Compiling Join-Patterns *

Luc Maranget   Fabrice Le Fessant

INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex France.

{Luc.Maranget, Fabrice.Le-Fessant}@inria.fr

Abstract

The join-calculus is both a name passing calculus and a core language for concurrent and distributed programming. An essential part of its implementation is the compilation of *join-patterns*. Join-patterns define new channels and all the synchronizations they take part to at the same time. Relying on the experience based on our two implementations, we study the translation of join-patterns into deterministic finite-state automata as well as some related optimizations.

1 Introduction

Join-pattern is the distinctive feature of the join-calculus, seen both as a process calculus and as a programming language. On the calculus side, join-calculus can roughly be seen as a functional calculus plus join-patterns, thus achieving the same expressive power as previous name-passing process calculi [11]. Join-definitions are made of several clauses, each clause being a pair of a join-pattern and of a guarded process. A join-pattern expresses a synchronization between several *names* (or channels). When messages are pending on all the names that appear in a given join-pattern, then the corresponding clause is said to be active and its guarded process *may* be fired. A definition whose join-patterns share some names expresses sophisticated synchronizations. In such a definition, a message on a name that appears in several active clauses is consumed as soon as one of the corresponding guarded processes is fired.

Join-languages are built on top of the join-calculus taken as a core language. Therefore, names are first-class citizens, computations are first abstracted as collections of asynchronous processes, and join-patterns provide an unique, clear and powerful mechanism for synchronizing these computations. The documentation for the join-language [8] includes a tutorial that shows how join definitions may encode classical synchronization constructs such as locks, barriers, shared counters,...

On the implementation side, join-patterns are meant to be heavily used by programmers, as the only synchronization primitive available. Thus, their compilation requires much care. At the moment, we propose two compilers: the `join` compiler [8], a language of its own, and the `jocaml` compiler [4], an extension of the Objective Caml functional language.

Section 2 of this paper succinctly presents the join-calculus syntax and semantics. Then, section 3 introduces the kind of automata we use to compile join-synchronization, while section 4 presents two techniques for implementing them. The first technique directly derives from automata description and is used in our `join` compiler. The second technique performs some extra runtime tests, this is the technique used in our `jocaml` compiler. Sections 5 and 6 discuss optimizations and section 7 concludes.

---

*This work is partly supported by the ESPRIT CONFER-2 WG-21836

2 A rapid tour of the join-calculus

2.1 Syntax

We here rephrase the traditional presentation of the core join-calculus [5], where names are the only value. Thus, we ignore the issue of system primitives and constants, since names provide sufficient expressive power for our purpose of describing our implementation of pattern matching (However we use primitives and constants in our examples). We slightly change the syntax of [5], in order to match the one of the `join` programming language.

We use x to denote a name in general.

$$\begin{array}{rcl} P & ::= & x(x_i{}^{i \in 1..p}) \\ & | & \texttt{let } D \texttt{ in } P \\ & | & P \mid P \end{array} \qquad \begin{array}{rcl} D & ::= & J = P \\ & | & D \texttt{ and } D \\ \\ J & ::= & x(x_i{}^{i \in 1..p}) \\ & | & J \mid J \end{array}$$

A process P is either a message, a defining process, or a parallel composition of processes (note that names are polyadic, meaning that messages may be made of several values) ; a definition D consists of one or several clauses $J = P$ that associate a guarded process P to a specific message pattern J ; a join-pattern J consists of one or several messages in parallel. We say that the pattern $J = \ldots x(x_i{}^{i \in 1..p}) \ldots$ defines the name x and that a definition defines the set of the names defined by its patterns. Moreover, patterns are linear, i.e. names may appear at most once in a given pattern.

Processes and definitions are known modulo renaming of bound variables, as substitution performs α-conversion to avoid captures.

2.2 Semantics

This semantics is specified as a reflexive chemical abstract machine (RCHAM) [3]. The state of the computation is a *chemical soup* $\mathcal{D} \Vdash \mathcal{P}$ that consists of two multisets : active definitions $\mathcal{D}$ and running processes $\mathcal{P}$.

The chemical soup evolves according to two families of rules : *Structural rules* $\rightleftharpoons$ are reversible ($\rightharpoonup$ is heating, $\leftharpoondown$ is cooling) ; they represent the syntactical rearrangement of terms (heating breaks terms into smaller ones, cooling builds larger terms from their components). *Reduction rules* $\longrightarrow$ consume specific processes present in the soup, replacing them by some others ; they are the basic computation steps.

We present simplified chemical rules (see [5, 6] for the complete set of rules). Following the chemical tradition, every rule applies on any matching subpart of the soup, non-matching sub-parts of the soup being left implicit.

$$\begin{array}{rlcrll} & \Vdash P_1 \mid P_2 & \rightleftharpoons & & \Vdash P_1, P_2 & \text{S-Par} \\ D_1 \texttt{ and } D_2 & \Vdash & \rightleftharpoons & D_1, D_2 & \Vdash & \text{S-And} \\ & \Vdash \texttt{let } D \texttt{ in } P & \rightleftharpoons & D & \Vdash P & \text{S-Def} \\ \\ J = P & \Vdash \varphi(J) & \longrightarrow & J = P & \Vdash \varphi(P) & \text{R-}\beta \end{array}$$

Two of the rules above have side conditions :

- (S-Def) the names defined in D must not appear anywhere in solution but in the reacting process and definition D and P. This condition is global ; in combination with α-renaming it enforces lexical scoping.
- (R-β) $\varphi(\cdot)$ substitute actual names for the received variables in J and P.

Additionally, we only consider well-typed terms and reductions. See [7] for details on a rich polymorphic type system for the join calculus. Here, this mostly amounts to assuming that message and name arity always agree.

In this paper, we take particular interest in the reduction (R-β). Informally, when there are messages pending on all the names defined in a given join-pattern, then the process guarded by this join-pattern may be fired. When firing is performed, we say that a *matching* occurs. On the semantics level, there is a message $(x_i\ ^{i\in 1..p})$ pending on a name x when there is an active molecule $x(x_i\ ^{i\in 1..p})$ in the chemical soup.

Thus, we may define the reactivity status of a given chemical soup as the multiset of the active molecules in it. Later on in this paper, we shall consider various abstractions of this reactivity status.

2.3 The join programming languages

Apart from primitives, join-languages support *synchronous* names, which the core join-calculus does not provide directly. Synchronous names send back results, a bit like functions. However synchronous names may engage in any kind of join-synchronization, just as asynchronous names do. A program written using synchronous names can be translated into the core join-calculus alone. The translation is analogous to continuation passing style transformation in the λ-calculus. In our implementation, as far as pattern matching is concerned, a synchronous name behave like it was asynchronous and carried one additional continuation argument. All implementation difficulties concentrate in managing this extra argument, whose presence had no effect on pattern matching itself.

The `join` language [8] is our first prototype. All examples in this paper are in `join` syntax. The system consists in a bytecode compiler and a bytecode interpreter. Both compiler and interpreter are Objective Caml [9] programs and it is easy to lift Objective Caml data types and functions into `join` abstract data types and primitives. For instance, `join` programs easily draw graphics, using the graphics Objective Caml library. As a consequence, `join` can be seen either as a language of its own, featuring many primitives, or as a distributed layer on top of Objective Caml. Continuations are encoded using ad hoc threads, which are created and scheduled by the `join` bytecode interpreter.

The `jocaml` system is our second implementation. In `jocaml`, all join-calculus constructs for concurrency, communication, synchronization and process mobility are directly available as syntactical extensions to Objective Caml. On the runtime environment side, we have supplemented the original Objective Caml runtime system (which already provides a thread library) with a special "join" library and a distributed garbage collector [13]. On the compiler side, the Objective Caml compiler has been extended to translate join-calculus source code into functions calls to the "join" library. However, we also introduced a few new instructions to Objective Caml bytecode, but only to handle code mobility, a feature orthogonal to pattern matching. The `jocaml` system is currently available as a prototype version [4].

3 Pattern matching in join definitions

3.1 Principle

Consider the following join definition :

```
let A(n) | B() = P(n)
and A(n) | C() = Q(n)
;;
```

This defines three names A, B and C. Name A has arity one, whereas names B and C have arity zero. Names may be synchronous or asynchronous, depending on whether there are `reply ... to ...` constructs applying to them inside the guarded processes $P(n)$ and $Q(n)$ or not.

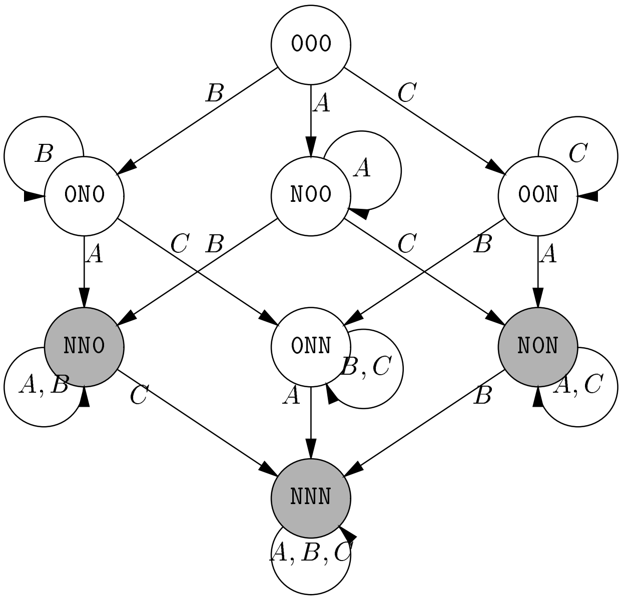
According to the general join-calculus semantics, the guarded process $P(n)$ may be fired whenever there are some messages pending on A and B. Similarly, $Q(n)$ may be fired whenever there are some messages pending on A and C. In both cases, the formal parameter n is replaced by (or bound to in the implementation) one of the messages pending on A.

Reactivity information is to be considered at the definition level, since matching is indeed performed at this level. Moreover, in order to use finite-state automata, we want this information to range on a finite set of possible values. As far as matching is concerned and by the linearity of patterns, only the presence or absence of messages matters. Thus, let us call `0` the status of a name without any message pending, and `N` the status of a name with at least one message pending. Then, the status of a definition is a tuple consisting of the statuses of the names it defines, once some arbitrary order of these names has been adopted.

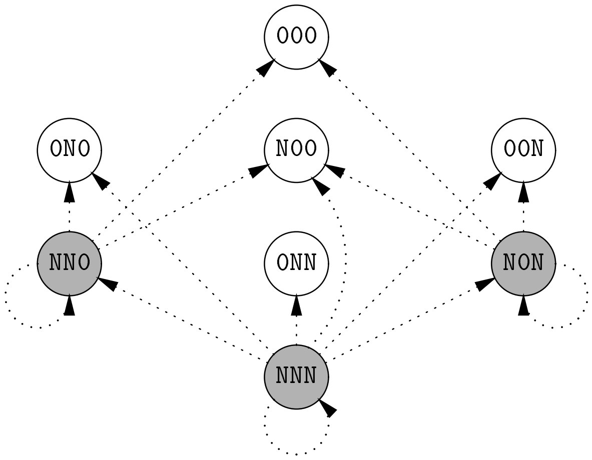
For instance, if some messages are pending on B and C, whereas none is pending on A, then the status of the A, B, C definition is a three-tuple written `ONN`.

A *matching status* is defined as a status that holds enough `N`, so that at least one guarded process can be fired.

Definition status evolves towards matching status as messages arrive. This yields a first kind of "increasing" transitions. More specifically, when a message arrives on some name, then this name status either evolves from `O` to `N` or remains `N`. Definition status evolves accordingly. In the A, B, C case we get the following transitions. (In this diagram, transitions are labeled by the name that gets a new message and matching statuses are filled in gray.)



Definition status also evolves when matching occurs. This yields new, "decreasing", transitions that we call matching transitions. According to the join-calculs semantics, matching may occur at any moment (provided of course that matching is possible). As a consequence, matching transitions start from matching states and they are unlabelled. In the A, B, C case, they are as follows:



Observe that there may be several matching transitions starting from a given status. This is not always a consequence of the non-deterministic semantics of the join-calculus.

Often, ambiguity is only apparent. For instance, matching transitions starting from `NNO` lead to `NNO`, `NOO`, `ONO` and `OOO`. When such a matching occurs, two messages are consumed (one pending on A and one pending on B) then, depending on whether there are some messages left pending on A and B or not, status evolves to `NNO`, `NOO`, `ONO` or `OOO`. From the implementation point of view, this means that a little runtime testing is required once matching has been performed. Here, we pay a price for using finite-state automata.

However, some true non-determinism is still present. Consider status NNN for instance. Then, both guarded processes of the A, B, C definition can now be fired. The choice of firing either $P(n)$ or $Q(n)$ will result in either consuming one message pending on A and one on B, or consuming one message pending on A and one on C.

Finally, a view of join-matching compilation can be given by taking together both kinds of transitions. This yields a non-deterministic automaton.

Note that matching of non-linear patterns can also be compiled using automata. For instance, if a name appears at most twice in one or more pattern, then it status will ramge on 0, 1 and N. We do not present this extension in greater detail, for simplicity, and because we do not implement non-linear patterns.

3.2 Towards deterministic automata

For efficiency and simplicity reasons we choose to implement matching using deterministic automata that react to message reception.

Fortunately, it is quite possible to do so. It suffices to perform matching as soon as possible. More precisely, when a message arrives and carries definition status into matching status, matching is performed immediately, while definition status is adjusted to reflect message consumption. This results in pruning the status space just below matching statuses.

In practise, in the A, B, C case, we get the automaton of figure 1.

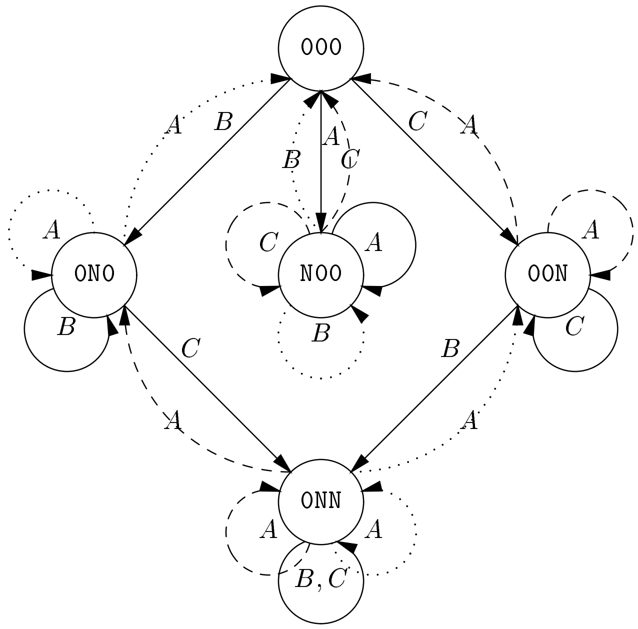


Figure 1: Automaton in the A, B, C case

Observe that all transitions are now labeled and that a name labels a transition when message reception on this name triggers that transition. Furthermore, matching transitions that correspond to firing $P(n)$ or firing $Q(n)$ are now represented differently (the former by a dotted arrow, the latter by a dashed arrow). This highlights the difference between false and true non-deterministic transitions: real non-determinism is present when there are both dotted and dashed edges with the same label starting from the same node.

For instance, there are two B-labeled dotted transitions starting from NOO. Non-determinism is only apparent here, since $P(n)$ is fired in both cases and that the selected transition depends only on whether there is at least one message left pending on A or not after firing $P(n)$.

By contrast, from status ONN, the automaton may react to the arrival of a message on A in two truly different manners, by firing either $P(n)$ or $Q(n)$. This is clearly shown in figure 1 by the A-labeled edges that start from status ONN, some of them being dashed and the others being dotted. A simple way to avoid such a non-deterministic choice at run-time is to perform it at compile-time. That is, here, we suppress either dotted or dashed A-labeled transitions that start from `ONN`.

In the rest of the paper, we take automata such as the one of figure 1 as suitable abstractions of join-pattern compilation output.

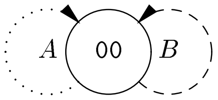
3.3 Automata and semantics

Both the "match as soon as possible" behavior and the deletion of some matching transitions have a price in terms of semantics. More precisely, some CHAM behaviors now just cannot be observed anymore. However, the CHAM semantics is a non-deterministic one : an initial configuration of the CHAM *may* evolve into a variety of configurations. Furthermore, there is no fairness constraint of any kind and no particular event is required to occur.

As an example of the consequence of the "match as soon as possible" behavior, consider this definition :

```
let A() = print(1);
and B() = print(2);
and A() | B() = print(3);
;;
```

Then, we get the following automaton :

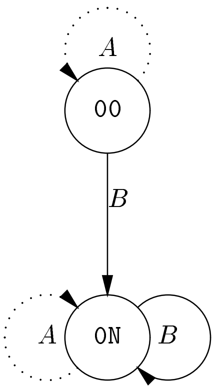


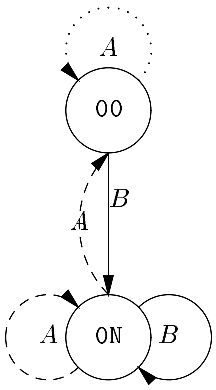
Status `NN` that is preceded by the two matching statuses `ON` and `NO` cannot be reached. As a consequence, the above program will never print a `3`, no matter how many messages are sent on A and B.

Next, to illustrate the effect of deleting ambiguous matching transitions, consider the following definition :

```
let A() = print(1);
and A() | B() = print(2);
```

Such a definition will get compiled into one of the following deterministic automata :





In the case of the left automaton, only `1` will ever get printed. In the case of the right automaton, `2` will be printed when some messages arrives on B and then on A. Both automata lead to correct implementations of the semantics. However the second automata looks a better choice than the first one, since it yields more program behaviors.

4 Runtime definitions

4.1 Basics

Names are the basic values of the join-calculus, and thus any implementation of the join-calculus must supply a runtime representation for them. For instance, a name can be sent on some appropriate channel. At runtime, we must indeed send something.

However, names that are defined together in the same join definition interact when matching is tested for and performed. Moreover, by the very idea behind the join-calculus, matching is the only synchronization primitive. In other words, only names that are defined by the same join definition have some kind of interaction that is of the runtime system responsibility.

This makes possible and desirable to compile a source definition into a runtime "definition", a single vector structure that groups all the names defined in a given definition. Names must still exist as individuals, they can be represented as infix pointers into their definition (as in `join`), or as a definition pointer and an index (as in `jocaml`).

Both the `join` and `jocaml` systems implement the automata of the previous section. However, they do so in quite different manners. The former focuses on minimizing runtime testing, while the latter involves a systematic runtime testing of the current status at every message arrival.

4.2 Definitions in `join`

Runtime definitions are vector structures. Every name defined in a definition occupies two slots in the vector structure. The first entry holds a code pointer that stands for the name itself, while the second entry holds a pointer to a queue of pending messages, queues being organized as linked lists. Runtime definitions include additional slots that hold the values of the variables that are free in guarded processes. This technique resembles much the one used by the SML/NJ compiler [1] to represent mutually recursive functions. Message sending on name x is performed by stacking message values and then calling the code for name x. This code is retrieved by dereferencing twice the infix pointer that represents x at runtime.

However, there is a big difference between mutually recursive functions and join definitions. The code for name x is automaton code that reacts to the arrival of a new message on that name. The compiler issues various versions of name code, one per possible status of the definition that defines x. Typically, name code either saves a message into the queue for x (in the non-matching case), or retrieves messages from other queues before firing a guarded process (in the matching case). In all cases, definition status may then need an update, which is performed by updating all code entries in the definition.

4.3 Definitions in `jocaml`

In the `jocaml` system, a name is a pointer to a definition plus an index. Definitions are still vectors structures, but there is only one entry per name for message queues. Additionally, definitions hold guarded closures (i.e. guarded process code plus free variable values), a status field and a matching data structure.

Status field holds the current status of the definition as a bit-field. Each name status is encoded by a bit, using bit 1 for `N` and bit 0 for `0`, and bit position is given by name index.

Message sending is performed by calling a generic C function from the "join" library, taking message value, a definition pointer and a name index as arguments. When a message is received on name x, the bit for x is checked in the current status bit-field. If the bit is set, some messages on name x are already present. Thus, definition status does not change. Since the current status before message sending is guaranteed to be a non-matching one, the message is queued and the function is exited.

Otherwise, the current definition status is searched in the matching structure for x. This matching structure is an array of pattern encoding, guarded process index pairs. Pattern encodings are bit-fields, just like status encodings. corresponding to a join pattern containing name x, from which name x has been removed. Using a sequential search by a bitwise `and` with each pattern encoding, the current status can be identified as matching or non-matching in at most N_x tests, where N_x is the number of clauses whose pattern contains x.

If no match is found, the automaton state is updated and the message value is queued in the queue for x. Otherwise, a guarded process index has been found, and is used to retrieve the associated guarded closure. Arguments to the guarded process are extracted from the queues identified by the matching status found. Status is updated at the same moment (when a queue becomes empty a bit is erased). Finally the guarded process is fired.

Therefore, the performance of this technique much relies on fast comparisons and modifications of definition statuses. The best result is achieved when statuses are encoded by machine integers. In that case, the number of names that a definition can define is limited by the integer size of the hoisting Objective Caml system (which typically is 31 or 63 bits). If this is not considered enough, then statuses have to be encoded using several integers or one string. Both kinds of status encodings can be mixed, using integers for small definitions and strings for larger ones. However, in the current `jocaml` system, we use a single integer to hold status, and a technique (described in section 6) is used to associate several channels to a same bit in the status bit-field.

5 The pragmatics of compilation

This section is dedicated to optimizations that are first pertinent for the `join` technique and that are performed by the current version of the `join` compiler.

We first introduce optimizations that improve the runtime behavior of programs, both in speed and dynamic memory usage. Then, we show how to reduce emitted code size. We focus on optimizing definitions written in object-oriented style, as described in the tutorial [8]. As this programming style proved quite frequent, it is normal for us compiler writers to concentrate our efforts on such definitions.

In this style, a definition is an objet. Object state is encoded by asynchronous *state names*, while synchronous *methods* access or modify object state. For instance, given one state name S and n methods m_1, m_2,... m_n taken in that order, we get:

```
let create(x_0) =
  let S(x) | m_1() = P_1(x)
  and S(x) | m_2() = P_2(x)
    ....
  and S(x) | m_n() = P_n(x) in
  S(x_0) | reply m_1,m_2,...,m_n to create
;;
```

The synchronous call `create(v)` creates a new object (i.e. a new S, m_1, m_2,... m_n definition) and then sends back a n-tuple of its methods. Moreover, this call initializes object state with the value v.

5.1 Refined status

As a working example of an object-style definition, consider the following adder:

```
let create(x_0) =
  let S(x) | get() = S(x) | reply x to get
  and S(x) | add(y) = S(x+y) | reply to add in
  S(x_0) | reply get,add to create
;;
```

The adder has one state name S and two methods *get* and *add*. We then try to figure out some "normal" runtime behavior for it. As the initial `S(x_0)` is forked as soon as the adder definition has been created, a highly likely initial situation is that there is one message pending on S and none on the other names. Later on, as some external agent invokes the *get* or *add* method, the message pending on S is consumed and the appropriate guarded process is fired. Either process quickly sends a message on S. Thus, a likely behavior is for the queue of S to alternate between being empty and holding one element, the queue being empty for short periods. By some aspects of the compilation of "`|`" and of our scheduling policy that we will not examine here, this behavior is almost certain.

As a matter of fact, this "normal" life cycle involves a blatant waste of memory, queue elements (cons cells) are allocated and deallocated in the general dynamic fashion, while the runtime usage of these cells would allow a more efficient policy. It is more clever not to allocate a cell for the only message pending on S and to use the queue entry attributed to S in the runtime definition as a placeholder. On the status side, this new situation is rendered by a new "1" status. Hence, S now possesses a three valued status: `0` (no message), `1` (one message in the queue slot) or `N` (some messages organized in a linked list). Thus, assuming for the time being, that there may be an arbitrary number of messages pending on S, the adder compiles

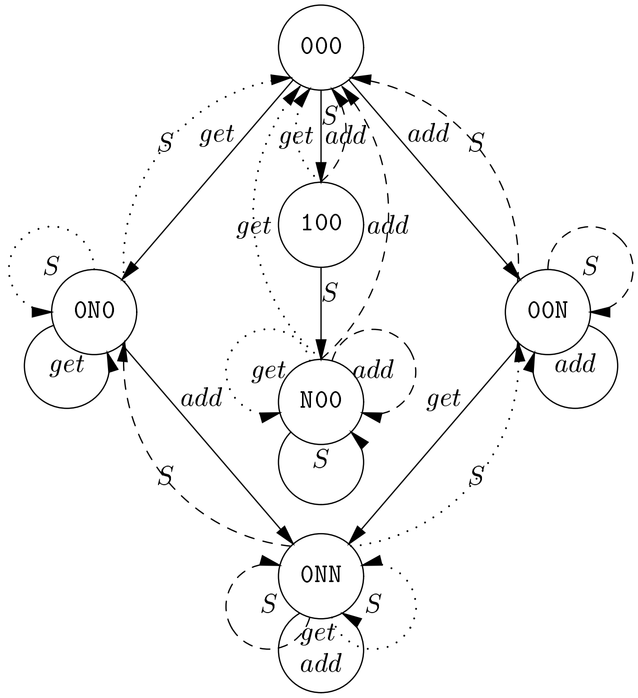


Figure 2: Full automaton for the adder

into the automaton of figure 2 (adder names are taken in the order S, *get*, *add*). This new automaton can be seen as an evolution of the A, B, C automaton of figure 1, with a slight change in channel names.

Using the status `1` not only spares memory, it also avoids some of the runtime tests that compute post-matching status. Basically, when a matching consumes the sole message pending on a name with status `1`, then the automaton already knows that this name queue is empty. For instance, when the automaton of figure 2 is in the `100` status and that a message arrive on either one of the two methods, then the appropriate process is fired and status goes back to `000` without any runtime test. By contrast, when the automaton is in the `00N` status and that a message arrive on S, the second guarded process is fired immediately, but a test on *add* queue is then performed : if this queue is now empty then status goes back to `000`, otherwise status remains `00N`. Receiving a message on S when status is `0NN` is a bit more complicated. First, the automaton chooses to consume a message pending on either one of the two methods and to fire the appropriate process (figure 2 does not specify this choice). Then, the queue of the selected method has to be tested, in order to determine post-matching status.

Status `1` is easy to implement using the `join` compilation technique. The compiler issues different method codes for `100` and `N00`, and different codes can find S argument at different places. Implementing status `1` in `jocaml` would be more tricky, since the encoding of states using bit-patterns would be far less straightforward than with `0/N` statuses only. As a consequence, the `jocaml` compiler does not perform the space optimization described in this section.

5.2 Taking advantage of semantical analysis

The automaton of figure 2 has a `N00` status, to reflect the case when there are two messages or more pending on S. However, one quite easily sees that that status `N00` is useless. First, as S does not escape from the scope of its definition, message sending on S is performed at three places only : once initially (by `S(x_0)`) and once in each guarded process. Thus, there is one message pending on S initially. A single message pending on S is consumed by any match and the process fired on that occasion is the only one to send one message on S. Therefore, there cannot be two messages or more pending on S. As a consequence the full automaton can be simplified by suppressing the `N00` node and every edge that starts from it or leads to it.

In particular, there is no more S-labeled edge starting from node `100`. In the `join` implementation this means that the code entry for S needs not be updated when going from status `000` to `100`. This entry is simply left as it is. Symmetrically, the code entry for S will not have to be restored when status goes back to `000` after matching.

Another important usage of semantical analysis is determining which names are state names. For a given definition, the output of the analyzer is a status set $\mathcal{S}$, which is a safe approximation of the actual runtime statuses of that definition. State names are the asynchronous names such that all statuses in $\mathcal{S}$ give them the status `0` or `1`.

The current `join` compiler includes a rudimentary name usage analyzer, which suffices for object definitions given in the style of the S, m_1, m_2, …, m_n definitions , where all state variables are asynchronous and do not escape from the scope of their definition.

An promising alternative would be to design an ad hoc syntax for distributed objects, or, and this would be more ambitious, a full object-oriented join-calculus. Then, the state variables of object-definitions would be apparent directly from user programs.

5.3 Avoiding status space explosion

Consider any definition that defines n names. Ignoring `1` statuses, the size of the status space of a given definition already is 2^n. The size of the non-matching status space is thus bounded by 2^n. As a full automaton for this definition has one state per non-matching status, status space size explosion would be a real nuisance in the case of the `join` compiler. In particular, there are n times the number of non-matching statuses automaton code entries to create.

Unfortunately the exponential upper bound is reached by practical programs, as demonstrated by the general object-oriented definition given at the beginning of this section 5. In that case, all definition statuses such that S has the `0` status are non-matching. In such a situation, using runtime testing, as `jocaml` does, is not that much a penalty, when compared to code size explosion.

We thus introduce dynamic behavior in the automata. We do so on a name per name basis : the status of state names will be encoded by automata states as before, whereas method statuses will now be explicitly checked at runtime. Thus, we introduce "`?`", a new status, which means that nothing is known about the number of messages pending on a name. Additionally, we state that all methods will have the `?` status, as soon as there is one message or more pending on any of the methods.

This technique can be seen as merging some states of the full automaton compiled by considering complete status information into new states with `?` statuses in them.

For instance, in the adder example, we get the automaton of figure 3, where the three statuses `0N0`, `0NN` and `00N` of figure 2 merge into the new status `0??`. (Note that we also take advantage of name usage analysis to delete status `N00`.)

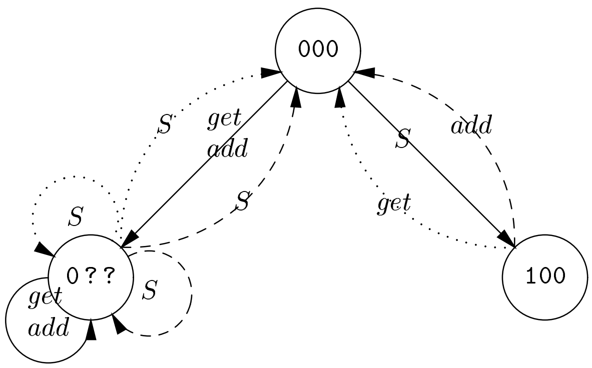


Figure 3: Final automaton for the adder

Information on where runtime testing has to be performed can be inferred from the diagram of figure 3. For instance, assume that current status is `0??` and that a message arrives on S. Since there is at least one message pending on a method, a matching will occur. Tests are needed though, before matching to determine the matching clause, and after matching to determine post-matching status. Abstractly, the first series of tests changes the `?` statuses in either `0` or `N` statuses, while the second series checks if there are still messages pending on names with `?` status. We are still investigating on how to organize these tests efficiently without producing too much code (see [2, 10] for a discussion of the size of such code in the context of compiling ML pattern-matching).

By contrast, when status is `100` and that a message arrives on *get* or *add*, then the corresponding matching is known immediately and the message pending on S is consumed. Then, the queue for S is known to be empty and status can be restored to `000` with no runtime testing at all. As message arrival order is likely to be first one message on S and then one message on *get* or *add* the final automaton of figure 3 responds efficiently to more frequent case, still being able to respond to less frequent cases (for instance, two messages on methods may arrive in a row). Furthermore, when trouble is over, the automaton has status `000` and is thus ready for the normal case. In this example, a penalty in code size is paid for improving code speed in the frequent, "normal" case, whereas this penalty is avoided in non-frequent cases, which are treated by less efficient code.

We introduced a `?` status on a name per name basis. However, there are other choices possible : *a priori*, there are many ways to merge full automata states into final automata states. However, if one really wants to avoid status space explosion the final automaton should be constructed directly, without first constructing the full automaton. Adopting our per name `?` status makes this direct construction possible. Additionally, the `?` status can be used by the simple static analyzer as a status for the names it cannot trace (e.g. names that escape their definition scope). This dramatically decreases the size of analyzer output and its running time.

6 Optimizing further

We here describe a simple transformation on join definitions, which does not rely on a full semantical analysis (such as name usage analysis), but only on a simple, local, syntactical analysis of join-patterns.

Let us take a simple example :

```
let S(x) | m_1(y) = P_1(x,y)
and S(x) | m_2(y) = P_2(x,y)
  ....
and S(x) | m_n(y) = P_n(x,y)
;;
```

In this example, a match occurs only when there are messages pending both on S and on one of the methods m_1, m_2,... Thus, from the synchronization point of view, all the methods are equivalent. And indeed, we can regroup them into one, single method channel m by transforming this join definition into :

```
let S(x) | m(p,y) = P[p] (x,y);;
let m_1(y) = m(1,y);;
let m_2(y) = m(2,y);;
 ....
let m_n(y) = m(n,y);;
```

Where P is the vector of processes $[P_1, P_2, ..., P_n]$.

Methods m_1 m_2,... are now simple wrappers. Method m_i now calls m with an additional i argument, which basically is the index of P_i in array P. At this point, we must emphasize that we describe this technique as a source to source transformation only for clarity. However, the produced source code may not be correct with respect to the join type system, when the types of methods are different. Anyhow, this optimization is implemented using ad hoc mechanisms, this both improves efficiency and solves the typing problem.

Currently, this optimization is performed by the `jocaml` compiler. This leads to a new interpretation of name indexes by the "join" library. The least significant bits in name indexes still encode names that actually take part to synchronization (here S and m), while their most significant bits (which were previously unused) now encode the extra i argument. This yields two benefits. First, the number of status checks decreases, as the number of matching statuses decreases. Second, the number of channels that can be defined in one definition can now exceed the hosting system integer size, provided some names can be grouped together for synchronization.

In the `join` compiler, this technique might be used to reduce automata size, since it lowers the number of non-matching statuses, by reducing the number of synchronizing names. Code entries for methods m_1, m_2,... would still be contained in the definition structure, they would only stack the index of the process to fire, and then call the code for method m. Moreover, they do not need to be updated after each transition of the automaton.

Finally, this technique can also be applied to more complex synchronizations. Given a definition that defines names x_1, x_2, ..., x_n, using patterns J_1, J_2, ... J_m. We say that two names are equivalent, when swapping them in the patterns yields the same set of patterns. We then replace equivalent names by a single name, plus an index.

Consider the following definition

```
let S_1(x) | m_1(y) = P_1(x,y)
and S_1(x) | m_2(y) = P_2(x,y)
and S_2(x) | m_1(y) = P_3(x,y)
and S_2(x) | m_2(y) = P_4(x,y)
;;
```

Then the set of defined names $\{S_1, S_2, m_1, m_2\}$ can be partitioned into $\{S_1, S_2\}$ and $\{m_1, m_2\}$. Then, the above program can be transformed into:

```
let S(p,x) | m(q,y) = P[p,q] (x,y);;
let m_1(y) = m(1,y);;
let m_2(y) = m(2,y);;
let S_1(y) = S(1,y);;
let S_2(y) = S(2,y);;
```

(with $P[1,1] = P_1$, $P[1,2] = P_2$, $P[2,1] = P_3$ and $P[2,2] = P_4$)

7 Conclusion and future work

In join-calculus, a name definition, all receptors on that name and all possible synchronizations on that name are grouped altogether in a single join definition. This enables the compilation of synchronization between concurrent or even distributed processes, using finite-state automata. In the distributed case, a message transport phase to the machine that currently hosts the join definition (and hence the automaton) is first performed. This strengthens our point of view that the join-calculus is the core of a distributed programming language that can be compiled in practice, mainly because it restricts reception on a channel to statically known parts of the program. The same argument applied to à la ML polymorphic typing in [7].

Taking a few benchmarks as a set of sensible join programs, both the `join` and the `jocaml` pattern matching compilation schemes prove adequate. In particular, none of the two schemes falls into the pitfall associated to the compilation technique used.

In the `join` case, one can be afraid of code size, the technique exposed in section 5.3 successfully avoids code size explosion in practical cases. The `jocaml` technique appears expensive in runtime checks and thus *a priori* produces slow code. We choose such a scheme of implementing automata using generic code, because it can be implemented simply by adding code to the Objective Caml bytecode interpreter. Using bytecode specialized for automata manipulation would have implied more important modifications of the Objective Caml bytecode interpreter. Moreover, the `jocaml` system runs faster than the `join` system, even for pure join programs, showing that its weaker compilation of join definitions is more than compensated by its total integration in the Objective Caml system.

Comparison with the Bologna implementation [12] of the join-calculus is also instructive. This system also produces bytecode, which is interpreted by a C program. It proves faster than `join` and slower that `jocaml` on most of the examples. Taking a glance at the Bologna source code reveals that it uses a scheme very similar to the one of `jocaml`, with two slight differences. First, status is systematically encoded as an array of integers. Second when a message arrives on a name x with an empty queue, all patterns are tested (whereas in `jocaml` only the patterns that contain x are tested).

Performance of a given join system depends on many factors. In particular, scheduling policy and message queue management have a dramatic impact on it. Furthermore, a policy that gives good results on one benchmark can be defeated by another. For these reasons, we cannot tell which compilation technique is the best by comparing different implementations.

Clearly, we now need to integrate all our compilation techniques in the same system, in order to compare them more thoroughly and to experiment further. However, the definition of reactivity status and the automata of section 3 provide a sound basis for these future investigations. Apart from future language development and fully implementing the failure semantics of the join-calculus, we also plan to investigate more on the implementation of threads, attempting to minimize thread suspension and creation.

References

[1] A. W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[2] L. Augustsson. Compiling pattern matching. In *Proc. of the ACM Functional Programming and Computer Architecture Conference*, 1985.

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

[4] F. L. Fessant. The jocaml system prototype. Software and documentation available on the Web, `http ://pauillac.inria.fr/jocaml/`, 1998.

[5] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, 1996.

[6] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, 1996. LNCS 1119.

[7] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit Typing à la ML for the join-calculus. In *Proc. of the 1997 8th International Conference on Concurrency Theory.* Springer-Verlag, 1997.

[8] C. Fournet and L. Maranget. The join system. Software and documentation available on the Web, `http ://pauillac.inria.fr/join/`, 1997.

[9] X. Leroy, J. Vouillon, and D. Doligez. The Objective Caml system. Software and documentation available on the Web, `http ://pauillac.inria.fr/ocaml/`, 1996.

[10] L. Maranget. Compiling lazy pattern matching. In *Proc. of the 1992 conference on Lisp and Functional Programming.* ACM Press, 1992.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes : part 1. Research Report ECS-LFCS-89-85, University of Edinburgh, 1989.

[12] L. Padovani. The bologna join system. Software and documentation (in italian) available on the Web, `ftp ://ftp.cs.unibo.it/pub/asperti/jcb01.tar.gz`, 1997.

[13] M. Shapiro, P. Dickman, and D. Plainfossé. SSP chains : Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, INRIA, Nov. 1992. Also available as Broadcast Technical Report #1.