Compiling Pattern Matching in Join-Patterns

Qin Ma, Luc Maranget

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Compiling Pattern Matching in Join-Patterns

Qin Ma Luc Maranget

N° 5160

5th April 2004

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5160--FR+ENG

Compiling Pattern Matching in Join-Patterns

Qin Ma Luc Maranget

Thème 1 — Réseaux et systèmes
Projet Moscova

Rapport de recherche n° 5160 — 5th April 2004 — 29 pages

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.

Key-words: algebraic data type, pattern matching, functionnal programming, concurrent programming, observational equivalence, join-calculus,

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Résumé : Nous nous proposons d'étendre le join-calcul par le filtrage sur les données définies par des types algébriques. Nous poursuivons deux objectifs complémentaires : déterminer une sémantique conforme à l'intuition de la combinaison du filtrage et de la concurrence ; concevoir un schéma pratique de compilation de nos définitions étendues vers le join-calcul plus la construction de filtrage avec priorité du langage ML. Pour exprimer et prouver la correction de cette transformation, nous développons la théorie du join-calcul appliqué, un calcul de processus avec passage de valeurs et définitions par filtrage.

Mots-clés : types algériques, définitions par filtrage, programmation fonctionnelle, programmation concurrente, équivalence dans les calculs de processus, join-calcul

1 Introduction

The join-calculus [5] is a process calculus in the tradition of the π-calculus of Milner and Parrow [15]. 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, the 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 in the following sense: once a join-pattern is matched, then the corresponding guarded process may be fired.

It is tempting to extend the matching mechanism of join-patterns, so that *message* contents are also taken into account. As an example, let us consider the following list-based implementation of a concurrent stack:[1]

```
def pop(r) & State(x::xs) ▷ r(x) & State(xs)
or  push(v) & State(ls) ▷ State (x::ls)
```

The second join-pattern `push(v) & State(ls)` is an ordinary one, it is matched whenever there are messages both on `State` and `push`. By contrast, the first join-pattern is an extended one, the formal argument of channel `State` is a *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.

It should be noticed that a similar stack can be implemented in join-calculus without pattern arguments, using instead the match construct of ML:

```
def push(v) & Empty() ▷ Some (v::[])
 or push(v) & Some(ls) ▷ Some (v::ls)
 or pop(r) & Some(ls)  ▷
      match ls with | x::[] → r(x) & Empty() | x::xs → r(x) & Some(xs)
```

Obviously, the second definition requires more programming effort. Moreover, it is not immediately apparent that messages on `Some` are non-empty lists, and that the match construct thus never fails More generally, encoding the equivalent of pattern arguments in ordinary join can be quite cumbersome.

Join-definitions with (constant) pattern arguments appear informally in functional nets [17]. We here generalize this idea to full patterns. Semantics remains smooth, since both join-pattern matching and pattern matching rest upon classical substitution (or semi-unification). However, efficient implementation is more

[1] We use Objective Caml syntax for lists, with *nil* being `[]` and *cons* being infix `::`

involved. We address this issue by transforming join-definitions with pattern arguments into equivalent, ordinary, join-definitions, thereby leaving most of pattern matching compilation to an ordinary ML pattern matching compiler. Such a transformation is not straightforward. Namely, there is a gap between, non-deterministic, join-pattern matching and, deterministic, ML pattern matching. For instance, in our example of a stack, `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 ahead.

The rest of the paper is organized as follows: Section 2 gives a rapid review of pattern matching and describes key ideas of the compilation. Section 3 presents the semantics of our extended calculus as well as appropriate equivalence relations. Section 4 presents the algorithm of the transformation, which essentially works by building a meet semi-lattice of patterns, and Section 5 goes through a complete example. Finally, we deal with correctness in Section 6.

2 A journey through patterns

2.1 A rapid tour of pattern matching

Patterns and values are built the usual way as (well-sorted) terms, over constructor signatures defined by algebraic data types. In contrast to values, patterns may have variables in it, and we restrict all variables in a pattern being pairwise distinct, called linear. Moreover, patterns may contain wild-cards "_" in place of some variables when names are irrelevant, and a pattern may of course contain several wild-cards. A value v (of type t) is an instance of pattern π (of type t) when there exists a substitution σ, such that $\pi\sigma = v$. In other words, pattern π describes the prefix of instance v, and additionally binds its variables to sub-terms of v. In the following, we write $S(\pi)$ for the set of the instances of pattern π. We have the following relations among patterns (see [11]):

- Pattern π_1 and π_2 are incompatible ($\pi_1 \# \pi_2$) when $S(\pi_1) \cap S(\pi_2) = \emptyset$.
- Pattern π_1 is less precise then pattern π_2 ($\pi_1 \preceq \pi_2$) when $S(\pi_2) \subseteq S(\pi_1)$.
- Patterns π_1 and π_2 are compatible when they share at least one instance. Two compatible patterns admit a least upper bound (for $\preceq$) written $\pi_1 \uparrow \pi_2$, whose instance set is $S(\pi_1) \cap S(\pi_2)$.

- Patterns π_1 and π_2 are equivalent ($\pi_1 \equiv \pi_2$) when $S(\pi_1) = S(\pi_2)$. If so, their least upper bound is their representative, written $\pi_i \updownarrow \pi_2$.[2]

ML pattern matching is deterministic, even when patterns are overlapping (*i.e.*, compatible). More precisely, consider the following ML pattern matching

$$\mathsf{match\ e\ with} \mid \pi_1 \to \mathtt{Q}_1 \mid \pi_2 \to \mathtt{Q}_2 \mid \ldots \mid \pi_n \to \mathtt{Q}_n$$

Pattern π_i is matched by the values in set $S(\pi_i) \setminus (\cup_{1 \leq j < i} S(\pi_j))$ and only by those. In other words, given some value v, patterns π_1, π_2, ... , π_n are checked for having v as an instance, in that order, stopping as soon as a match is found. pattern matching is exhaustive when $\cup_{1 \leq i \leq n} S(\pi_j)$ is the whole set of values (of the considered type).

2.2 From patterns in join to ML pattern matching

At the time of join-pattern synchronizations, one should test messages against pattern arguments to avoid inappropriate message consumptions. Therefore, suppose we want to transform the following join-definition:

$$\begin{array}{l}\mathsf{def}\ \mathtt{c}(\pi_1)\ \&\ \mathtt{d}(\ldots) \triangleright \mathtt{P}_1 \\ \mathsf{or}\ \mathtt{c}(\pi_2)\ \&\ \mathtt{e}(\ldots) \triangleright \mathtt{P}_2\end{array}$$

Our idea is to refine channel c into more precise ones, each of which carries the instances of patterns π_1 or π_2.

$$\begin{array}{l}\mathsf{def}\ \mathtt{c}_{\pi_1}(\ldots)\ \&\ \mathtt{d}(\ldots) \triangleright \mathtt{P}_1 \\ \mathsf{or}\ \mathtt{c}_{\pi_2}(\ldots)\ \&\ \mathtt{e}(\ldots) \triangleright \mathtt{P}_2\end{array}$$

Then, we add a new reaction rule to dispatch the messages on channel c to either $\mathtt{c}_{\pi_1}$ or $\mathtt{c}_{\pi_2}$:

$$\begin{array}{l}\mathsf{or}\ \mathtt{c}(\mathtt{v}) \triangleright \mathsf{match\ v\ with} \\ \qquad \mid \pi_1 \to \mathtt{c}_{\pi_1}(\ldots) \\ \qquad \mid \pi_2 \to \mathtt{c}_{\pi_2}(\ldots) \\ \qquad \mid \_ \to \emptyset\end{array}$$

The notation $\emptyset$ stands for the null process, by which, messages that are instances of neither π_1 nor π_2 can be safely discarded.

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. However, further refinements can handle this situation.

- 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:

[2] Equivalence does not reduce to structural equality because of variables renaming or typing. For instance, we have $\_ \equiv (\_, \_)$ (here we state $\_ \updownarrow (\_, \_) = (\_, \_)$)

$$
\begin{array}{l}
\quad \mathsf{or}\ \mathsf{c}(\mathsf{v}) \triangleright \mathsf{match}\ \mathsf{v}\ \mathsf{with} \\
\qquad\qquad \mid\ \pi_2 \to \mathsf{c}_{\pi_2}(\ldots) \\
\qquad\qquad \mid\ \pi_1 \to \mathsf{c}_{\pi_1}(\ldots) \\
\qquad\qquad \mid\ \_ \to \emptyset
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{def}\ \mathsf{c}_{\pi_1}(\ldots)\ \&\ \mathsf{d}(\ldots) \triangleright \mathsf{P}_1 \\
\quad \mathsf{or}\ \mathsf{c}_{\pi_2}(\ldots)\ \&\ \mathsf{d}(\ldots) \triangleright \mathsf{P}_1 \\
\quad \mathsf{or}\ \mathsf{c}_{\pi_2}(\ldots)\ \&\ \mathsf{e}(\ldots) \triangleright \mathsf{P}_2
\end{array}
$$

Observe that non-determinism is now more explicit: an instance of π_2 sent on channel c can be consumed by either reaction rule. We can shorten the new definition a little by using or in join-patterns:

$$
\begin{array}{l}
\mathsf{def}\ (\mathsf{c}_{\pi_1}(\ldots)\ \mathsf{or}\ \mathsf{c}_{\pi_2}(\ldots))\ \&\ \mathsf{d}(\ldots) \triangleright \mathsf{P}_1 \\
\quad \mathsf{or}\ \mathsf{c}_{\pi_2}(\ldots)\ \&\ \mathsf{e}(\ldots) \triangleright \mathsf{P}_2
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{def}\ \mathsf{c}_{\pi_1 \updownarrow \pi_2}(\ldots)\ \&\ \mathsf{d}(\ldots) \triangleright \mathsf{P}_1 \\
\quad \mathsf{or}\ \mathsf{c}_{\pi_1 \updownarrow \pi_2}(\ldots)\ \&\ \mathsf{e}(\ldots) \triangleright \mathsf{P}_2 \\
\quad \mathsf{or}\ \mathsf{c}(\mathsf{v}) \triangleright \mathsf{match}\ \mathsf{v}\ \mathsf{with} \\
\qquad\qquad \mid\ \pi_1 \updownarrow \pi_2 \to\ \ \mathsf{c}_{\pi_1 \updownarrow \pi_2}(\ldots) \\
\qquad\qquad \mid\ \_ \to \emptyset
\end{array}
$$

- 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:

$$
\begin{array}{l}
\mathsf{def}\ (\mathsf{c}_{\pi_1}(\ldots)\ \mathsf{or}\ \mathsf{c}_{\pi_1 \uparrow \pi_2}(\ldots))\ \&\ \mathsf{d}(\ldots) \triangleright \mathsf{P}_1 \\
\quad \mathsf{or}\ (\mathsf{c}_{\pi_2}(\ldots)\ \mathsf{or}\ \mathsf{c}_{\pi_1 \uparrow \pi_2}(\ldots))\ \&\ \mathsf{e}(\ldots) \triangleright \mathsf{P}_2 \\
\quad \mathsf{or}\ \mathsf{c}(\mathsf{v}) \triangleright \mathsf{match}\ \mathsf{v}\ \mathsf{with} \\
\qquad\qquad \mid\ \pi_1 \uparrow \pi_2 \to \mathsf{c}_{\pi_1 \uparrow \pi_2}(\ldots) \\
\qquad\qquad \mid\ \pi_1 \to \mathsf{c}_{\pi_1}(\ldots) \\
\qquad\qquad \mid\ \pi_2 \to \mathsf{c}_{\pi_2}(\ldots) \\
\qquad\qquad \mid\ \_ \to \emptyset
\end{array}
$$

  Notice 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 $\mathsf{c}_\pi(\ldots)$. We now demonstrate variable management by means of our

stack example. Here, the relevant patterns are $\pi_1 = \ell$ and $\pi_2 =$ `x::xs` and we are in the case where $\pi_1 \preceq \pi_2$ (and $\pi_2 \not\preceq \pi_1$ because of instance [ ]). Our idea is to let dispatching focus on instance checking, and to perform variable binding after synchronization:

```
def push(v) & (State_::_(z) or State_(z)) ▷ match z with ℓ →  State(v::ℓ)
 or pop(r)  & State_::_(z) ▷ match z with x::xs → r(x) & State(xs)
 or State(v) ▷ match v with
               | _::_ → State_::_(v)
               | _ → State_(v)
```

One may believe that the matching of the pattern `x::xs` needs to be performed twice, but it is not necessary. The compiler in fact knows that the matching of `z` against `x::xs` (on second line) cannot fail. As a consequence, no test needs to be performed here, only the binding of the pattern variables. Moreover, the existing optimizing pattern matching compiler of [11] can be fooled into producing minimal code for such a situation by simply asserting that the compiled matching is exhaustive.

3 The applied join-calculus

To express pattern matching in both the join-patterns and the guarded processes, we introduce and study an extension of the join-calculus which we called the applied join-calculus, by analogy with "the applied π-calculus" [1]. The applied join-calculus builds upon the pure join-calculus [5], and extends it with constructors, value-passing and pattern matching constructs.

3.1 Syntax and scopes

The syntax of the applied join-calculus is given in Figure 1. Constructors of algebraic data types have an arity and are ranged over by C. A constructor with arity 0 is a constant. We assume an infinite set of variables, ranged over by $a, b, \ldots, y, z$.

Two new syntactic categories are introduced: expressions and patterns. At the first glance, both expressions and patterns are terms constructed from variables and constructors, where n matches the arity of constructor C. However, patterns may have occasional wild-cards, and we require patterns to be linear. ML pattern matching is added as a process, which matches the value of the expression against a list of patterns. Moreover, in contrast to ordinary name-passing join-calculus, there are two more radical extensions: first, message contents become expressions, that

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

Figure 1: Syntax of the applied join-calculus

is, we have value-passing; second, when a channel name is defined in a join-pattern, we also specify what pattern the message content should satisfy.

There are two kinds of bindings: the definition process $\mathsf{def}\ D\ \mathsf{in}\ P$ binds all the channel names defined in D ($\mathsf{dn}[D]$) in the scope of P; while the reaction rule $J \triangleright P$ or the ML pattern matching $\mathsf{match}\ e\ \mathsf{with}\ \mid \pi_1 \to P_1 \mid \ldots \mid \pi_m \to P_m$ bind all the local variables ($\mathsf{rv}[J]$ or $\mathsf{rv}[\pi_i]$) in the scope of P or P_i, $i \in \{1, \ldots, m\}$.

The definitions of the set of defined channel names $\mathsf{dn}[\cdot]$, the set of local variables $\mathsf{rv}[\cdot]$, and the set of free variables $\mathsf{fv}[\cdot]$ are almost the same as in the join-calculus,

except for the following modifications or extensions to adopt patterns.

$$\begin{array}{rcl} \mathsf{rv}[c(\pi)] & \stackrel{\text{def}}{=} & \mathsf{rv}[\pi] \\ \\ \mathsf{rv}[\_] & \stackrel{\text{def}}{=} & \emptyset \\ \mathsf{rv}[x] & \stackrel{\text{def}}{=} & \{x\} \\ \mathsf{rv}[C(\pi_1, \pi_2, \ldots, \pi_n)] & \stackrel{\text{def}}{=} & \mathsf{rv}[\pi_1] \uplus \mathsf{rv}[\pi_2] \uplus \ldots \uplus \mathsf{rv}[\pi_n] \end{array}$$

$$\begin{array}{l} \mathsf{fv}[\mathsf{match}\ u\ \mathsf{with}\ \ |^{i\in I}\ \pi_i \to P_i] \\ \qquad\qquad \stackrel{\text{def}}{=}\ \mathsf{fv}[u] \cup (\bigcup_{i\in I} \mathsf{fv}[P_i] \setminus \mathsf{fv}[\pi_i]) \end{array}$$

We assume a type discipline in the style of the type system of the join-calculus [7], extended with constructor 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 the arity checking of polyadic join-calculus is replaced by a well-typing assumption in our calculus, which is monadic and whose message contents can be tuples. However, 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. Generally speaking, in name-passing calculi semantics, the latter kind of variables are (almost) treated as channel names, that is, values. While, in any reasonable semantics, the former kind of variables cannot be treated so. Reduction will operate on *variable-closed* (*closed* for short) terms, whose free variables are all of channel type.

Finally, we use the or construct in join-patterns as syntax sugar, in the following sense:

$$J \,\&\, (J_1 \,\mathsf{or}\, J_2) \triangleright P = (J \,\&\, J_1 \triangleright P) \,\mathsf{or}\, (J \,\&\, J_2 \triangleright P)$$

3.2 Reduction semantics

We establish the semantics in the reflexive chemical abstract machine style [5, 3]. A *chemical solution* is a pair $\mathcal{D} \vdash \mathcal{P}$, where $\mathcal{D}$ is a multiset of join-definitions, and $\mathcal{P}$ is a multiset of processes. Extending the notion of closeness to solutions, a solution is closed when all the join-definitions and processes in it are closed. The chemical rewrite rules are given in Figure 2. They apply to closed solutions, and consist of two kinds: structural rules $\rightharpoonup$ or $\rightharpoondown$ represent the syntactical rearrangement of the terms,

$$
\begin{array}{lrcl}
\textsc{Str-Null} & \vdash \emptyset & \rightleftharpoons & \vdash \\
\textsc{Str-Par} & \vdash P \,\&\, P' & \rightleftharpoons & \vdash P, P' \\
\textsc{Str-Top} & \top \vdash & \rightleftharpoons & \vdash \\
\textsc{Str-And} & D \,\mathsf{or}\, D' \vdash & \rightleftharpoons & D, D' \vdash \\
\textsc{Str-Def} & \vdash \mathsf{def}\ D\ \mathsf{in}\ P & \rightleftharpoons & D \vdash P \\
\textsc{React} & J \triangleright P \vdash J\sigma & \longrightarrow & J \triangleright P \vdash P\sigma \\
\textsc{Match} & \vdash \mathsf{match}\ \pi_i\rho\ \mathsf{with}\ \mid \pi_1 \to P_1 \mid \ldots \mid \pi_m \to P_m & & \\
 & & \longrightarrow & \vdash P_i\rho
\end{array}
$$

Side conditions:

STR-DEF	$\mathsf{dn}[D]$ is fresh
REACT	σ substitutes expressions for $\mathsf{rv}[J]$
MATCH	ρ substitutes expressions for $\mathsf{rv}[\pi_i]$ and $\forall j < i, \pi_j \not\leq \pi_i\rho$

Figure 2: RCHAM of the applied join-calculus

and reduction rules $\longrightarrow$ represent the computation steps. We follow the convention to omit the part of the solution which remains unchanged during rewrite.

Matching of message contents by formal 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.

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 1. *Denote $\rightleftharpoons^*$ as the transitive closure of $\rightharpoonup \cup \rightharpoondown$,*

1. *$P \equiv Q$ iff $\vdash P \rightleftharpoons^* \vdash Q$*
2. *$P \longrightarrow Q$ iff $\vdash P \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \vdash Q$*

Obviously, we have the structural rule, namely, if $P \longrightarrow Q$, $P \equiv P'$, and $Q \equiv Q'$, then $P' \longrightarrow Q'$.

3.3 Equivalence relation

In this section, we equip the applied join-calculus with equivalence relations to allow equational reasoning among 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 [16], and adapted to many other process caluli [9, 2], including the join-calculus [5]. We take *weak barbed congruence* [16] as our basic notion of "behavioral equivalence" for closed processes.

Definition 2 (Barb predicates). *Let P be a closed process, and c be a channel name*

1. *P has a strong barb on c: $P\downarrow_c$, iff $P \equiv (\mathsf{def}\ D\ \mathsf{in}\ Q)\ \&\ c(e)$, for some D, Q and e.*
2. *P has a weak barb on channel c: $P\Downarrow_c$, iff $P \longrightarrow^* P'$ such that $P'\downarrow_c$.*

Definition 3 (Weak barbed bisimulation). *A binary relation $\mathcal{R}$ on closed processes is a weak barbed bisimulation if, whenever $P\mathcal{R}Q$, we have*

1. *If $P \longrightarrow P'$, then $\exists Q'$, s.t. $Q \longrightarrow^* Q'$ and $P'\mathcal{R}Q'$. Vice versa.*
2. *For any c, $P\Downarrow_c$ iff $Q\Downarrow_c$.*

By definition, Weak barbed bisimilarity ($\dot{\approx}$) is the largest weak barbed bisimulation.

A *context* $C[\cdot]$ is a term built by the grammar of process with a single process placeholder $[\cdot]$. An *executive contexts* $E[\cdot]$ is a context in which the placeholder is not guarded. Namely:

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

We say a context is closed if all the free variables in it are of channel type.

Definition 4 (Weak barbed congruence). *A binary relation on closed processes is a weak barbed congruence if it is a weak barbed bisimulation and closed by application of any closed executive 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 executive contexts, it can be proved that the full congruence does not provide more discrimination power. Similarily to what Fournet has established for the pure join-calculus in his thesis [4], we first have the property that $\approx$ is a closure under substitution.

Lemma 1. *Given two closed processes P and Q, if $P \approx Q$, then for any substitution σ, $P\sigma \approx Q\sigma$.*

Then based on this property, the full congruence is also guaranteed.

Theorem 1. *Weak barbed congurence $\approx$ is a closure of any closed contexts.*

As for the pure join-calculus, Lemma 1 can be proved by constructing an executive context to mimic the behavior of any subustution. And Theorem 1 holds basically following the fact that the essence of a guarded context is substitution.

Up to now, we define the weak barbed congruence as expressing the equivalence of two processes at runtime. However, this is not sufficient for reasoning the behavior of our compilation, which applies perfectly well to processes with free variables. In other words, we also need a way to express the equivalence of two processes statically. Of course, the static equivalence must imply the runtime one. Therefore, the equivalence relation of any processes, whether closed or not, is defined in terms of the runtime equivalence relation $\approx$, using substitutions to close up.

Definition 5 (Static equivalence). *Two processes P and Q are statically equivalent ($\doteq$), if for any substitution σ such that $P\sigma$ and $Q\sigma$ are closed, $P\sigma \approx Q\sigma$.*

Following the definition, we can check that $\doteq$ is closed by substitution.

Lemma 2. $P \doteq Q \implies \forall \sigma. P\sigma \doteq Q\sigma$

Proof. This is equivalent to prove

$$P \doteq Q \implies \forall \sigma_1, \sigma_2.\ P\sigma_1\sigma_2 \approx Q\sigma_1\sigma_2 \qquad (*)$$

where σ_1 is the substitution of this lemma, and σ_2 is the substitution in the static equivalence definition that closes $P\sigma_1$ and $Q\sigma_1$. Taking σ as $\sigma_1 \circ \sigma_2$, statement $(*)$ holds by definition. □

More importantly, $\doteq$ is also closed by any contexts (see Theorem 2 below). To prove the theorem, we need the following lemma.

Lemma 3. *For any closed process P, if P reduces and only reduces to P', then $P \approx P'$.*

Proof. Let $\mathcal{R} = \{(\mathsf{def}\ D\ \mathsf{in}\ P \,\&\, Q,\ \mathsf{def}\ D\ \mathsf{in}\ P' \,\&\, Q)\}$ for any closed D and Q. It is easy to check that $(\equiv \mathcal{R} \equiv)$ is a weak barbed congruence, because: if $P \equiv P_0 \,\&\, J\sigma$, then $P' \equiv P_0' \,\&\, J\sigma$ where P_0 reduces and only reduces to P_0'; and we have $\longrightarrow$ preserves barbs. Furthermore, we have $P \equiv (\mathsf{def}\ \top\ \mathsf{in}\ P \,\&\, \emptyset)\mathcal{R}(\mathsf{def}\ \top\ \mathsf{in}\ P' \,\&\, \emptyset) \equiv P'$, *i.e.* $P \equiv \mathcal{R} \equiv P'$. □

Theorem 2. *The static equivalence* $\fallingdotseq$ *is a full congruence.*

Proof. We demonstrate $\fallingdotseq$ is closed by executive contexts, definition contexts, and pattern matching contexts respectively.

1. Closed by executive contexts, namely

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

For any substitution σ such that $(E[P])\sigma$ and $(E[Q])\sigma$ closed, we should have $(E[P])\sigma \approx (E[P])\sigma$. We write $(E[P])\sigma$ as $E\sigma[P\sigma_1]$ and $(E[Q])\sigma$ as $E\sigma[Q\sigma_1]$, where $E\sigma[\cdot]$, $P\sigma_1$, $Q\sigma_1$ closed and σ_1 is σ minus the bindings for the channel names bound by E in $[\cdot]$. By hypothesis $P \fallingdotseq Q$, we have $P\sigma_1 \approx Q\sigma_1$, which is closed by closed executive context $E\sigma[\cdot]$, hence we conclude.

2. Closed by definition contexts, namely

$$P \fallingdotseq Q \implies \mathsf{def}\ J \triangleright P\ \mathsf{or}\ D\ \mathsf{in}\ R \fallingdotseq \mathsf{def}\ J \triangleright Q\ \mathsf{or}\ D\ \mathsf{in}\ R$$

For any substitution σ, such that $(\mathsf{def}\ J \triangleright P\ \mathsf{or}\ D\ \mathsf{in}\ R)\sigma$, $(\mathsf{def}\ J \triangleright Q\ \mathsf{or}\ D\ \mathsf{in}\ R)\sigma$ closed, we should prove

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

namely

$$\mathsf{def}\ J \triangleright P\sigma_1\ \mathsf{or}\ D\sigma\ \mathsf{in}\ R\sigma \approx \mathsf{def}\ J \triangleright Q\sigma_1\ \mathsf{or}\ D\sigma\ \mathsf{in}\ R\sigma \qquad (\sharp)$$

where $D\sigma$ and $R\sigma$ closed and $\sigma_1 = \sigma \setminus \mathsf{rv}[J]$. By hypothesis $P \fallingdotseq Q$ and Lemma 2, we have $P\sigma_1 \fallingdotseq Q\sigma_1$.

We build the following relation $\mathcal{R}$ on closed processes

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

and we analyze the following three aspects of $\mathcal{R}$: closure of closed executive contexts; refinement of barbs; and reduction bisimulation.

($\star$) $\mathcal{R}$ is a closure of closed executive context up to $\equiv$.

For any closed $E[\cdot]$, with necessary α-conversion, we have

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

where $A \,\&\, K \approx B \,\&\, K$, because $\approx$ is closed by the closed executive context $[\cdot] \,\&\, K$.

For the rest of our analysis, we write $\mathcal{C}[X, Y]$ for the closed process $\mathsf{def}\ J \triangleright X\ \mathsf{or}\ D\ \mathsf{in}\ Y$. Moreover, because the symmetry of relation $\mathcal{R}$, we only dicuss from left to right. The analysis for the opposite direction is similar.

($\star$) $\mathcal{R}$ refines barbs, that is $\mathcal{C}[S, A]\Downarrow_c \implies \mathcal{C}[T, B]\Downarrow_c$.

Obviously, $\mathcal{C}[T, \cdot]$ is a closed executive context, and by hypothesis $A \approx B$, we have

$$\mathcal{C}[T, A] \approx \mathcal{C}[T, B] \tag{1}$$

Now we prove

$$\text{For any closed process } Z,\ \mathcal{C}[S, Z]\Downarrow_c \implies \mathcal{C}[T, Z]\Downarrow_c \tag{2}$$

We induce on the length of reduction steps of $\mathcal{C}[S, Z]$ before it reaches the strong barb $\downarrow_c$.

Base case: $n = 0$, we have $\mathcal{C}[S, Z]\downarrow_c$, thus, $Z \equiv Z_0 \,\&\, c(e)$, hence $\mathcal{C}[T, Z]\downarrow_c$.

Step case: We have $\mathcal{C}[S, Z] \longrightarrow W \longrightarrow^n \downarrow_c$, and we distinguish the ways in which the first reduction can occur.

- $Z \longrightarrow Z'$ and $W = \mathcal{C}[S, Z']$. Then $\mathcal{C}[T, Z] \longrightarrow \mathcal{C}[T, Z']$. By hypothesis we have $\mathcal{C}[S, Z'] \longrightarrow^n \downarrow_c$, therefore $\mathcal{C}[T, Z']\Downarrow_c$ by induction hypothesis, so $\mathcal{C}[T, Z]\Downarrow_c$.
- $Z \equiv Z_0 \,\&\, J\rho$ and $W = \mathcal{C}[S, Z_0 \,\&\, S\rho]$. Then $\mathcal{C}[T, Z] \longrightarrow \mathcal{C}[T, Z_0 \,\&\, T\rho]$. By hypothesis we have $\mathcal{C}[S, Z_0 \,\&\, S\rho] \longrightarrow^n \downarrow_c$, thus $\mathcal{C}[T, Z_0 \,\&\, S\rho]\Downarrow_c$ by induction hypothesis with Z as $Z_0 \,\&\, S\rho$. Moreover, since $S \doteqdot T$ and $S\rho$, $T\rho$ closed, we have $S\rho \approx T\rho$, so that $Z_0 \,\&\, S\rho \approx Z_0 \,\&\, T\rho$, so that $\mathcal{C}[T, Z_0 \,\&\, S\rho] \approx \mathcal{C}[T, Z_0 \,\&\, T\rho]$ by (1). That is we have $\mathcal{C}[T, Z_0 \,\&\, T\rho]\Downarrow_c$, hence $\mathcal{C}[T, Z]\Downarrow_c$.
- $Z \equiv Z_0 \,\&\, J_i\rho_i$, D is of the form $\ldots\ \mathsf{or}\ J_i \triangleright P_i\ \mathsf{or} \ldots$, and $W = \mathcal{C}[S, Z_0 \,\&\, P_i\rho_i]$. Then $\mathcal{C}[T, Z] \longrightarrow \mathcal{C}[T, Z_0 \,\&\, P_i\rho_i]$. By hypothesis we have $\mathcal{C}[S, Z_0 \,\&\, P_i\rho_i] \longrightarrow^n \downarrow_c$, therefore we have $\mathcal{C}[T, Z_0 \,\&\, P_i\rho_i]\Downarrow_c$ by induction hypothesis with Z as $Z_0 \,\&\, P_i\rho_i$, hence $\mathcal{C}[T, Z]\Downarrow_c$.

By (2), (1) and $\approx$ refining barbs, we get

$$\mathcal{C}[S, A]\Downarrow_c \implies \mathcal{C}[T, A]\Downarrow_c \implies \mathcal{C}[T, B]\Downarrow_c$$

($\star$) $\mathcal{R}$ is a reduction-based bisimulation up to $\approx$ on the right.

We first prove the following statement

$$\text{If } \mathcal{C}[S, A] \longrightarrow M, \text{ then } \mathcal{C}[T, A] \longrightarrow N, \text{ and } M\mathcal{R}N. \tag{3}$$

We distinguish the three cases of M.

- $A \longrightarrow A'$ and $M = \mathcal{C}[S, A']$. Then $\mathcal{C}[T, A] \longrightarrow \mathcal{C}[T, A']$, and $\mathcal{C}[S, A']$, $\mathcal{C}[T, A']$ satisfy relation $\mathcal{R}$.
- $A \equiv A_0 \,\&\, J\rho$ and $M = \mathcal{C}[S, A_0 \,\&\, S\rho]$. Then $\mathcal{C}[T, A] \longrightarrow \mathcal{C}[T, A_0 \,\&\, T\rho]$, where $S\rho$, $T\rho$ closed. Because $S \fallingdotseq T$, we have $S\rho \approx T\rho$, and as a consequence, $A_0 \,\&\, S\rho \approx A_0 \,\&\, T\rho$. That is $\mathcal{C}[S, A_0 \,\&\, S\rho]$ and $\mathcal{C}[T, A_0 \,\&\, T\rho]$ satisfy relation $\mathcal{R}$.
- $A \equiv A_0 \,\&\, J_i\rho_i$, D has form $\ldots$ or $J_i \triangleright P_i$ or $\ldots$, and $M = \mathcal{C}[S, A_0 \,\&\, P_i\rho_i]$. Then $\mathcal{C}[T, A] \longrightarrow \mathcal{C}[T, A_0 \,\&\, P_i\rho_i]$, and $\mathcal{C}[S, A_0 \,\&\, P_i\rho_i]$, $\mathcal{C}[T, A_0 \,\&\, P_i\rho_i]$ satisfy relation $\mathcal{R}$.

Moreover, because of statement (1), we have

$$\text{If } \mathcal{C}[T, A] \longrightarrow N, \text{ then } \exists N' \text{ s.t. } \mathcal{C}[T, B] \longrightarrow^* N', \text{ and } N \approx N'. \tag{4}$$

Statement (3) together with statement (4) justify

$$\text{If } \mathcal{C}[S, A] \longrightarrow M, \text{ then } \exists N' \text{ s.t. } \mathcal{C}[T, B] \longrightarrow^* N', \text{ and } M\mathcal{R} \approx N'. \tag{5}$$

Let $\overline{\mathcal{R}}$ be $\equiv\mathcal{R}\equiv\approx$. Following the analysis of $\mathcal{R}$ above, one easily checks that $\overline{\mathcal{R}}$ is a weak barbed congruence. Obviously, the two processes of statement ($\sharp$) satisfy relation $\mathcal{R}$, hence $\overline{\mathcal{R}}$, therefore ($\sharp$) holds, namely, $\fallingdotseq$ is closed by any definition contexts.

3. Closed by pattern matching contexts, namely

$$P \fallingdotseq Q \implies \mathsf{match}\ e\ \mathsf{with}\ \ldots \pi_k \to P \ldots \fallingdotseq \mathsf{match}\ e\ \mathsf{with}\ \ldots \pi_k \to Q \ldots$$

For any substitution σ, such that we have $(\mathsf{match}\ e\ \mathsf{with}\ \ldots \pi_k \to P \ldots)\sigma$ and $(\mathsf{match}\ e\ \mathsf{with}\ \ldots \pi_k \to Q \ldots)\sigma$ closed, we should prove

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

namely

$$\mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to P\sigma' \ldots \approx \mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to Q\sigma' \ldots \qquad (\natural)$$

where $e\sigma$ closed and $\sigma' = \sigma \setminus \mathsf{rv}[\pi_k]$.

Depending on the value of $e\sigma$, $\mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to P\sigma' \ldots$ reduces and only reduces to either $P(\sigma' \circ \delta_k)$ or $R_i\delta_i$, where $P(\sigma' \circ \delta_k)$ closed, and R_i is the ith guarded process in this pattern matching. And we have the similar statement for $\mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to Q\sigma' \ldots$. Therefore, by Lemma 3, we have

$$\mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to P\sigma' \ldots \quad \approx \quad P(\sigma' \circ \delta_k) \qquad (6)$$
$$\mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to Q\sigma' \ldots \quad \approx \quad Q(\sigma' \circ \delta_k) \qquad (7)$$

or

$$\mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to P\sigma' \ldots \quad \approx \quad R_i\delta_i \qquad (8)$$
$$\mathsf{match}\ e\sigma\ \mathsf{with}\ \ldots\pi_k \to Q\sigma' \ldots \quad \approx \quad R_i\delta_i \qquad (9)$$

Because $P \fallingdotseq Q$, according to Lemma 2, we have $P(\sigma' \circ \delta_k) \approx Q(\sigma' \circ \delta_k)$. Then following the transitivity of $\approx$, either by (6) and (7), or (8) and (9), we have the statement $(\natural)$ holds.

□

There is still a good property worth noticing: in fact, for the closed subset of the applied join-calculus, we have $\fallingdotseq$ and $\approx$ coincide. This is almost straightforward following the definition of static equivalence and Lemma 1.

4 The compilation $[\![P]\!]$

We formalize the intuitive idea described in Section 2 as a transformer Y_c, which transforms a join-definition *w.r.t.* channel c. The algorithm essentially works by constructing the meet semi-lattice of the formal pattern arguments of channel c in D, modulo pattern equivalence $\equiv$, and with relation $\preceq$ as partial order. Moreover, we visualize the lattice as a Directed Acyclic Graph, 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 [19].

Algorithm Y_c**:** Given D, the join-definition to be transformed.

Step 0: Pre-process

1. Collect all the pattern arguments of channel c into the sequence: $S = \pi_1; \pi_2; \ldots; \pi_n$.
2. Compute the equivalence classes of S, *w.r.t.* $\equiv$. Record the representative patterns of those into another sequence S'.
3. Perform exhaustiveness check on S', if not exhaustive, issue an warning.
4. **IF** There is only one pattern in S', and that S' is exhaustive
   THEN goto Step 5 (In that case, no dispatching is needed.)

Step 1: Closure by upper bound

For any pattern γ and pattern sequence $X = \gamma_1; \gamma_2; \ldots; \gamma_n$, we define $\gamma \uparrow X$ as the sequence $\gamma \uparrow \gamma_{i_1}; \gamma \uparrow \gamma_{i_2}; \ldots; \gamma \uparrow \gamma_{i_m}$, where the γ_{i_k}'s are the patterns from X that are compatible with γ.

We also define function F, which takes a pattern sequence X as argument and returns a pattern sequence.

IF X is empty

THEN $F(X) = X$

ELSE Decompose X as $\gamma; X'$ and state $F(X) = \gamma; F(X'); \gamma \uparrow F(X')$

Compute the sequence $U = F(S')$. It is worth noticing that U is the sequence of valid patterns $(\pi'_{i_1} \uparrow \ldots (\pi'_{i_{k-1}} \uparrow \pi'_{i_k}) \ldots)$, with $1 \leq i_1 < i_2 < \ldots < i_k \leq n'$, and $1 \leq k \leq n'$, where we decompose S' as $\pi'_1; \pi'_2; \ldots; \pi'_{n'}$.

Step 2: Up to equivalence

Compute the equivalence classes of U *w.r.t.* relation $\equiv$. This yields a sequence of representatives U'.

Step 3: Build DAG

Build a directed acyclic graph $G(V, E)$ corresponding to the semi-lattice $(U', \preceq)$.

1. $V = \emptyset, E = \emptyset$.
2. For each pattern γ in U', add a new vertex v into V and labeled the vertex with γ, written as $\mathsf{label}(v) = \gamma$.
3. $\forall (v, v') \in V \times V, v \neq v'$, if $\mathsf{label}(v) \preceq \mathsf{label}(v')$, then add an edge from v' to v into E.

Step 4: Add dispatcher

Assume following one topological order, the vertices of G are indexed as $v_1, \ldots, v_m$. We extend the join-definition D with a dispatcher on channel c of the form: $c(x) \triangleright \mathsf{match}\ x\ \mathsf{with}\ \mathcal{L}$, where x is a fresh variable and $\mathcal{L}$ is built as follows:

1. Let j ranges over $\{1, \ldots, m\}$. Following the above topological order, for all vertices v_j in V append a rule "$\mid \mathsf{label}(v_j) \rightarrow c_j(x)$" to $\mathcal{L}$, where c_j is a fresh channel name.
2. If S' is not exhaustive, then add a rule "$\mid \_ \rightarrow \emptyset$" at the end.

Step 5: Rewrite reaction rules

For each reaction rule defining channel c in D: $J_i \,\&\, c(\pi_i) \triangleright Q_i$, we rewrite it according to the following policy. Let $Q'_i = \mathsf{match}\ x_i\ \mathsf{with}\ \pi_i \rightarrow Q_i$, where x_i is a fresh variable.

IF coming from Step 0

THEN rewrite to $J_i \,\&\, c(x_i) \triangleright Q'_i$

ELSE

1. Let v_{j_i} be the unique vertex in V, *s.t.* $\mathsf{label}(v_{j_i}) \equiv \pi_i$.
2. We collect all the predecessors of v_{j_i} in G, and we record the indexes of them, together with j_i, into a set notated as $I(\pi_i)$.
3. Rewrite to $J_i \,\&(\bigvee_{j \in I(\pi_i)} c_j(x_i)) \triangleright Q'_i$, where $\bigvee$ is the generalized or construct of join-patterns.

To transform a join-definitions D, assuming $\mathsf{dn}[D] = \{a, b, \ldots, c\}$, we just do $Y_a Y_b \ldots Y_c(D)$. And the compilation of process $[\![P]\!]$ is then defined as follows.

$$
\begin{array}{rcl}
[\![\emptyset]\!] & \stackrel{\text{def}}{=} & \emptyset \\
[\![x(e)]\!] & \stackrel{\text{def}}{=} & x(e) \\
[\![P \,\&\, P]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \,\&\, [\![P]\!] \\
[\![\mathsf{def}\ D\ \mathsf{in}\ P]\!] & \stackrel{\text{def}}{=} & \mathsf{def}\ Y_a Y_b \ldots Y_c(D)\ \mathsf{in} [\![P]\!] \\
[\![\mathsf{match}\ e\ \mathsf{with}\ \mid^{i \in I} \pi_i \rightarrow P_i]\!] & \stackrel{\text{def}}{=} & \mathsf{match}\ e\ \mathsf{with}\ \mid^{i \in I} \pi_i \rightarrow [\![P_i]\!]
\end{array}
$$

Observe that the compilation preserves the interface of join-definitions. Namely, it only affects definitions D, while messages sendings remain the same.

5 Example of compilation

Given the following join-definition of an enriched integer stack

```
def push(v) & State(ls) ▷ State (v::ls)
 or pop(r) & State(x::xs) ▷ r(x) & State(xs)
 or insert(n) & State(0::xs) ▷ State(0::n::xs)
 or last(r) & State([x]) ▷ r(x) & State([x])
 or swap() & State(x1::x2::xs) ▷ State(x2::x1::xs)
 or pause(r) & State([]) ▷ r()
 or resume(r) ▷ State([]) & r()
```

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 *w.r.t.* channel `State`.

Step 0 We collect the pattern arguments of channel `State` into S

$$S = \texttt{ls};\ \texttt{x::xs};\ \texttt{0::xs};\ \texttt{x::[]};\ \texttt{x}_1\texttt{::x}_2\texttt{::xs};\ \texttt{[]}$$

Because none of these patterns is equivalent to another, $S' = S$. Additionally, S' is exhaustive (pattern `ls` alone covers all possibilities).

Step 1,2 We compute all possible least upper bounds among several patterns from S' and record the representatives of the resulting equivalence classes into U'.

$$U' = \texttt{ls};\ \texttt{x::xs};\ \texttt{0::xs};\ \texttt{x::[]};\ \texttt{x}_1\texttt{::x}_2\texttt{::xs};\ \texttt{[]};\ \texttt{0::x}_2\texttt{::xs};\ \texttt{0::[]}$$

Notice that the last two patterns are new, where

$$\begin{aligned} \texttt{0::x}_2\texttt{::xs} &= \texttt{0::xs} \uparrow \texttt{x}_1\texttt{::x}_2\texttt{::xs} \\ \texttt{0::[]} &= \texttt{0::xs} \uparrow \texttt{x::[]} \end{aligned}$$

Step 3 We build the semi-lattice $(U', \preceq)$, see Figure 3.

Step 4 One possible topological order of the vertices is also shown at the right of Figure 3. Following that order, we build the dispatcher on channel `State`.

```
or State(y) ▷ match y with
              | 0::x2::xs → State1(y)
              | 0::[] → State2(y)
              | x1::x2::xs → State3(y)
              | 0::xs → State4(y)
              | x::[] → State5(y)
              | x::xs → State6(y)
              | [] → State7(y)
              | ls → State8(y)
```

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

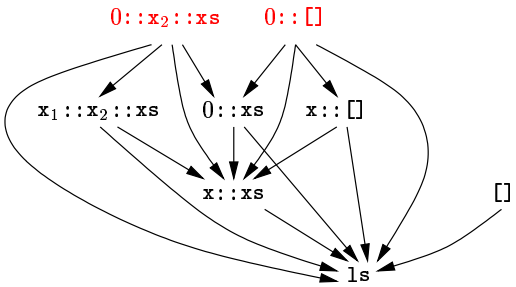
```
insert(n) & (State1(x3) or State2(x3) or State4(x3))
          ▷ match x3 with 0::xs → State(0::n::xs)
```

where $\texttt{State}_1$, $\texttt{State}_2$ and $\texttt{State}_4$ are the fresh channel names corresponding to vertices 1, 2, 4 respectively, and $\texttt{x}_3$ 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.

```
def push(v) & (State1(x1) or ... or State8(x1))
            ▷ match x1 with ls → State (v::ls)
 or pop(r) & (State1(x2) or ... or State6(x2))
            ▷ match x2 with x::xs → r(x) & State(xs)
 or insert(n) & (State1(x3) or State2(x3) or State4(x3))
            ▷ match x3 with 0::xs → State(0::n::xs)
 or last(r) & (State2(x4) or State5(x4))
            ▷ match x4 with [x] → r(x) & State([x])
 or swap() & (State1(x5) or State3(x5))
            ▷ match x5 with x1::x2::xs → State(x2::x1::xs)
 or pause(r) & State7(x6) ▷ r()
 or resume(r) ▷ State([]) & r()
```

As discussed at the end of Section 2, ML pattern matchings in the guarded processes are here only for binding pattern variables. Therefore, if the original pattern does not contain any variables (*c.f.* the `pause` rule), we can discard the explicit `match` construct from guarded processes, as shown in the above program.



1. `0::x2::xs`
2. `0::[]`
3. `x1::x2::xs`
4. `0::xs`
5. `x::[]`
6. `x::xs`
7. `[]`
8. `ls`

Figure 3: The semi-lattice of patterns and the topological order

6 Correctness

A system written in the extended join-calculus of Section 3 is a process P. The compilation $[\![P]\!]$ replaces all the join-definitions D in P by $Y_a Y_b \dots Y_c(D)$, where $\mathsf{dn}[D] = \{a, b, \dots, c\}$. To guarantee the correctness, we require the systems before and after the compilation to be statically equivalent. Namely, the following theorem should hold.

Theorem 3. *For any process P, $[\![P]\!] \eqsim P$.*

Proof. Obviously, applying the transformation to all join-definitions in P simultaneously or one by one yields the same result. Therefore, $[\![P]\!]$ can be viewed as the result after a sequence of Y_c-formed transformations, each of which deals with one D of P *w.r.t.* some channel $c \in \mathsf{dn}[D]$. If we can prove for any Y_c-formed transformation, $\eqsim$ is preserved, then by transitivity, this theorem is also proved. □

Denote the system as $C[\mathsf{def}\ D\ \mathsf{in}\ P]$ to highlight the join-definition that we compile. To guarantee the correctness of the transformer Y_c, we require

$$C[\mathsf{def}\ D\ \mathsf{in}\ P] \eqsim C[\mathsf{def}\ Y_c(D)\ \mathsf{in}\ P]$$

which reduces to the following lemma as an immediate consequence of the fact that $\eqsim$ is closed under any context $C[\cdot]$, as stated in Theorem 2.

Lemma 4. *For any join-definition D, channel name $c \in \mathsf{dn}[D]$, and process P,*

$$\mathit{def}\ D\ \mathit{in}\ P \eqsim \mathit{def}\ Y_c(D)\ \mathit{in}\ P$$

Before giving the proof of the crucial Lemma 4, we first study some properties of the transformation of Y_c.

6.1 D and $Y_c(D)$

According to the algorithm of Section 4, there are two cases during the procedure of Y_c, as decided in Step 0:

Case "jump" : For any reaction rule of the form $J_i \,\&\, c(\pi_i) \triangleright Q_i$ in D, $i = 1 \ldots n$, the pattern π_i is irrefutable, namely, $\pi_i \equiv \_$, and in $Y_c(D)$, we have the corresponding reaction rule $J_i \,\&\, c(x_i) \triangleright \mathsf{match}\ x_i\ \mathsf{with}\ \pi_i \to Q_i$, where x_i is fresh.

Case "go through" : The general case. We recall the notations of the DAG $G(V, E)$ built by the algorithm. G has m vertices, and following the topological order, the vertices are indexed as $v_1, \ldots, v_m$. Let j ranges over $\{1, \ldots, m\}$. Each vertex v_j is labeled by a pattern, denoted as $\gamma_j = \mathsf{label}(v_j)$. Each vertex is also assigned with a fresh channel names, called c_j.

For any reaction rule of the form $J_i \,\&\, c(\pi_i) \triangleright Q_i$ in D, $i = 1 \ldots n$, there exists a unique vertex in G called v_{j_i}, such that $\gamma_{j_i} \equiv \pi_i$. We use $I(\pi_i)$ to record the indices of the predecessors of v_{j_i} as well as j_i. Notice that we have $\pi_i \preceq \gamma_j$ iff $j \in I(\pi_i)$. In $Y_c(D)$, we have a corresponding reaction rule as $J_i \,\&\, (\bigvee_{j \in I(\pi_i)} c_j(x_i)) \triangleright \mathsf{match}\ x_i\ \mathsf{with}\ \pi_i \to Q_i$, where the variable x_i is fresh. Moreover, we add a dispatcher on channel c in $Y_c(D)$ as

$$
\begin{array}{l}
c(x) \triangleright \mathsf{match}\ x\ \mathsf{with} \\
\qquad \mid \gamma_1 \to c_1(x) \\
\qquad \mid \ldots \\
\qquad \mid \gamma_m \to c_m(x) \\
\qquad \mid \_ \to \emptyset \qquad\qquad (*\text{if non-exhaustive}*)
\end{array}
$$

where x is a fresh variable.

6.2 Property of the dispatcher

Let u, v, etc. range over closed expressions, that is over *values*. In some sense, that is modulo pattern equivalence $\equiv$, the patterns of the dispatcher (the γ_j's) are all the least upper bounds of the patterns of the original D (the π_i's). Thus, the π_i's and the γ_j's admit the same instances: $\cup_{1 \le i \le n} S(\pi_i) = \cup_{1 \le j \le m} S(\gamma_j)$. As an immediate consequence, let us consider $\aleph = \{u \mid \forall i, u \notin S(\pi_i)\}$ the set of values that do not match any the the original π_i. Then, the values of $\aleph$ do not match any γ_j either, and those values are silently eaten by the dispatcher. Thus, given any value u such that there exists at least one π_i such that $u \in S(\pi_i)$, then the dispatcher must forward u onto some channel c_j. In that case, let us temporarily denote j as a function of u alone: $j = f(u)$.

Lemma 5. *For any π_i that admits value u as an instance, then $f(u) \in I(\pi_i)$.*

Proof. Given some value u, we assume the existence of π_i from the original patterns, such that $u \in S(\pi_i)$. Let K be the set of indices $\{k \mid u \in S(\pi_k)\}$ and let γ be $\uparrow_{k \in K} \pi_k$ (γ exists, since $u \in S(\pi_i)$). By steps 1–3 of the the compilation algorithm Y_c, there exists some vertex v_j with $\mathsf{label}(v_j) \equiv \gamma$.

Then, on one hand, since $\pi_i \preceq \gamma$, we have $j \in I(\pi_i)$. On the other hand, the dispatcher forwards message u onto c_j. For, we have :

1. Value u is an instance of γ.
2. Furthermore, by the subset lattice construction, γ is the most precise from the patterns of the dispatcher, such that u is an instance of γ. And thus, since the patterns of the dispatchers are ordered topologically (with precision order $\preceq$), value u cannot be an instance of the patterns of the dispatcher that appear (strictly) before γ.

□

6.3 Proof of Lemma 4

Following the definition of $\fallingdotseq$, we should prove, for any substitution δ such that $(\mathsf{def}\ D\ \mathsf{in}\ P)\delta$ and $(\mathsf{def}\ Y_c(D)\ \mathsf{in}\ P)\delta$ closed, then the two closed processes are weak barbed congruent. Because $(Y_c(D))\delta = Y_c(D\delta)$, that is

$$\mathsf{def}\ D\delta\ \mathsf{in}\ P\delta_1 \ \approx\ \mathsf{def}\ Y_c(D\delta)\ \mathsf{in}\ P\delta_1 \qquad (\dagger)$$

where $\delta_1 = \delta \setminus \mathsf{dn}[D]$, and $D\delta$, $P\delta_1$, and $Y_c(D\delta)$ are closed. We prove (†) for the two cases of Y_c respectively.

"go through" case We construct the following relation $\mathcal{R}$

$$\mathcal{R} = \{(\mathsf{def}\ D\ \mathsf{in}(P \,\&\, Q),\ \mathsf{def}\ Y_c(D)\ \mathsf{in}(P \,\&\, \widehat{Q}))\}$$

for any closed D and P, and Q and $\widehat{Q}$ stand for

$$\begin{array}{rcl} Q & = & (\prod_{\sigma \in \Sigma} c(\pi_i\sigma)) \,\&\, (\prod_{\psi \in \Psi} Q_i\psi) \,\&\, (\prod_{u \in U} c(u)) \\ \widehat{Q} & = & (\prod_{\sigma \in \Sigma} c_{f(\pi_i\sigma)}(\pi_i\sigma)) \,\&\, (\prod_{\psi \in \Psi} \mathsf{match}\ \pi_i\sigma''\ \mathsf{with}\ \pi_i \to Q_i\theta) \end{array}$$

$\prod$ is the generalized parallel composition; Σ is a multiset of substitutions on domain $\mathsf{rv}[\pi_i]$ for some i, ranged over by σ; Ψ is a multiset of substitutions on domain

$\mathsf{rv}[J_i] \uplus \mathsf{rv}[\pi_i]$ for some i, ranged over by ψ, where for any $\psi \in \Psi$, let $\sigma'' = \psi \restriction \mathsf{rv}[\pi_i]$ [3] and $\theta = \psi \restriction \mathsf{rv}[J_i]$, we have $\psi = \theta \oplus \sigma''$, and we require $\sigma'' \circ \theta = \theta \oplus \sigma''$; U is a multiset of the elements from $\aleph$. Note that Σ, Ψ and U range over all appropriate multisets.

Intuitively, we use Q and $\widehat{Q}$ to fill up the differences caused by D and $Y_c(D)$. More specifically: a message $c(\pi_i\sigma)$ may be forwarded to $c_{f(\pi_i\sigma)}(\pi_i\sigma)$ by the dispatcher in $Y_c(D)$; furthermore, if a guarded process $Q_i\psi$ is triggered from D, then from $Y_c(D)$, we have the corresponding guarded process $\mathsf{match}\ \pi_i\sigma''\ \mathsf{with}\ \pi_i \to Q_i\theta$ triggered; finally, a message on channel c with a non-matchable content will be eaten by $Y_c(D)$.

We prove $\mathcal{R}$ is a weak barbed congruence up to $\equiv$.

($\star$) For any closed executive context $E[\cdot]$, we have

$$\begin{array}{rcl} E[\mathsf{def}\ D\ \mathsf{in}(P \,\&\, Q)] & \equiv & \mathsf{def}\ D\ \mathsf{or}\ D'\ \mathsf{in}(P \,\&\, P' \,\&\, Q) \\ E[\mathsf{def}\ Y_c(D)\ \mathsf{in}(P \,\&\, \widehat{Q})] & \equiv & \mathsf{def}\ Y_c(D)\ \mathsf{or}\ D'\ \mathsf{in}(P \,\&\, P' \,\&\, \widehat{Q}) \end{array}$$

where $\mathsf{dn}[D] \cap \mathsf{dn}[D'] = \emptyset$, so that $Y_c(D)\ \mathsf{or}\ D' = Y_c(D\ \mathsf{or}\ D')$. Namely, $\mathcal{R}$ is a closure of closed executive contexts up to $\equiv$.

($\star$) We now show that $\mathcal{R}$ is a reduction bisimulation up to $\equiv$. We only detail the non-trivial cases in the following.

(1) If there is a message $c(\pi_i\sigma')$ in P, the right part can forwarded it to a message $c_{f(\pi_i\sigma')}(\pi_i\sigma')$ by the dispatcher in $Y_c(D)$. This reduction is simulated in the left part by no reduction, and we add the new substitution σ' into Σ.

(2) Similarly, if there is a message $c(u')$ in P, for some $u' \in \aleph$, the right part can eats the message by the dispatcher in $Y_c(D)$. This reduction is simulated by no reduction in the left part and we add u' into U.

(3) If a reduction uses the reaction rule $J_i \,\&\, c(\pi_i) \triangleright Q_i$ to consume a molecule $J_i\theta' \,\&\, c(\pi_i\sigma)$ in the left part for some $\sigma \in \Sigma$ and $\mathsf{dom}(\theta') = \mathsf{rv}[J_i]$, it can be simulated by consuming the molecule $J_i\theta' \,\&\, c_{f(\pi_i\sigma)}(\pi_i\sigma)$ in the right part using the corresponding reaction rule $J_i \,\&(\bigvee_{j\in I(\pi_i)} c_j(x_i)) \triangleright \mathsf{match}\ x_i\ \mathsf{with}\ \pi_i \to Q_i$, because of $f(\pi_i\sigma) \in I(\pi_i)$ by Lemma 5. The derivatives are still in $\mathcal{R}$ up to $\equiv$, with Σ shrinks to $\Sigma \setminus \{\sigma\}$, and Ψ expands to $\Psi \cup \{\theta' \oplus \sigma\}$. We assume α-conversion when necessary to guarantee $\sigma \circ \theta' = \theta' \oplus \sigma$. Vice versa.

(4) Similar to (3) but the left part consumes a molecule $J_i\theta' \,\&\, c(\pi_i\sigma')$, where σ' is not from Σ. Then the right part simulates this reduction by first forwarding the message $c(\pi_i\sigma')$ to the message $c_{f(\pi_i\sigma')}(\pi_i\sigma')$ as in (2), then consuming the molecule $J_i\sigma_i'' \,\&\, c_{f(\pi_i\sigma')}(\pi_i\sigma')$. Ψ expands to $\Psi \cup \{\theta' \oplus \sigma'\}$.

[3] $\psi \restriction \mathcal{V}$ denotes the restriction of the substitution ψ on the set of variables $\mathcal{V}$.

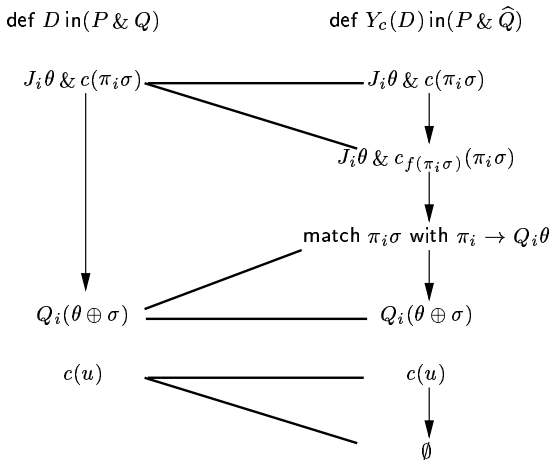


Figure 4: Reduction bisimulation in "go through" case

(5) The match $\pi_i\sigma''$ with $\pi_i \to Q_i\theta$ in $\widehat{Q}$ of the right part can be reduced to $(Q_i\theta)\sigma''$ by the MATCH rule. Because we have $\sigma'' \circ \theta = \theta \oplus \sigma''$, the result of the reduction equals to $Q_i(\theta \oplus \sigma'')$, 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 $\pi_i\sigma''$ with $\pi_i \to Q_i\theta$ from $\widehat{Q}$ into $Q_i\psi$ as in (5).

The bisimulation of reductions in $\mathcal{R}$ is also illustrated by the diagram in Figure 4.

($\star$) Considering the barb refinement. First, it is easy to check that $\mathcal{R}$ preserves strong barbs. Moreover, as discussed above, we have that $\mathcal{R}$ is also a reduction bisimulation up to $\equiv$. Therefore, $\mathcal{R}$ refines weak barbs.

The two processes in (†) satisfy relation $\mathcal{R}$ with D as $D\delta$, P as $P\delta_1$, Q and $\widehat{Q}$ as $\emptyset$, therefore satisfy the weak barb congruence $\equiv \mathcal{R} \equiv$. Hence, we proved (†) for the "go through" case.

"jump" case We build a similar relation $\mathcal{R}$, but with Q and $\widehat{Q}$ as follows:

$$\begin{array}{rcl} Q & = & \prod_{\psi\in\Psi} Q_i\psi \\ \widehat{Q} & = & \prod_{\psi\in\Psi} \text{match } \pi_i\sigma'' \text{ with } \pi_i \to Q_i\theta \end{array}$$

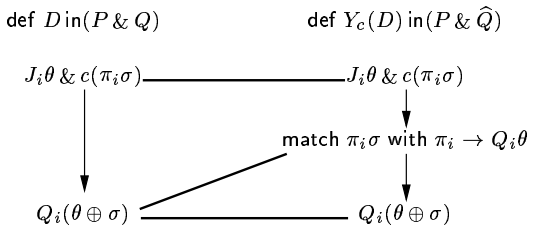


Figure 5: Reduction bisimulation in "jump" case

and the reduction bisimulation illustrated in the diagram of Figure 5.

7 Conclusion and future work

In this paper we have introduced the applied join-calculus. The applied join-calculus inherits its capabilities of communication and concurrency from the pure join-calculus and supports value-passing. The one significant extension lies in providing the power of pattern matching. Thus, the applied join-calculus is a more precise and realistic model combining both functional and concurrent programming.

Our calculus is thus "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, 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 which 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 can be seen as the combination of two basic steps: dispatching and forwarding. The design and correctness of the dispatcher essentially stems from pattern matching theory, while inserting an internal forwarding step in communications is a natural idea, which intuitively does not change process behavior. Various works give formal treatments of the intuitive correctness of forwarders, in contexts different from ours. For instance, forwarders occur in models of concrete distribution in the π-calculus [14, 8]. Of course, our interest in forwarders has quite different motivations. In particular, our dispatcher may forward messages on several channels, taking message contents into account, thereby performing some demulti-

plexing. However, the proof techniques and objective (which can be summarized as "full abstraction") are quite similar.

As regards implementation, we claim that our transformation can be integrated easily in the the current JoCaml system [10]. The JoCaml system is built on top of Objective Caml [12], a dialect of ML, which features a sophisticated pattern matching compiler [11]. Our transformation naturally takes place between the typing and pattern matching compilation phases of the existing compiler. More significantly, this should be the only addition. In particular, our solution does not require any modification of the existing runtime system since the join-pattern synchronization remains as before. It is worth observing that a direct implementation of 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 a side note, an attempt to transplant our compilation scheme to a similar extension of the π-calculus would be problematic, since, in the π-calculus, the receivers on some channel are scattered in the full program and might even not be known statically.

The integration of pattern matching into the join-calculus is part of our effort to develop a practical concurrent programming language with firm semantical foundations (a similar effort is for instance Scala [18]). 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 introduce algebraic patterns as formal arguments of channel definitions. Doing so, we provide a more convenient (or "expressive") language to programmers. From that perspective, pattern matching and join-calculus appear to live well together, with mutual benefits.

In previous work, we have designed an object-oriented extension of the join-calculus [6, 13], 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.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.

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

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

[4] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Palaiseau, Nov. 1998.

[5] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 372–385, 1996.

[6] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57:23–69, 2003.

[7] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of 8th International Conference on Concurrency Theory (CONCUR'97)*, LNCS 1243, pages 196–212, 1997.

[8] P. Gardner, C. Laneve, and L. Wischik. Linear forwarders. In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR 2003)*, LNCS 2761, pages 415–430, 2003.

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

[10] F. Le Fessant. The JoCaml system. Software and documentation available at `http://pauillac.inria.fr/jocaml`, 1998.

[11] F. Le Fessant and L. Maranget. Optimizing pattern-matching. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001)*, pages 26–37, 2001.

[12] X. Leroy, D. Doligez, and *et al.* the Objective Caml System, version 3.07. Software and documenation available at `http://caml.inria.fr/`, 2003.

[13] Q. Ma and L. Maranget. Expressive synchronization types for inheritance in the join calculus. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS 2003)*, LNCS 2895, pages 20–36, 2003.

[14] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, LNCS 1443, pages 856–867, 1998.

[15] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.

[16] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming (ICALP'92)*, volume LNCS 623, pages 685–695, 1992.

[17] M. Odersky. Functional nets. In *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, LNCS 1782, pages 1–25, 2000.

[18] M. Odersky. the Scala Language. Reference available at `http://lamp.epfl.ch/~odersky/scala/`, 2002.

[19] P. Pritchard. On computing the subset graph of a collection of sets. *Journal of Algorithms*, 33(2):187–203, 1999.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399