D\sigma \textbf{ in } P\sigma_1 \approx \textbf{def } Y_x(D\sigma) \textbf{ in } P\sigma_1 \tag{7.1}$$

where σ_1 is σ minus the (possible) bindings of the variables of $\mathsf{dv}[D]$. Notice that all subcomponents $D\sigma$, $Y_x(D\sigma)$ and $P\sigma_1$ are closed. Hence, to prove (7.1), it suffices to prove that Y_x is correct for closed terms (Lemma 7.4 below). $\square$

Lemma 7.4. *For any closed join definition D, channel name $x \in \mathsf{dv}[D]$, and closed process P, we have:*

$$\mathbf{def}\ D\ \mathbf{in}\ P \approx \mathbf{def}\ Y_x(D)\ \mathbf{in}\ P$$

Proof. There are two subcases.

Case "go through". We construct the following relation $\mathcal{R}$:

$$\mathcal{R}\!= \{(\mathbf{def}\ D\ \mathbf{in}\ (P \,\&\, Q),\ \mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P \,\&\, \widehat{Q}))\}$$

Above, process P and definition D range respectively over closed processes and closed definitions; while Q and $\widehat{Q}$ are particular. Dissect the structure of D as:

$$D = \ldots\ \mathbf{or}\ J_i \,\&\, x(\pi_i) \triangleright Q_i\ \mathbf{or}\ \ldots$$

We define Q and $\widehat{Q}$ to be:

$$Q = (\textstyle\prod_{\delta\in\Delta} x(\pi_i\delta)) \,\&\, (\prod_{\psi\in\Psi} Q_i\psi) \,\&\, (\prod_{u\in U} x(u))$$
$$\widehat{Q} = (\textstyle\prod_{\delta\in\Delta} x_{\pi_i\delta}(\pi_i\delta)) \,\&\, (\prod_{\psi\in\Psi} \mathbf{match}\ \pi_i\psi_{\pi_i}\ \mathbf{with}\ \pi_i \to Q_i\psi_{J_i})$$

We note $\prod$ the generalized parallel composition. Note that processes Q and $\widehat{Q}$ are (implicitly) parameterized by the multisets of substitutions Δ and Ψ, and by the multiset of values U. In the definition of $\mathcal{R}$, Δ, Ψ and U range over all appropriate multisets. More precisely, given any reaction rule $J_i \,\&\, x(\pi_i) \triangleright Q_i$ from D, we note δ any (closed) substitution on domain $\mathsf{rv}[\pi_i]$. Then, Δ stands for any multiset of such substitutions δ. Similarly, let ψ be a (closed) substitution on domain $\mathsf{rv}[J_i] \uplus \mathsf{rv}[\pi_i]$. Moreover, for any such ψ, let ψ_{π_i} be $\psi \restriction \mathsf{rv}[\pi_i]$ (the restriction of ψ on domain $\mathsf{rv}[\pi_i]$), and ψ_{J_i} be $\psi \restriction \mathsf{rv}[J_i]$. Because $\mathsf{rv}[\pi_i] \cap \mathsf{rv}[J_i] = \emptyset$, the substitution ψ is the sum of ψ_{π_i} and ψ_{J_i}, written $\psi = \psi_{\pi_i} \uplus \psi_{J_i}$, and we further require $\psi_{\pi_i} \circ \psi_{J_i} = \psi_{J_i} \uplus \psi_{\pi_i}$. Then, Ψ is any multiset of such substitutions ψ. Finally, U is a multiset of elements from $\aleph$.

Intuitively, we use Q and $\widehat{Q}$ to bridge the differences caused by D and $Y_x(D)$. More specifically: a message $x(\pi_i\delta)$ may be forwarded to $x_{\pi_i\delta}(\pi_i\delta)$ by the dispatcher in $Y_x(D)$; furthermore, if a guarded process $Q_i\psi$ is triggered from D, then from $Y_x(D)$, we have the corresponding guarded process $\mathbf{match}\ \pi_i\psi_{\pi_i}\ \mathbf{with}\ \pi_i \to Q_i\psi_{J_i}$ triggered; finally, a message on channel x with a non-matching content, that is from $\aleph$, will be eaten by $Y_x(D)$.

We analyze the following three aspects of $\mathcal{R}$: closure by (closed) evaluation contexts; reduction bisimulation; and preservation of barbs.

- $\mathcal{R}$ is closed by closed evaluation contexts up to $\equiv$ (*i.e.* Lemma 3.11.(1)). For any closed evaluation context $E[\cdot]$, we have:

  $$E[\mathbf{def}\ D\ \mathbf{in}\ (P \,\&\, Q)] \equiv \mathbf{def}\ D\ \mathbf{or}\ D'\ \mathbf{in}\ (P \,\&\, P' \,\&\, Q)$$
  $$E[\mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P \,\&\, \widehat{Q})] \equiv \mathbf{def}\ Y_x(D)\ \mathbf{or}\ D'\ \mathbf{in}\ (P \,\&\, P' \,\&\, \widehat{Q})$$

  where $\mathsf{dv}[D] \cap \mathsf{dv}[D'] = \emptyset$, so that $Y_x(D)\ \mathbf{or}\ D' = Y_x(D\ \mathbf{or}\ D')$. Therefore, we have $E[\mathbf{def}\ D\ \mathbf{in}\ (P \,\&\, Q)] \equiv\!\mathcal{R}\!\equiv E[\mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P \,\&\, \widehat{Q})]$.
- $\mathcal{R}$ is a reduction bisimulation (*i.e.* a special case of Lemma 3.11.(2) because the identity in included in $\equiv$). We only detail the nontrivial cases.

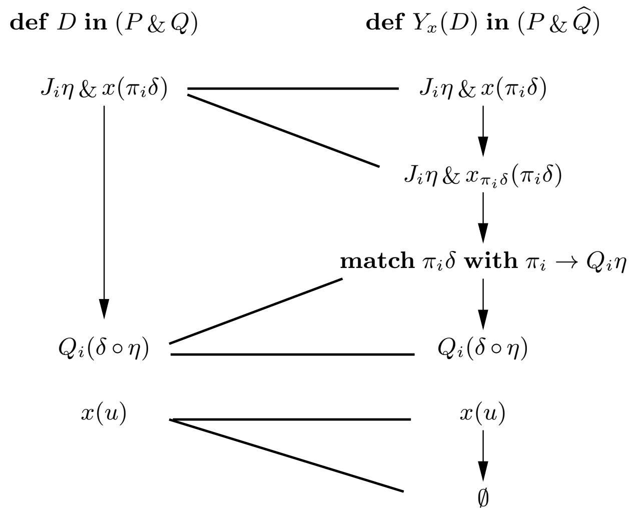


Figure 5: Reduction chasing in case "go through"

(1) If there is a message $x(\pi_i\delta')$ in P, the right part can forward it to a message $x_{\pi_i\delta'}(\pi_i\delta')$ by the dispatcher in $Y_x(D)$. This reduction is simulated in the left part by no reduction, and we add the new substitution δ' into Δ.
(2) Similarly, if there is a message $x(u')$ in P, for some $u' \in \aleph$, the right part can eat the message by the dispatcher in $Y_x(D)$. This reduction is simulated by no reduction in the left part and we add u' into U.
(3) If a reduction according to the reaction rule $J_i \,\&\, x(\pi_i) \triangleright Q_i$ consumes a molecule $J_i\eta \,\&\, x(\pi_i\delta)$ in the left part, for some $\delta \in \Delta$ (*i.e.* $x(\pi_i\delta)$ occurs in Q) and $J_i\eta$ from P, with $\mathsf{dom}(\eta) = \mathsf{rv}[J_i]$; it can be simulated by consuming $J_i\eta \,\&\, x_{\pi_i\delta}(\pi_i\delta)$ in the right part, using the corresponding reaction rule $J_i \,\&(\bigvee_{j \in I(\pi_i)} x_{\omega_j}(z_i)) \triangleright$**match** z_i **with** $\pi_i \to Q_i$, because $x_{\pi_i\delta} \in \{x_{\omega_j} \mid j \in I(\pi_i)\}$ (Lemma 7.3). The derivatives are still in $\mathcal{R}$, with Δ shrinking to $\Delta \setminus \{\delta\}$, and Ψ expanding to $\Psi \cup \{\eta \uplus \delta\}$. We assume α-conversion when necessary to guarantee $\delta \circ \eta = \eta \uplus \delta$. Vice versa.
(4) Similar to the previous case but this time the left part consumes a molecule $J_i\eta \,\&\, x(\pi_i\delta')$, where δ' is not from Δ. Then, the right part simulates this reduction by first forwarding the message $x(\pi_i\delta')$ to the message $x_{\pi_i\delta'}(\pi_i\delta')$ as in case ?2, then consuming the molecule $J_i\eta \,\&\, x_{\pi_i\delta'}(\pi_i\delta')$. Ψ expands to $\Psi \cup \{\eta \uplus \delta'\}$.
(5) The **match** $\pi_i\psi_{\pi_i}$ **with** $\pi_i \to Q_i\psi_{J_i}$ in $\widehat{Q}$ of the right part can be reduced to $(Q_i\psi_{J_i})\psi_{\pi_i}$ by the semantic rule Match. Because we have $\psi_{\pi_i} \circ \psi_{J_i} = \psi_{J_i} \uplus \psi_{\pi_i}$, the result of the reduction equals to $Q_i(\psi_{J_i} \uplus \psi_{\pi_i})$, that is $Q_i\psi$. This reduction is simulated by no reduction in the left part. However, the process P becomes $P \,\&\, Q_i\psi$, and Ψ shrinks to $\Psi \setminus \{\psi\}$.
(6) If a reduction involves $Q_i\psi$ from Q of the left part, for some $\psi \in \Psi$, it can be simulated by first reducing the correspondent **match** $\pi_i\psi_{\pi_i}$ **with** $\pi_i \to Q_i\psi_{J_i}$ from $\widehat{Q}$ into $Q_i\psi$ as in the previous case.

Figure 5 summarizes the various cases we just examined, where thick lines express the $\mathcal{R}$ relation.

$$\begin{tikzcd}
\mathbf{def}\ D\ \mathbf{in}\ (P\,\&\,Q) & \mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P\,\&\,\widehat{Q}) \\
J_i\eta\,\&\,x(\pi_i\delta) \arrow[r, dash] \arrow[dd] & J_i\eta\,\&\,x(\pi_i\delta) \arrow[d] \\
 & \mathbf{match}\ \pi_i\delta\ \mathbf{with}\ \pi_i \to Q_i\eta \arrow[d] \arrow[dl, dash] \\
Q_i(\delta\circ\eta) \arrow[r, dash] & Q_i(\delta\circ\eta)
\end{tikzcd}$$

Figure 6: Reduction chasing in case "jump"

- $\mathcal{R}$ preserves barbs (*i.e.* Lemma 3.11.(3)). We demonstrate $\mathbf{def}\ D\ \mathbf{in}\ (P\,\&\,Q)\downarrow_y \implies \mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P\,\&\,\widehat{Q})\Downarrow_y$ and vice versa. We distinguish the cases that make $\mathbf{def}\ D\ \mathbf{in}\ (P\,\&\,Q)\downarrow_y$ hold.
  (1) $Q\downarrow_y$. We have $y \notin \mathsf{dv}[D]$. Because all variables in $\mathsf{dv}[Y_x(D)] \setminus \mathsf{dv}[D]$ are fresh, we also have $y \notin \mathsf{dv}[Y_x(D)]$. According to the structure of Q, we must have $Q_i\psi\downarrow_y$ for some $\psi \in \Psi$. Then in $\widehat{Q}$, we have $\mathbf{match}\ \pi_i\psi_{\pi_i}\ \mathbf{with}\ \pi_i \to Q_i\psi_{J_i}$ reduces to $Q_i\psi$ and $Q_i\psi\downarrow_y$. That is, $(\mathbf{match}\ \pi_i\psi_{\pi_i}\ \mathbf{with}\ \pi_i \to Q_i\psi_{J_i})\Downarrow_y$, *i.e.* $\widehat{Q}\Downarrow_y$, *i.e.* $\mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P\,\&\,\widehat{Q})\Downarrow_y$.
  (2) $P\downarrow_y$. Obvious.
  The proof of the other direction, *i.e.* $\mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P\,\&\,\widehat{Q})\downarrow_y \implies \mathbf{def}\ D\ \mathbf{in}\ (P\,\&\,Q)\Downarrow_y$, is obvious since the only case for $\mathbf{def}\ Y_x(D)\ \mathbf{in}\ (P\,\&\,\widehat{Q})\downarrow_y$ is when $P\downarrow_y$.

  Following the analysis above, $\mathcal{R}$ is a weak barbed congruence up to $\equiv$. By Lemma 3.12, we have $\mathcal{R}$ is a weak barbed congruence.

  Let Δ, Ψ and U be empty sets. We have the two processes of Lemma 7.4 satisfy relation $\mathcal{R}$, hence $\approx$. That is, we proved that Lemma 7.4 holds for case "go through".

Case "jump". We build another relation $\mathcal{R}$, with Q and $\widehat{Q}$ defined as follows:

$$Q = \prod\nolimits_{\psi\in\Psi} Q_i\psi$$
$$\widehat{Q} = \prod\nolimits_{\psi\in\Psi} \mathbf{match}\ \pi_i\psi_{\pi_i}\ \mathbf{with}\ \pi_i \to Q_i\psi_{J_i}$$

and we summarize the property of reduction bisimulation by the diagram of Figure 6. □

8. Implementing applied join

We carried out the practical implementation work of the applied join calculus as an extension of the JoCaml system. The extended system is publicly released [31]. The release includes a tutorial that makes extensive use of algebraic patterns in join patterns. In this section, we first sketch out the structure of the extended JoCaml compiler, pointing out where the transformation should take place. Then some optimizations of our algorithm Y_x are reported.

8.1. **Front end of the (extended) JoCaml compiler.** The JoCaml compiler is an extension of the OCaml compiler, as the JoCaml language is an extension of the OCaml language. Extensions are confined to the first four phases of the compiler.

More precisely, there are additional tokens in the lexer (such as the keyword **def**). Then, all the constructs of Figure 1 are parsed and rendered as specific constructs in the abstract syntax tree. Typed syntax undergoes a similar extension. Amongst those first three compiler phases, only the typer significantly differs from the original OCaml compiler, since the JoCaml compiler has to deal with the specific rules for typing the join calculus polymorphically [18]. Finally, the typed syntax is translated to *lambda-code*, which basically is λ-calculus enriched with primitive types and calls to the runtime library. All constructs specific to JoCaml disappear, being replaced by calls to specific primitives in a "Join" library, built on top of one of the OCaml thread libraries. In the following, we denote as "the JoCaml runtime", the ordinary (thread aware) OCaml runtime, plus the thread library, plus the Join library. To summarize, extending the OCaml system to the JoCaml system amounts to modifying the front end of the compiler, and to writing the Join library.

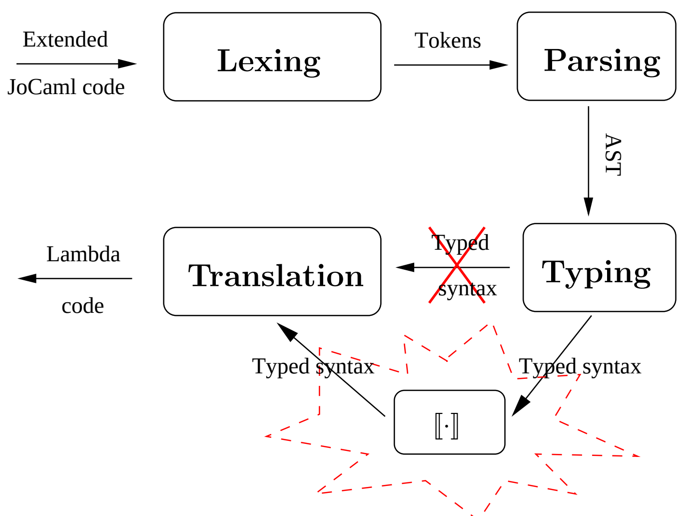


Figure 7: The extended JoCaml compiler front end

Extending JoCaml to handle pattern arguments in join definitions requires further modifications. Figure 7 shows the structure of the extended JoCaml compiler. With respect to plain JoCaml (without algebraic pattern matching in join definitions), the parser and the typer have to be modified to take pattern arguments in channel definitions into account. However these extensions are mechanical. The critical modification manifests itself as an extra sub-phase (enclosed in the dashed polygon) between the typing phase and the translation phase. Not surprisingly, the additional phase carries out the transformation from extended join definitions to plain ones, by implementing the compilation scheme $[\![\cdot]\!]$ of Section 5. Once this new transformation is performed, all join definitions in the typed trees are plain ones (without pattern arguments). Then, the translator to lambda-code and, more importantly, the JoCaml runtime system need not be changed, with respect to the ones of the original JoCaml system.

We in fact also slightly extended the translator, for the sake of performing a few optimizations (see Section 8.2) and of avoiding excessive duplications of guarded processes (see

Section 8.3). The optimizations we perform make use of the sophisticated pattern matching compiler and analyzer that are already present in the standard OCaml compiler.

8.2. Matching optimizations.

8.2.1. *Avoiding redundant matchings.* As discussed at the end of Section 4, the compilation introduces redundant matchings. For instance, in the stack example, we get:

$$
\begin{array}{l}
\textbf{def } \mathit{pop}(r) \;\&\; \mathit{State}_{x::xs}(z) \triangleright \textbf{match } z \textbf{ with } x::xs \rightarrow r(x) \;\&\; \mathit{State}(xs)\\
\quad \ldots\\
\textbf{or } \mathit{State}(y) \triangleright \textbf{match } y \textbf{ with}\\
\qquad\qquad\qquad \mid \; \_::\_ \rightarrow \mathit{State}_{x::xs}(y)\\
\qquad\qquad\qquad\quad \ldots
\end{array}
$$

A *pop* operation apparently involves matching the *State* argument twice: once in the dispatcher, to select the appropriate forwarding channel $\mathit{State}_{x::xs}$, and again in the reaction rule, to perform the bindings of variables x and xs to the head and tail of the cons-cell z.

However, by construction, the value of argument z is guaranteed to be an instance of the pattern $x::xs$. This remark is general (see Lemma 7.3): for any matching **match** z_i **with** $\pi_i \rightarrow Q_i$ introduced in reaction rules by Step 5 of algorithm Y_x, the value of z_i always matches the pattern π_i. In other words, the matching **match** z_i **with** $\pi_i \rightarrow Q_i$ never fails, hence no test need to be performed at all. As a consequence, in the case of the pattern $x::xs$, we aim at getting the the following lambda-code:[5]

$$
\begin{array}{l}
\textbf{let } x = \mathit{field}\ 0\ z \textbf{ in}\\
\textbf{let } xs = \mathit{field}\ 1\ z \textbf{ in}\\
\ldots
\end{array}
$$

Primitives "*field* 0 *z*" and "*field* 1 *z*" extract the head and tail from the cons-cell z.

The requirement is then to write a specific matching compiler that does not issue tests when test outputs can be predicted at compile time. In fact, such a matching compiler is already present in the OCaml compiler: as it stands, the optimizing pattern matching compiler of [26] can output such code, provided it is informed that the compiled matching has only one clause and never fails, which is exactly the case for all the matchings **match** z_i **with** $\pi_i \rightarrow Q_i$ introduced in reaction rules by Step 5 of algorithm Y_x. Incidentally, the condition "the matching can never fail" is expressed simply as "the matching is exhaustive". We also rely on a later phase of the OCaml compiler to inline **let**-bindings when appropriate.

As a final remark, it is worth observing that, when the original pattern does not contain variables, the compilation of **match** z_i **with** $\pi_i \rightarrow \cdots$ yields no code: neither test, nor binding.

8.2.2. *Avoiding useless forwarding channels.* Simple analysis of the dispatcher matching enables use to spare some of the forwarding channels. Let us first re-consider the example of the complete stack. Our transformer Y applied to channel *State* yields the following dispatcher:

[5] In examples, we show lambda-code as OCaml code, enriched with a few primitives.

$$
\begin{array}{l}
\textbf{or } \mathit{State}(z) \triangleright \textbf{match } z \textbf{ with} \\
\qquad\qquad\qquad \mid 0::\_::\_ \to \mathit{State}_1(z) \\
\qquad\qquad\qquad \mid [0] \to \mathit{State}_2(z) \\
\qquad\qquad\qquad \mid \_::\_::\_ \to \mathit{State}_3(z) \\
\qquad\qquad\qquad \mid 0::\_ \to \mathit{State}_4(z) \\
\qquad\qquad\qquad \mid [\_] \to \mathit{State}_5(z) \\
\qquad\qquad\qquad \mid \_::\_ \to \mathit{State}_6(z) \\
\qquad\qquad\qquad \mid [] \to \mathit{State}_7(z) \\
\qquad\qquad\qquad \mid \_ \to \mathit{State}_8(z)
\end{array}
$$

In the matching above, some clauses are never matched at runtime. For instance, the last clause "$\_ \to \mathit{State}_8(z)$" is useless, because of the two immediately preceding clauses "$\_::\_\to \ldots$" and "$[] \to \ldots$" that obviously match all the lists. As a consequence, the forwarding channel State_8 never carries any message hence it is also useless. Similarly, channels State_4 and channel State_6 are useless. We can optimize by removing both the useless clauses from the dispatcher and all occurrences of useless channels from the rewritten join patterns.

To summarize, by applying the optimizations discussed so far, the stack example after compilation looks as follows:

$$
\begin{array}{l}
\textbf{def } \mathit{push}(v) \ \&\ (\mathit{State}_1(z_1) \textbf{ or } \mathit{State}_2(z_1) \textbf{ or } \mathit{State}_3(z_1) \textbf{ or } \mathit{State}_5(z_1) \textbf{ or } \mathit{State}_7(z_1)) \\
\qquad \triangleright \mathit{State}\ (v::z_1) \\
\ \textbf{or } \mathit{pop}(r) \ \&\ (\mathit{State}_1(z_2) \textbf{ or } \mathit{State}_2(z_1) \textbf{ or } \mathit{State}_3(z_1) \textbf{ or } \mathit{State}_5(z_1)) \\
\qquad \triangleright r(\mathit{field}\ 0\ z_2) \ \&\ \mathit{State}(\mathit{field}\ 1\ z_2) \\
\ \textbf{or } \mathit{insert}(n) \ \&\ (\mathit{State}_1(z_3) \textbf{ or } \mathit{State}_2(z_3)) \\
\qquad \triangleright \mathit{State}(0::n::\mathit{field}\ 1\ z_3) \\
\ \textbf{or } \mathit{last}(r) \ \&\ (\mathit{State}_2(z_4) \textbf{ or } \mathit{State}_5(z_4)) \\
\qquad \triangleright \textbf{let } x = \mathit{field}\ 0\ z_4 \textbf{ in } r(x) \ \&\ \mathit{State}([x]) \\
\ \textbf{or } \mathit{swap}() \ \&\ (\mathit{State}_1(z_5) \textbf{ or } \mathit{State}_3(z_5)) \\
\qquad \triangleright \textbf{let } m = \mathit{field}\ 1\ z_5 \textbf{ in } \mathit{State}(\mathit{field}\ 0\ m::\mathit{field}\ 0\ z_5::\mathit{field}\ 1\ m) \\
\ \textbf{or } \mathit{pause}(r) \ \&\ \mathit{State}_7(z_6) \triangleright r() \\
\ \textbf{or } \mathit{resume}(r) \triangleright \mathit{State}([]) \ \&\ r() \\
\ \textbf{or } \mathit{State}(z) \triangleright \textbf{match } z \textbf{ with} \\
\qquad\qquad\qquad \mid 0::\_::\_ \to \mathit{State}_1(z) \\
\qquad\qquad\qquad \mid [0] \to \mathit{State}_2(z) \\
\qquad\qquad\qquad \mid \_::\_::\_ \to \mathit{State}_3(z) \\
\qquad\qquad\qquad \mid [\_] \to \mathit{State}_5(z) \\
\qquad\qquad\qquad \mid [] \to \mathit{State}_7(z)
\end{array}
$$

Thanks to the optimization, three cases are spared from the dispatcher, three channels are not allocated, and the size of the **or** join-patterns decrease significantly.

To integrate this optimization into the implementation, we modify the algorithm Y_x, as regards dispatcher construction (Step 4) and rewriting of reaction rules (Step 5). In Step 4, after the topological sort, we check the usefulness of each vertex. More specifically, to check whether vertex v_k is useful or not, with respect to the preceding vertices $v_1, \ldots, v_{k-1}$ in the topological order, we check the usefulness of pattern ω_k with respect to patterns $\omega_1, \ldots, \omega_{k-1}$, where ω_i is the annotation pattern of vertex v_i. For that purpose, we use the standard usefulness checker of OCaml [30], of which we present a simplified version in Section 2.1. Then, in Step 5 of the algorithm we retain only the v_k's that are useful.

8.3. **Compiling or in join patterns.** The compilation scheme $[\![\cdot]\!]$ introduces disjunctive composition into join patterns, a construct that JoCaml did not support before the introduction of pattern argument in join definitions. In this section, we describe our extensions to the JoCaml compiler so as to integrate this new feature.

When we introduced **or** in join patterns, we claimed that it is syntactic sugar. That is, we define this new construct by distributing & over **or**, until **or** reaches the reaction rule level, where we finally duplicate the reaction rules themselves.

$$(J_1 \textbf{ or } J_2 \textbf{ or } \cdots \textbf{ or } J_n) \triangleright P \stackrel{\text{def}}{=} (J_1 \triangleright P) \textbf{ or } (J_2 \triangleright P) \textbf{ or } \cdots \textbf{ or } (J_n \triangleright P)$$

The whole process of distributing & over **or** and of duplicating the rules can be summarized as "expansion of **or** in join patterns".

It is not difficult to see that the above mentioned expansion easily produces an exponential number of reaction rules. For instance, consider the definition:

$$\begin{array}{l} \textbf{def } a^1(\textbf{true}) \triangleright P_1 \textbf{ or } \ a^2(\textbf{true}) \triangleright P_2 \ldots \textbf{ or } a^n(\textbf{true}) \ \triangleright P_n \\ \textbf{or } a^1(\_) \ \& \ a^2(\_) \ \& \cdots \& \ a^n(\_) \triangleright P_0 \end{array}$$

For each channel a^i there are two forwarding channels $a^i_{\textbf{true}}$ and $a^i_{\_}$. As a consequence, after rewriting, the last reaction rule from the definition above becomes:

$$\textbf{or } (a^1_{\textbf{true}}(z_1) \textbf{ or } a^1_{\_}(z_1)) \ \& \ (a^2_{\textbf{true}}(z_2) \textbf{ or } a^2_{\_}(z_2)) \ \& \cdots \& \ (a^n_{\textbf{true}}(z_n) \textbf{ or } a^n_{\_}(z_n)) \triangleright P_0$$

And the expansion of **or** in join patterns finally yields 2^n reaction rules.

The extended JoCaml compiler indeed performs the expansion of **or** in join patterns as sketched above, except for one point: the guarded processes (P_0 in example) is not duplicated. Instead, guarded processes are compiled into (lambda-code) closures and duplication of guarded processes is performed by duplication of pointers to those closures.

We will illustrate two successive refinements of the idea of sharing guarded processes. But before that, let us first examine how guarded processes are compiled and triggered in the general case.

$$\begin{array}{l} \textbf{def } a(x) \ \& \ b(y) \triangleright P \\ \textbf{or } \ a(x) \ \& \ c(y) \triangleright Q \end{array}$$

The above join definition defines three channels organized in two reaction rules. Target lambda-code can be sketched as follows:

$$\begin{array}{l}
\textbf{let } \mathit{jdef} = \\
\quad \ldots\ldots \\
\quad \textbf{let } g_{\{a,b\}} = \textbf{fun } \mathit{jdef} \to \\
\qquad\qquad \textbf{let } x = \mathit{Join.get_queue\ jdef}\ i_a \textbf{ in} \\
\qquad\qquad \textbf{let } y = \mathit{Join.get_queue\ jdef}\ i_b \textbf{ in} \\
\qquad\qquad \mathit{Join.unlock\ jdef}; \\
\qquad\qquad \mathit{Join.spawn}\ (\textbf{fun}\ () \to [\![P]\!]_\lambda) \textbf{ in} \\
\quad \textbf{let } g_{\{a,c\}} = \textbf{fun } \mathit{jdef} \to \\
\qquad\qquad \textbf{let } x = \mathit{Join.get_queue\ jdef}\ i_a \textbf{ in} \\
\qquad\qquad \textbf{let } y = \mathit{Join.get_queue\ jdef}\ i_c \textbf{ in} \\
\qquad\qquad \mathit{Join.unlock\ jdef}; \\
\qquad\qquad \mathit{Join.spawn}\ (\textbf{fun}\ () \to [\![Q]\!]_\lambda) \textbf{ in} \\
\quad \ldots
\end{array}$$

The presented lambda-code only describes the compilation of guarded processes to closures $g_{\{a,b\}}$ and $g_{\{a,c\}}$. Those *guarded closures* are subparts of the complete compilation of the join definition. They appear as local bindings in the more complete definition *jdef*, which is not shown. We refer to [25] for a full explanation about how the JoCaml compiler deals with join definitions and guarded processes. Nevertheless, we give a brief description, based upon the example. Join definitions are compiled into vector-like structures, and channels are pairs of a pointer to such a structure and of a *channel slot* (written i_a etc. above). Channel slots are small integers. Here, we assume i_a to be 0, i_b to be 1, and i_c to be 2. Based upon channel slots, join patterns are compiled into bitsets. In this example, we have 110 for pattern "$a(x)$ & $b(y)$" and 101 for "$a(x)$ & $c(y)$". The join definition runtime structure holds a list of pairs made of such a bitset and of a pointer to a guarded closure (`[`(110,$g_{\{a,b\}}$; (101,$g_{\{a,c\}}$)`]` in our example). This *join matching list* can be seen as the result of reaction rules compilation. The definition structure also holds a mutex, an array of queues (indexed by channel slots), and an internal bitset that describes the current status of queues. In response to message sending over a channel, specific code from the Join library first locks the mutex, alters the internal bitset, stores the message in the appropriate queue, and then attempt a match. In case a match is found, the corresponding closure ($g_{\{a,b\}}$ or $g_{\{a,c\}}$ above) is called, with the definition itself as an argument.

Notice that the closures $g_{\{a,b\}}$ or $g_{\{a,c\}}$ have the responsibility to bind formal arguments x and y to the appropriate actual arguments, which are extracted from the appropriate queues (lines 4–5) and 9–10), and to release the mutex (lines 6 and 11). The guarded process is finally triggered by the means of the primitive *Join.spawn* that takes a closure as argument (lines 7 and 12) and creates a new thread to run that closure. Here, $[\![P]\!]_\lambda$ and $[\![Q]\!]_\lambda$ represent the compilation to lambda-code of P and Q respectively. It is to be noticed that formal parameters may occur free in P and Q.

Now let us consider the compilation of join definitions with **or** in their join patterns, such as this one: **def** $a(x)$ & $(b(y)$ **or** $c(y)) \triangleright P$. Target lambda-code can be sketched as follows:

```
1   let jdef =
2     ......
3     let p = fun jdef x y →
4                   Join.unlock jdef;
5                   Join.spawn (fun () → [[P]]λ) in
6     let g{a,b} = fun jdef →
7                   let x = Join.get_queue jdef i_a in
8                   let y = Join.get_queue jdef i_b in
9                   p jdef x y in
10    let g{a,c} = fun jdef →
11                  let x = Join.get_queue jdef i_a in
12                  let y = Join.get_queue jdef i_c in
13                  p jdef x y in
14    ......
```

As a consequence of the expansion of the disjunctive pattern "$b(y)$ **or** $c(y)$", the join matching list is `[`(110,$g_{\{a,b\}}$; (101,$g_{\{a,c\}}$)`]`, like in the previous example. The two guarded closures $g_{\{a,b\}}$ and $g_{\{a,c\}}$ are different, because the value bound to the formal argument y has to be extracted either from the queue of channel b or from the queue of channel c, depending upon the matched join pattern being "$a(x)$ & $b(y)$" or "$a(x)$ & $c(y)$". However, the task of unlocking the mutex and of triggering the process P is common to both and is performed by a third closure p (lines 3–5), which is called by the two guarded closures $g_{\{a,b\}}$ and $g_{\{a,c\}}$ at lines 9 and 13 respectively. As a result, duplication of most of the guarded process code is avoided and a reasonable amount of sharing is achieved. One should observe that the interface between the library code that performs join matching and the guarded closures is preserved: guarded closures are still functions that take a join structure as argument.

It is in fact possible for the compiler to completely share guarded closures between reactions rules that originate from **or** pattern expansion. But then, guarded closure code must be abstracted further with respect to the exact join pattern that is matched. The idea of dictionary can be used for this purpose. A *dictionary* is an array built by the compiler. Dictionaries represent mappings from formal parameters to channel slots and the compilation of a join pattern now yields a pair of a bitset and of a dictionary. More significantly, disjunctive patterns are now compiled into a series of such pairs. For instance, the pattern "$a(x)$ & ($b(y)$ **or** $c(y)$)" is now compiled into the two pairs "$(110, \texttt{[}|0\ ;\ 1|\texttt{]})$" and "$(101, \texttt{[}|0\ ;\ 2|\texttt{]})$", where for instance the dictionary component "$\texttt{[}|0\ ;\ 2|\texttt{]}$" expresses that the formal parameters x and y are to be bound to messages sent on channels a (at slot 0) and c (at slot 2) respectively. The compiler then generates guarded closures abstracted with respect to dictionaries.

$$
\begin{array}{l}
\textbf{let } g_{\{a,(b|c)\}} = \textbf{fun } \mathit{jdef}\ \mathit{dict} \rightarrow \\
\qquad\qquad \textbf{let } x = \mathit{Join.get\_queue}\ \mathit{jdef}\ (\mathit{field}\ 0\ \mathit{dict})\ \textbf{in} \\
\qquad\qquad \textbf{let } y = \mathit{Join.get\_queue}\ \mathit{jdef}\ (\mathit{field}\ 1\ \mathit{dict})\ \textbf{in} \\
\qquad\qquad \mathit{Join.unlock}\ \mathit{jdef}; \\
\qquad\qquad \mathit{Join.spawn}\ (\textbf{fun}\ () \rightarrow [\![P]\!]_{\lambda})
\end{array}
$$

where "*field i dict*" returns the ith element of the dictionary "*dict*". The join matching list now becomes the following list of triples:

$$\texttt{[}\ (110, \texttt{[}|0\ ;\ 1|\texttt{]}, g_{\{a,(b|c)\}})\ ;\ (101, \texttt{[}|0\ ;\ 2|\texttt{]}, g_{\{a,(b|c)\}})\ \texttt{]}$$

In case a join-pattern bitset is matched, the corresponding closure in the triple is called, with the join definition structure and additionally the dictionary in the triple as arguments.

Adding one dictionary component is the price we should pay to achieve complete sharing of guarded closures. However, such a dictionary is not necessary for reaction rules whose pattern is not disjunctive. In that case, the compiler can avoid the extra "*field i dict*" calls and replace them by the appropriate channel slots, which are known at compile time. However, for the sake of keeping an uniform structure of the join matching list, guarded closures should always accept the extra "*dict*" argument, even when not needed. A simple solution is to consider a dummy dictionary, to be passed to such guarded closures that do not need a dictionary.

The current implementation of JoCaml does not use dictionaries. We are still lacking experience to be able to assert whether they are worth the price or not.

9. Related work

Applied join is "impure" in the sense of Abadi and Fournet's applied π-calculus [1]. We too extend an archetypal name passing calculus with pragmatic constructs, in order to provide a full semantics that handles realistic language features without cumbersome encodings. It is worth noticing that like in [1], we distinguish between variables and names (only variables of channel type are treated as names), a distinction that is seldom made in pure calculi. Since we aim to prove a program transformation correct, we define the equivalence on open terms, those that contain free variables. Abadi and Fournet are able to require their terms to have no free variables, since their goal is to prove properties of program execution, namely the correctness of security protocols.

Our compilation scheme presented in Section 5 can be seen as the combination of two basic steps: refining channels and forwarding by dispatcher. The desired property of the forwarding behavior (Lemma 7.3) constitutes the core of the correctness proof of the compilation scheme, which essentially stems from pattern matching theory. There are other work that perform the formal treatments of forwarders, for instance [32, 19], but in different contexts. Our forwarder demultiplexes messages into separate channels according to the pattern of the messages, while [32, 19] use plain channel-to-channel linear forwarders to achieve the locality property, *i.e.* reception on a given channel takes place on an unique site. It is to be noticed that the equivalence proof of [19] is with respect to ordinary barbed congruence and by the means of a labelled transition system. Yet another example is the correctness proof of the compilation of join patterns to smooth orchestrators in [24]. The compilation of [24] is less involved than ours since it basically amounts to inserting forwarders.

We established the correctness of our compilation scheme by showing the programs before and after compilation to be behavioral congruent. It is usual practice in the literature to prove correctness of program transformations by showing semantics preservation. ([11] is a survey). Here, variations are numerous: they consist in different connections between source and target formalism (two independent languages, or with the target being a sub-set of the source), different semantics (denotational vs. operational), different equivalence relations (observational equivalence, refinement relation, simulation, etc.), and different settings (sequential, concurrent, parallel, object-oriented, etc.), Consequently, proof techniques also differ. For example, recent work of Blazy *et al.* [9] reports the formal verification of a C compiler front-end in the Coq proof assistant. It handles two independent source and target languages, both with big-step operational semantics. The major difficulty of the correctness proof resides in relating the different memory states and evaluation environments of the two languages. A simulation relation is demonstrated from target code to source code by induction on evaluation derivation and case study over the last applied evaluation rule. Closer to our work, [12] shows the correctness of an optimizing translation that compiles away pattern matching in Scala. Proof techniques analogous with ours are applied, *i.e.* they also tackle contexts explicitly by proving congruence and define observational equivalence on open terms based on the one between closed terms and closing up by substitutions. Moreover, specific to its extractor-base pattern matching, extractors are required to always terminate without exception in order to achieve the correctness.

We now review some programming languages that support concurrency and examine how our work can be related to those. Languages whose model for concurrency directly stems from the join calculus should benefit from our work. More precisely, if a language already offers à la ML pattern matching and join definitions, then its authors can implement our ideas in their framework, and their implementation effort would be small. An early example of a language based upon the join calculus is Funnel [13]. Funnel later evolved into Scala [14], where à la ML pattern matching is supported and join style concurrency is provided in terms of a library [20]. Another similar work is [42], which introduces join style concurrency in Haskell. We believe that extending the two settings above with algebraic patterns as formal arguments can be made by direct application of our techniques. Smooth orchestrators [24] differ from join definitions in rather subtle ways: an orchestrator is syntactically similar to a join definition and can be seen as defining competing reaction rules; however, (1) once a reaction rule of an orchestrator is selected and continuation fired, the whole orchestrator (together with other non-selected competing rules) gets expired and discarded; and (2) the definitions of channels and of orchestrators that synchronize them are separated. Point (2) above is quite subtle: one can orchestrate receptions on channels whose definitions are unknown, provided all the orchestrated channels are defined on the same site. Nevertheless, orchestrators are controlled by finite automata that extends the ones of [25] for join definitions. Thus, the adaptation of our techniques to orchestrators looks feasible.

In addition, there is a sustained interest in integrating join calculus into object-oriented languages : polyphonic C$^\sharp$ and its successor Cω [7] for C$^\sharp$; and JoinJava [23] for Java. Unfortunately, the issue here is the lack of pattern matching, which neither C$^\sharp$ nor Java offers. A detailed discussion on the introduction of à la ML pattern matching in object-oriented languages would be out of scope. Briefly, proposed solutions are either by the means of preprocessing [6], or by tighter language integration [14, 39]. As our compilation scheme requires precise information on pattern semantics (*e.g.* to decide the precision relation $\preceq$), we think that solutions of the second kind would facilitate the extension of the introduced pattern matching to join patterns.

Erlang [4] features both pattern matching and concurrency. However, concurrency in Erlang is based upon the actor model [21, 2]. In this model, messages are sent to *actors* and actors manage a queue of messages. Moreover, the reception behavior of an actor can be specified by the `receive` m construct. This construct is similar to ML pattern matching `match` v `with` m, except for the value matched v, which is left implicit. The semantics of `receive` m can be described as follows: attempt a match in the actor's queue, scanning it from the oldest to the most recent message, stopping when a match is found. This simple combination of message passing and pattern matching proves convenient, as witnessed by the success of Erlang. However, Erlang in general misses a simple and efficient handling of synchronization between actors as join patterns offer. Lacking necessary knowledge of Erlang internals, it is difficult for us to assess whether the selection of messages from actors queues can benefit from our techniques or not. In any case, difference in semantics is outstanding and we conjecture that an adaption of our technique would not be immediate. In particular, the existence of one message queue per receiving agent is central to Erlang model, while a join definition naturally handles several message queues.

Finally, we discuss the transplantation of our compilation scheme to a language whose semantics for concurrency is based upon the original π-calculus of [33], like for instance Pict [37], or PiDuce [10] without orchestrators. Such a task is apparently impossible. Namely, on the one hand, we propose a *compilation* scheme, and we thus need to isolate all the instances of reception on a given channel from program source ; while, on the other hand, the π-calculus features *unrestricted input capability*. More precisely, in the π-calculus, any process that knows of some channel x can input on it. As a channel name x can be passed via messages, reception on x may occur anywhere. The join calculus originates from a radical solution to the distributed implementation issue: channels and reception behaviors are defined by a synthetic construct, and input on channels cannot occur anywhere else. However, there are other solutions that retain the π-calculus as a basis while restricting input capability, such as the localized π-calculus [32]. Moreover, the located channels of Nomadic Pict [43] allows to lift such solutions to a distributed setting. Given such frameworks, we shall assume that all receptors on a given channel are known statically. Then, we can extend the input construct $x(y).P$ as $x(\pi).P$, where π is pattern, and expect to be able to translate this extended language into ordinary π-calculus. In that process, we see at least one additional complication. Let π_1 and π_2 be two patterns that are compatible (*i.e.* that have instances in common), and let us consider the following program, an analog of the simple examples of Section 4.

$$x(\pi_1).P_1 \mid x(\pi_2).P_2$$

The above process significantly differs from a join definition, since a successful input does not discard the other input. A tentative translation in the spirit of ours would be the parallel composition of a dispatcher:

$$!\ x(z).\mathtt{match}\ z\ \mathtt{with}\ \pi_1 \uparrow \pi_2 \rightarrow x_{\pi_1 \uparrow \pi_2}(z)\ \mathtt{|}\ \pi_1 \rightarrow x_{\pi_1}(z)\ \mathtt{|}\ \pi_2 \rightarrow x_{\pi_2}(z)$$

and of the following process:

$$(x_{\pi_1 \uparrow \pi_2}(z).Q_1 + x_{\pi_1}(z).Q_1) \mid (x_{\pi_1 \uparrow \pi_2}(z).Q_2 + x_{\pi_2}(z).Q_2)$$

Where Q_i is $\mathtt{match}\ z\ \mathtt{with}\ \pi_i \rightarrow P_i$, and "+" is internal choice that we use here to express input-guarded choice. Thus, we need input-guarded choice. This is a noticeable complication, even though input-guarded choice can be expressed in the π-calculus without choice [35]. Another concern is the usage of the replication operator "!" in the dispatcher. Clearly, the adaptation of our technique to a π-calculus setting is not immediate.

10. Conclusion and future work

This paper is part of our effort to develop a practical concurrent programming language with firm semantical foundations. In our opinion, a programming language is more than an accumulation of features. That is, features interact sometimes in unexpected ways, especially when intimately entwined. Here, we have studied the interaction between pattern matching and concurrency. The framework we have used was the applied join calculus — an extension of the join calculus with algebraic data types. Applied join inherits its capabilities of communication and concurrency from join and supports value passing. More significantly, it allows algebraic pattern matching in both formal arguments of channel definitions and guarded processes. Compared with join, applied join provides a more convenient (or "pragmatic"), precise and realistic language model to programmers. From that perspective, pattern matching and join calculus appear to live well together, with mutual benefits. The result of this work reinforces our interest in using à la ML pattern matching as a general purpose programming paradigm, and join calculus as the basic paradigm for concurrency.

Exploiting the fact that JoCaml already had an efficient implementation for both ML pattern matching and join primitives, we have designed the implementation of applied join as defining a practical compilation scheme that transforms extended join definitions into ordinary ones plus ML pattern matching. We have solved the non-determinism problem during the design of this compilation scheme. Moreover, we have actually integrated it into the JoCaml system with several optimizations. It is worth observing that a direct implementation of extended join-pattern matching at the runtime level would significantly complicate the management of message queues, which would then need to be scanned in search of matching messages before consuming them. As we remarked, our compilation technique may yield code of exponential size. However, we expect such blowup not to occur in practice, an expectation which is apparently confirmed by our preliminary experiments in the JoCaml system. Should this prove wrong in the future, we could face the issue in two manners : either complicate the runtime system as sketched above, or design a direct implementation of **or** in join patterns.

A theory of process equivalence has also been developed in applied join in order to assess the correctness of our compilation scheme. In archetypal name passing calculi, where every free variable is of channel type, it is sufficient to only consider terms closed in our sense, *i.e.* terms without free variables of non-channel type, when defining equivalence relations. By contrast, applied join supports real values and its static transformations should apply to open processes with free variables of non-channel type. To tackle this problem, we have first defined a weak barbed congruence to express the equivalence of two closed processes, then we have lifted the equivalence relation to open processes by closing up by all substitutions. The resulting relation is called "open equivalence". We have demonstrated it is also a full congruence and have proved our compilation scheme correct by showing that the processes before and after transformation are open equivalent. The proof technique we have used, which can be summarized as "full abstraction", stems from pattern matching theory and the fact that inserting an internal forwarding step in communications does not change process behavior.

In previous work, we have designed an object-oriented extension of the join calculus [17, 27, 29], which appeared to be more difficult. The difficulties reside in the refinement of the synchronization behavior of objects by using the inheritance paradigm. We solved the problem by designing a delicate way of rewriting join patterns at the class level. However, the introduction of algebraic patterns in join patterns impacts this class-rewriting mechanism. The interaction is not immediately clear. Up to now, we are aware of no object-oriented language where the formal arguments of methods can be patterns. We thus plan to investigate such a combination of pattern matching and inheritance, both at the calculus and language level.

Another interesting future work would be to extend our framework with more sophisticated patterns for XML data. As a matter of fact, the authors of Scala have already extended the notion of pattern matching to the processing of XML data with the help of regular expression patterns (a similar system is PiDuce [10]). Their extension makes Scala suitable for developing web service applications. Our model of pattern matching in join calculus works with general algebraic data types. At the moment, we do not see any particular barrier that prevent our model from also working with XML trees.

Acknowledgement

The authors wish to thank James Leifer and Jean-Jacques Lévy for fruitful discussions and comments. We also thank the anonymous referees for their suggestions.

References

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π–calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[4] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall, 1996.

[5] Paolo Baldan, Andrea Bracciali, and Roberto Bruni. Bisimulation by unification. In *Proceedings of the9th International Conference on Algebraic Methodology and Software Technology (AMAST'02)*, LNCS 2422, pages 254–270, 2002.

[6] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the18th Conference on Rewriting Techniques and Applications (RTA'07)*, LNCS 4533, pages 36–47, 2007.

[7] Nick Benton, Gavin Bierman, Luca Cardelli, Erik Meijer, Claudio Russo, and Wolfram Schulte. The Comega Language. `http://research.microsoft.com/comega/`, 2005.

[8] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[9] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Proceedings of the14th International Symposium on Formal Methods (FM'06)*, LNCS 4085, pages 460–475, 2006.

[10] Allen L. Brown, Cosimo Laneve, and L. Gregory Meredith. Piduce, a process calculus with native xml datatypes. In *Proceedings of the2nd International Workshop on Web Services and Formal Methods (WS-FM'05)*, LNCS 3670, pages 18–34, 2005.

[11] Maulik A. Dave. Compiler verification: A bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

[12] Burak Emir, Qin Ma, and Martin Odersky. Translation correctness for first-order object-oriented pattern matching. In *Proceedings of the5th ASIAN Symposium on Programming Languages and Systems (APLAS'07)*, 2007. to appear in LNCS.

[13] Martin Odersky *et al.*. The FUNNEL language. `http://lamp.epfl.ch/funnel/`, 2000.

[14] Martin Odersky *et al.*. The Scala Language. `http://lamp.epfl.ch/~odersky/scala/`, 2002.

[15] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, November 1998.

[16] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385, 1996.

[17] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.

[18] Cédric Fournet, Luc Maranget, Cosimo Laneve, and Didier Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of the8th International Conference on Concurrency Theory (CONCUR'97)*, LNCS 1243, pages 196–212, 1997.

[19] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. Linear forwarders. In *Proceedings of the14th International Conference on Concurrency Theory (CONCUR'03)*, LNCS 2761, pages 415–430, 2003.

[20] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. Technical Report LAMP-REPORT-2007-004, EPFL, 2007.

[21] Carl Hewitt, Peter Bishop, and RIchard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, pages 235–245, 1973.

[22] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995.

[23] G Stewart Itzstein and David Kearney. Join Java: An alternative concurrency semantics for Java. Technical Report ACRC-01-001, University of South Australia, 2001.

[24] Cosimo Laneve and Luca Padovani. Smooth orchestrators. In *Proceedings of the9th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'06)*, LNCS 3921, pages 32–46, 2006.

[25] Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. In *Proceedings of the3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, ENTCS 16(3), 1998.

[26] Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

[27] Qin Ma and Luc Maranget. Expressive synchronization types for inheritance in the join calculus. In *Proceedings of the1st Asian Symposium on Programming Languages and Systems (APLAS'03)*, LNCS 2895, pages 20–36, 2003.

[28] Qin Ma and Luc Maranget. Compiling pattern matching in join-patterns. In *Proceedings of the15th International Conference on Concurrency Theory (CONCUR'04)*, LNCS 3170, pages 417–431, 2004.
[29] Qin Ma and Luc Maranget. Information hiding, inheritance and concurrency. Rapport de recherche RR-5631, INRIA-Rocquencourt, 2005.
[30] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3), 2007.
[31] Luc Maranget, Qin Ma, and Louis Mandel. The JoCaml system. `http://jocaml.inria.fr/`, 2007.
[32] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of the25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, LNCS 1443, pages 856–867, 1998.
[33] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.
[34] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proceedings of the19th International Colloquium on Automata, Languages, and Programming (ICALP'92)*, LNCS 623, pages 685–695, 1992.
[35] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encoding. *Information and Computation*, 63(1):1–59, 2000.
[36] Martin Odersky. Functional nets. In *Proceedings of the9th European Symposium on Programming (ESOP'00)*, LNCS 1782, pages 1–25, 2000.
[37] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
[38] Paul Pritchard. On computing the subset graph of a collection of sets. *Journal of Algorithms*, 33(2):187–203, 1999.
[39] Adam Richard and Ondrej Lhoták. Oomatch: pattern matching as dispatch in java. In *Proceedings of the22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 771–772, 2007.
[40] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33(1):69–97, 1996.
[41] Davide Sangiorgi and Robin Milner. The problem of "Weak Bisimulation up to". In *Proceedings of the3rd International Conference on Concurrency Theory (CONCUR'92)*, LNCS 630, pages 32–46, 1992.
[42] Satnam Singh. Higher-order combinators for join patterns using STM. In *Proceedings of the1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, 2006.
[43] Pawel T. Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, 2000.