Compiling Pattern Matching in Join-Patterns

Qin Ma and Luc Maranget

INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
{Qin.Ma, 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.

1 Introduction

The join-calculus [5] is a process calculus in the tradition of the π-calculus of Milner, Parrow and Walker [16]. 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 fire 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]

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

The second join-pattern $\mathit{push}(v)\ \&\ \mathit{State}(\mathit{ls})$ 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.

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

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 extended join-patterns, by using an extra private channel and ML pattern matching in guarded processes instead:

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

However, 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 ML pattern matching thus never fails.

Join-definitions with (constant) pattern arguments appear informally in functional nets [18]. Here we generalize this idea to full patterns. 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 more than just straightforward. Namely, 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 $\{[]\}$ 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 patterns and ML pattern matching, then goes on presenting the key ideas of our solution to carry out the transformation informally. We explain how the idea come into being step by step, and especially how we deal with the non-determinism problem. Section 3 presents the semantics of our extended calculus as well as appropriate equivalence relations. Section 4 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 5, and finally, we deal with the correctness of the compilation scheme in Section 6. Most of the proof details are confined to the complementary technical report [14] for lack of space.

2 A journey through patterns

2.1 Patterns and ML pattern matching

Patterns and values are built as usual as (well-sorted) terms, over constructor signatures defined by algebraic data types. In contrast to values, patterns may have variables[2] in them; we require all variables in a pattern to be pairwise distinct. That is, patterns are *linear*. 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] for full details):

- 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 $\pi_i \updownarrow \pi_2$.[3]

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

$$\mathsf{match}\ e\ \mathsf{with}\ \mid \pi_1 \rightarrow Q_1 \mid \pi_2 \rightarrow Q_2 \ \mid \ldots \mid \pi_n \rightarrow Q_n$$

Pattern π_i is matched by the values in set $S(\pi_i) \setminus (\cup_{1 \le j < i} S(\pi_j))$ and only by those. In other words, given some value v, patterns $\pi_1, \pi_2, \ldots, \pi_n$ are checked for having v as an instance, in that order, stopping as soon as a match is found. pattern matching is exhaustive when $\cup_{1 \le i \le n} S(\pi_j)$ is the whole set of values (of the considered type).

2.2 Transform pattern arguments in join to ML pattern matching

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

$$\begin{array}{ll} \mathsf{def} & c(\pi_1)\ \&\ d(\ldots) \triangleright P_1 \\ \mathsf{or} & c(\pi_2)\ \&\ e(\ldots) \triangleright P_2 \end{array}$$

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

[2] In patterns, we freely replace variables whose names are irrelevant by *wild-cards* "_".

[3] Because of typing, there exists non-trivial equivalences such as at any pair type, we have $\_ \equiv (\_, \_)$.

$$\begin{array}{l} \textsf{def}\ \ c_{\pi_1}(\ldots)\ \&\ d(\ldots) \triangleright P_1 \\ \ \ \textsf{or}\ \ c_{\pi_2}(\ldots)\ \&\ e(\ldots) \triangleright P_2 \end{array}$$

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

$$\begin{array}{l} \textsf{or}\ \ c(v) \triangleright \textsf{match}\ v\ \textsf{with} \\ \qquad\qquad \mid\ \pi_1 \to c_{\pi_1}(...) \\ \qquad\qquad \mid\ \pi_2 \to c_{\pi_2}(...) \\ \qquad\qquad \mid\ \_ \to \emptyset \end{array}$$

The notation $\emptyset$ stands for the null process, which 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, namely, the non-determinism problem. 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:

  $$\begin{array}{l} \textsf{or}\ \ c(v) \triangleright \textsf{match}\ v\ \textsf{with} \\ \qquad\qquad \mid\ \pi_2 \to c_{\pi_2}(...) \\ \qquad\qquad \mid\ \pi_1 \to c_{\pi_1}(...) \\ \qquad\qquad \mid\ \_ \to \emptyset \end{array}$$

  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:

  $$\begin{array}{l} \textsf{def}\ \ c_{\pi_1}(\ldots)\ \&\ d(\ldots) \triangleright P_1 \\ \ \ \textsf{or}\ \ c_{\pi_2}(\ldots)\ \&\ d(\ldots) \triangleright P_1 \\ \ \ \textsf{or}\ \ c_{\pi_2}(\ldots)\ \&\ e(\ldots) \triangleright P_2 \end{array}$$

  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:

  $$\begin{array}{l} \textsf{def}\ \ (c_{\pi_1}(\ldots)\ \textsf{or}\ c_{\pi_2}(\ldots))\ \&\ d(\ldots) \triangleright P_1 \\ \ \ \textsf{or}\ \ c_{\pi_2}(\ldots)\ \&\ e(\ldots) \triangleright P_2 \end{array}$$

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

  $$\begin{array}{l} \textsf{def}\ \ c_{\pi_1 \updownarrow \pi_2}(\ldots)\ \&\ d(\ldots) \triangleright P_1 \\ \ \ \textsf{or}\ \ c_{\pi_1 \updownarrow \pi_2}(\ldots)\ \&\ e(\ldots) \triangleright P_2 \\ \ \ \textsf{or}\ \ c(v) \triangleright \textsf{match}\ v\ \textsf{with} \\ \qquad\qquad \mid\ \pi_1 \updownarrow \pi_2 \to\ \ c_{\pi_1 \updownarrow \pi_2}(\ldots) \\ \qquad\qquad \mid\ \_ \to \emptyset \end{array}$$

- 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:

  $$\begin{array}{l} \textsf{def}\ \ (c_{\pi_1}(\ldots)\ \textsf{or}\ c_{\pi_1 \uparrow \pi_2}(\ldots))\ \&\ d(\ldots) \triangleright P_1 \\ \ \ \textsf{or}\ \ (c_{\pi_2}(\ldots)\ \textsf{or}\ c_{\pi_1 \uparrow \pi_2}(\ldots))\ \&\ e(\ldots) \triangleright P_2 \\ \ \ \textsf{or}\ \ c(v) \triangleright \textsf{match}\ v\ \textsf{with} \\ \qquad\qquad \mid\ \pi_1 \uparrow \pi_2 \to c_{\pi_1 \uparrow \pi_2}(\ldots) \end{array}$$

$| \ \pi_1 \rightarrow c_{\pi_1}(...)$
$| \ \pi_2 \rightarrow c_{\pi_2}(...)$
$| \ \_ \rightarrow \emptyset$

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 $c_\pi(\ldots)$. 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:

$\mathsf{def}\ \mathit{pop}(r)\ \&\ \mathit{State}_{x::xs}(z) \triangleright \mathsf{match}\ z\ \mathsf{with}\ x::xs \rightarrow r(x)\ \&\ \mathit{State}(xs)$
$\mathsf{or}\ \mathit{push}(v)\ \&\ (\mathit{State}_{x::xs}(z)\ \mathsf{or}\ \mathit{State}_{ls}(z)) \triangleright \mathsf{match}\ z\ \mathsf{with}\ ls \rightarrow \mathit{State}(v::ls)$
$\mathsf{or}\ \mathit{State}(v) \triangleright \mathsf{match}\ v\ \mathsf{with}$
$| \ x::xs \rightarrow \mathit{State}_{x::xs}(v)$
$| \ ls \rightarrow \mathit{State}_{ls}(v)$

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 first 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

In this section, 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 the pure join-calculus [5]. Moreover it extends to support algebraic value-passing, and algebraic pattern matching in both join-patterns and guarded processes.

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, 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

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

Fig. 1. Syntax of the applied join-calculus

extensions: first, message contents become expressions, that 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.

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

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

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

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

Side conditions:

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

Fig. 2. RCHAM of the applied join-calculus

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, 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 [17], and adapted to many other process calculi [9, 2], including the join-calculus [5]. We take *weak barbed congruence* [17] 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 $P'\mathcal{R}Q'$, and 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. Similarly to what Fournet has established for the pure join-calculus in his thesis [4], we first have the property that $\approx$ is closed under substitution because, roughly, name substitutions may be mimicked by executive contexts with "forwarders".

Lemma 1. *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".)*

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

Theorem 1. *Weak barbed congruence $\approx$ is closed under application of any closed context.*

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 ($\Bumpeq$), 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 $\Bumpeq$ is closed by substitution.

Lemma 2. $P \Bumpeq Q \Longrightarrow \forall \sigma . P\sigma \Bumpeq Q\sigma$

More importantly, $\Bumpeq$ is also closed with respect to all contexts. Namely, the following theorem holds.

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

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

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

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 [20].

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: $\Pi = \pi_1; \pi_2; \ldots; \pi_n$.
2. Let Π' be formed from Π by taking the $\updownarrow$ of all equivalent patterns; thus Π' is a sequence of pairwise non-equivalent patterns.
3. Perform exhaustiveness check on Π', if not exhaustive, issue a warning.
4. **IF** There is only one pattern in Π', and that Π' is exhaustive
   THEN goto Step 5 (In that case, no dispatching is needed.)

Step 1: Closure of least 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 $\Gamma = F(\Pi')$. It is worth noticing that Γ 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 Π' as $\pi'_1; \pi'_2; \ldots; \pi'_{n'}$.

Step 2: Up to equivalence

As in Step 0.2, build Γ' from Γ.

Step 3: Build DAG

Corresponding to the semi-lattice $(\Gamma', \preceq)$, build a directed acyclic graph $G(V, E)$.

1. $V = \emptyset, E = \emptyset$.
2. For each pattern γ in Γ', 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

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 "$\mid \mathsf{label}(v_j) \to c_j(x)$" to $\mathcal{L}$, where c_j is a fresh channel name.
2. If Π' is not exhaustive, then add a rule "$\mid \_ \to \emptyset$" 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 $Q'_i = \mathsf{match}\ x_i\ \mathsf{with}\ \pi_i \to Q_i$, 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.

Assume $\mathsf{dn}[D] = \{c_1, \ldots, c_n\}$ $(n \geq 0)$, where we assume an order on the channel names. To transform a join-definitions D, we just apply $Y_{c_n} \ldots Y_{c_1}(D)$. And the compilation of processes $[\![\cdot]\!]$ is defined as follows.

$$
\begin{aligned}
[\![\emptyset]\!] &\stackrel{\text{def}}{=} \emptyset \\
[\![x(e)]\!] &\stackrel{\text{def}}{=} x(e) \\
[\![P \,\&\, P']\!] &\stackrel{\text{def}}{=} [\![P]\!] \,\&[\![P']\!] \\
[\![\mathsf{def}\ D \ \mathsf{in}\ P]\!] &\stackrel{\text{def}}{=} \mathsf{def}\ Y_{c_n} \ldots Y_{c_1}(D)\ \mathsf{in}[\![P]\!] \\
[\![\mathsf{match}\ e\ \mathsf{with}\ |^{i \in I}\ \pi_i \rightarrow P_i]\!] &\stackrel{\text{def}}{=} \mathsf{match}\ e\ \mathsf{with}\ |^{i \in I}\ \pi_i \rightarrow [\![P_i]\!]
\end{aligned}
$$

Observe that the compilation preserves the interface of join-definitions. Namely, it only affects definitions D, while message sending remains the same.

5 Example of compilation

Given the following join-definition of an enriched integer stack

$\mathsf{def}\ \mathit{push}(v)\ \&\ \mathit{State}(\mathit{ls}) \triangleright \mathit{State}\ (v \mathrel{::} \mathit{ls})$
$\mathsf{or}\ \mathit{pop}(r)\ \&\ \mathit{State}(x \mathrel{::} \mathit{xs}) \triangleright r(x)\ \&\ \mathit{State}(\mathit{xs})$
$\mathsf{or}\ \mathit{insert}(n)\ \&\ \mathit{State}(0 \mathrel{::} \mathit{xs}) \triangleright \mathit{State}(0 \mathrel{::} n \mathrel{::} \mathit{xs})$
$\mathsf{or}\ \mathit{last}(r)\ \&\ \mathit{State}(\mathtt{[}x\mathtt{]}) \triangleright r(x)\ \&\ \mathit{State}(\mathtt{[}x\mathtt{]})$
$\mathsf{or}\ \mathit{swap}()\ \&\ \mathit{State}(x_1 \mathrel{::} x_2 \mathrel{::} \mathit{xs}) \triangleright \mathit{State}(x_2 \mathrel{::} x_1 \mathrel{::} \mathit{xs})$
$\mathsf{or}\ \mathit{pause}(r)\ \&\ \mathit{State}(\mathtt{[]}) \triangleright r()$
$\mathsf{or}\ \mathit{resume}(r) \triangleright \mathit{State}(\mathtt{[]})\ \&\ r()$

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 $\mathit{\Pi}$

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

Because none of these patterns is equivalent to another, $\mathit{\Pi}' = \mathit{\Pi}$. Additionally, $\mathit{\Pi}'$ is exhaustive (pattern *ls* alone covers all possibilities).

Step 1,2 $\mathit{\Gamma}$ extends $\mathit{\Pi}'$ with all possible least upper bounds. Then we form $\mathit{\Gamma}'$ from $\mathit{\Gamma}$ by taking the $\updownarrow$ of all equivalent patterns.

$$\mathit{\Gamma}' = \mathit{ls};\ x \mathrel{::} \mathit{xs};\ 0 \mathrel{::} \mathit{xs};\ \mathtt{[}x\mathtt{]};\ x_1 \mathrel{::} x_2 \mathrel{::} \mathit{xs};\ \mathtt{[]};\ 0 \mathrel{::} x_2 \mathrel{::} \mathit{xs};\ \mathtt{[0]}$$

Note that the last two patterns are new, where

$$\begin{aligned} 0\texttt{::}x_2\texttt{::}xs &= 0\texttt{::}xs \uparrow x_1\texttt{::}x_2\texttt{::}xs \\ \texttt{[}0\texttt{]} &= 0\texttt{::}xs \uparrow \texttt{[}x\texttt{]} \end{aligned}$$

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

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

$$\begin{aligned} \textsf{or } \mathit{State}(y) \triangleright \textsf{match } y \textsf{ with}\ & \\ &\mid 0\texttt{::}x_2\texttt{::}xs \rightarrow \mathit{State}_1(y) \\ &\mid \texttt{[}0\texttt{]} \rightarrow \mathit{State}_2(y) \\ &\mid x_1\texttt{::}x_2\texttt{::}xs \rightarrow \mathit{State}_3(y) \\ &\mid 0\texttt{::}xs \rightarrow \mathit{State}_4(y) \\ &\mid \texttt{[}x\texttt{]} \rightarrow \mathit{State}_5(y) \\ &\mid x\texttt{::}xs \rightarrow \mathit{State}_6(y) \\ &\mid \texttt{[]} \rightarrow \mathit{State}_7(y) \\ &\mid ls \rightarrow \mathit{State}_8(y) \end{aligned}$$

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

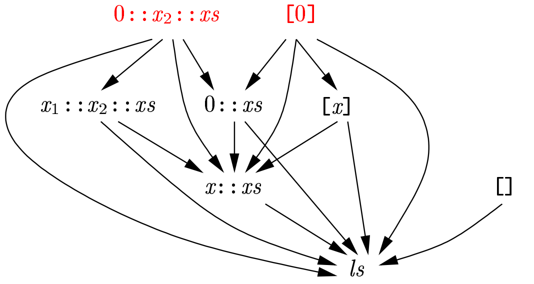
$$\begin{aligned} &\mathit{insert}(n)\ \&\ (\mathit{State}_1(x_3) \textsf{ or } \mathit{State}_2(x_3) \textsf{ or } \mathit{State}_4(x_3)) \\ &\qquad \triangleright \textsf{match } x_3 \textsf{ with } 0\texttt{::}xs \rightarrow \mathit{State}(0\texttt{::}n\texttt{::}xs) \end{aligned}$$

where State_1, State_2 and State_4 are the fresh channel names corresponding to vertices 1, 2, 4 respectively, and 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.

$$\begin{aligned} &\textsf{def } \mathit{push}(v)\ \&\ (\mathit{State}_1(x_1) \textsf{ or } \dots \textsf{ or } \mathit{State}_8(x_1)) \\ &\qquad \triangleright \textsf{match } x_1 \textsf{ with } ls \rightarrow \mathit{State}\ (v\texttt{::}ls) \\ &\textsf{or } \mathit{pop}(r)\ \&\ (\mathit{State}_1(x_2) \textsf{ or } \dots \textsf{ or } \mathit{State}_6(x_2)) \\ &\qquad \triangleright \textsf{match } x_2 \textsf{ with } x\texttt{::}xs \rightarrow r(x)\ \&\ \mathit{State}(xs) \\ &\textsf{or } \mathit{insert}(n)\ \&\ (\mathit{State}_1(x_3) \textsf{ or } \mathit{State}_2(x_3) \textsf{ or } \mathit{State}_4(x_3)) \\ &\qquad \triangleright \textsf{match } x_3 \textsf{ with } 0\texttt{::}xs \rightarrow \mathit{State}(0\texttt{::}n\texttt{::}xs) \\ &\textsf{or } \mathit{last}(r)\ \&\ (\mathit{State}_2(x_4) \textsf{ or } \mathit{State}_5(x_4)) \\ &\qquad \triangleright \textsf{match } x_4 \textsf{ with } \texttt{[}x\texttt{]} \rightarrow r(x)\ \&\ \mathit{State}(\texttt{[}x\texttt{]}) \\ &\textsf{or } \mathit{swap}()\ \&\ (\mathit{State}_1(x_5) \textsf{ or } \mathit{State}_3(x_5)) \\ &\qquad \triangleright \textsf{match } x_5 \textsf{ with } x_1\texttt{::}x_2\texttt{::}xs \rightarrow \mathit{State}(x_2\texttt{::}x_1\texttt{::}xs) \\ &\textsf{or } \mathit{pause}(r)\ \&\ \mathit{State}_7(x_6) \triangleright r() \\ &\textsf{or } \mathit{resume}(r) \triangleright \mathit{State}(\texttt{[]})\ \&\ r() \end{aligned}$$

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 ML pattern matching, as shown in the above program.



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

Fig. 3. The semi-lattice of patterns and the topological order

6 Correctness

A program 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 $Y_{c_n} \dots Y_{c_1}(D)$, where $\mathsf{dn}[D] = \{c_1, \dots, c_n\}$. To guarantee the correctness, we require the programs before and after the compilation to be statically equivalent. Namely, the following theorem should hold.

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

Proof. By structural induction on processes, and because $\Bumpeq$ is a full congruence and a transitive relation, it suffices to prove the following Lemma 3. □

Lemma 3. *For any join-definition D, channel name $c \in \mathsf{dn}[D]$, and process P,*

$$\textit{def}\, D\, \textit{in}\, P \Bumpeq \textit{def}\, Y_c(D)\, \textit{in}\, P$$

This lemma is crucial to the correctness of the compilation. The proof relies on the properties of the meet semi-lattice constructed from the pattern arguments. In particular the proof exploits the deterministic semantics of the ML pattern matching in the dispatcher, which is built following the topological order of the lattice. For lack of space, we omit the proof. Please refer to the complementary technical report.

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 [15, 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 demultiplexing. 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 current JoCaml system [10]. The JoCaml system is built on top of Objective Caml [12], a dialect of ML, which features a sophisticated ML pattern matching compiler [11]. Our transformation naturally takes place between the typing and ML 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 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.

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 [19]). 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.

Acknowledgements. The authors wish to thank James Leifer and Jean-Jacques Lévy for fruitful discussions and comments.

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of 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, Nov. 1998.
5. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of 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(1-2):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 CONCUR'97*, LNCS 1243, pages 196–212, 1997.
8. P. Gardner, C. Laneve, and L. Wischik. Linear forwarders. In *Proceedings of CONCUR'03*, 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. `http://pauillac.inria.fr/jocaml`, 1998.
11. F. Le Fessant and L. Maranget. Optimizing pattern-matching. In *Proceedings of ICFP'01*, pages 26–37, 2001.
12. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System, version 3.07. `http://caml.inria.fr/`, 2003.
13. Q. Ma and L. Maranget. Expressive synchronization types for inheritance in the join calculus. In *Proceedings of APLAS'03*, LNCS 2895, pages 20–36, 2003.
14. Q. Ma and L. Maranget. Compiling pattern matching in join-patterns. Rapport de recherche 5160, INRIA-Rocquencourt, Apr. 2004. Available at `http://pauillac.inria.fr/~ma/papers/ptjoin-tr.ps`.
15. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of ICALP'98*, LNCS 1443, pages 856–867, 1998.
16. 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.
17. R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings of ICALP'92*, volume LNCS 623, pages 685–695, 1992.
18. M. Odersky. Functional nets. In *Proceedings of ESOP'00*, LNCS 1782, pages 1–25, 2000.
19. M. Odersky. The Scala Language. `http://lamp.epfl.ch/~odersky/scala/`, 2002.
20. P. Pritchard. On computing the subset graph of a collection of sets. *Journal of Algorithms*, 33(2):187–203, 1999.