Inheritance in the Join Calculus

C. Fournet[*] C. Laneve[†] L. Maranget[‡] D. Rémy[‡]

June 9, 2000

Abstract

We propose an object-oriented calculus with internal concurrency and class-based inheritance that is built upon the join calculus. Method calls, locks, and states are handled in a uniform manner, using labeled messages. Classes are partial message definitions that can be combined and transformed. We design operators for behavioral and synchronization inheritance. We also give a type system that statically enforces basic safety properties. Our model is compatible with the JoCaml implementation of the join calculus.

1 Introduction

Object-oriented programming has long been praised as favoring abstraction, incremental development, and code reuse. Objects can be created by instantiating definition patterns called *classes*, and in turn complex classes can be built from simpler ones. To make this approach effective, the assembly of classes should rely on a small set of operators with a clear semantics and should support modular proof techniques. In a concurrency setting, such promises can be rather hard to achieve.

The design and the implementation of concurrent object-oriented languages, *e.g.* [2, 22, 1, 4], has recently prompted the investigation of the theoretical foundations of concurrent objects. Several works provide encodings of objects in process calculi [21, 20, 13, 6] or, alternatively, supplement objects with concurrent primitives [17, 3, 12]. Those works promote a unified framework for reasoning about objects and processes, but they do not address the composition of object definitions or its typechecking.

In this work, we model concurrent objects in a simple process calculus—a variant of the *join calculus* [8, 7], we design operators for behavioral and synchronization inheritance, and we give a type system that statically enforces basic safety properties.

The join calculus is a simple name-passing calculus, related to the pi calculus but with a functional flavor. It is the core of a distributed programming language, currently implemented as an extension of ML [9, 14]. In contrast with more traditional process calculi, communication channels are statically defined in the join calculus: when channels are created, their definition provides a set of *reaction rules* that specify, once for all, how messages sent on these names will

[*]Microsoft Research, 1 Guildhall Street, Cambridge, U.K.

[†]Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy

[‡]INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex France.

be synchronized and processed. Although the join calculus does not have a primitive notion of object, definitions encapsulate the details of synchronization, much as concurrent objects.

Applying the well-known objects-as-records paradigm to the join calculus, we obtain a simple language of objects with asynchronous message passing. Method calls, locks, and states are handled in a uniform manner, using labeled messages. There is no primitive notion of functions, calling sequences, or threads (they can all be encoded using continuation messages). Our language—the *objective join calculus*—allows fine-grain internal concurrency, as each object may send and receive several messages in parallel.

For every object of our language, message synchronization is defined and compiled as a whole. This allows an efficient compilation of message delivery into automata [15] and simplifies reasoning on objects. However, the static definition of behavior can be overly restrictive for the programmer. This suggests some compile-time mechanism for assembling partial definitions. To this end, we promote partial definitions into classes. Classes can be combined and transformed to form new classes. They can also be closed to create objects.

The class language is layered on top of the core objective calculus, with a semantics that reduces classes into plain object definitions. We thus retain strong static properties for all objects at run-time. Some operators are imported from sequential languages and adapted to a concurrent setting. For instance, multiple inheritance is expressed as a disjunction of join definitions, but some disjunctions have no counterpart in a sequential language. In addition, we propose a new operator, called *selective refinement*. Selective refinement applies to a parent class, and rewrites the parent reaction rules according to their synchronization patterns. Selective refinement treats synchronization concretely, but it handles the parent processes abstractly. Our approach is compatible with the JoCaml implementation of the join calculus [14], which already singles out synchronization patterns using concrete compile-time representation, and, on the contrary, compiles behaviors into functional closures.

Our design of the class language follows from common programming patterns in the join calculus. We motivate it further by coding some standard problematic examples that mix synchronization and inheritance.

We finally equip the language with a polymorphic type system in the style of [10]. (We introduce the type system at the end of the paper only, to separate design issues and typing issues.) In addition to basic safety properties, the type system enforces privacy. Indeed, the untyped objective join calculus lacks expressiveness as regards encapsulation[1]. In order to restrict access to the internal state of objects, labels are partitioned into two sets, *public* and *private*. Then, the typing system guarantees that private labels are accessed only from the body of a class used to create the object. The correctness of the typing system is expressed for an operational semantics supplemented with privacy information.

The paper is organized as follows. In section 2, we present the objective join calculus and develop a few examples. In section 3, we supplement the language with classes and give their semantics. In section 4, we survey the type system. In section 5, we discuss related works and possible extensions. In appendix A, we provide more involved examples of inheritance and concurrency. In appendix B, we define the typing system and state its correctness. The proofs appear in the long version of this paper [11].

[1]In the plain join calculus, this problem is less acute: for a given definition, each entry point is passed as a separate name, so lexical scoping on private names provides some privacy; on the other hand, large tuples of public names must be passed instead of single objects.

2 The objective join calculus

In this section, we focus on the core calculus dealing with objects. We illustrate the operations of the calculus, then we define its syntax and semantics.

2.1 Getting started

The basic operation of our calculus is asynchronous message passing in object style. For instance, the process $out.print\_int(n)$ sends a message with label $print\_int$ and with content n to an object named out, meant to print integers on the terminal.

Accordingly, the definition of an object describes how messages received on some labels can trigger processes. For example,

$$\mathsf{obj}\ continuation = reply(x) \triangleright out.print\_int(x)$$

defines an object that reacts to messages on $reply$ by printing their content on the terminal. Another example is the rendez-vous, or synchronous buffer:

$$\mathsf{obj}\ sbuffer = get(r)\ \&\ set(a,s) \triangleright r.reply(a)\ \&\ s.reply()$$

The object $sbuffer$ has two labels get and set; it reacts to the *simultaneous presence* of one message on each of these labels by passing a message to the continuation r, with label $reply$ and contents a, and passing an empty message to s. (Object r may be the previously-defined $continuation$; object s is another continuation taking no argument on $reply$.) As regards the syntax, concurrent execution and message synchronization are expressed using the same infix operator &. Also, the calculus is polyadic, *i.e.*, messages carry tuples of values.

Some labels may convey messages representing the internal state of an object, rather than an external method call. This is the case of label $Some$ in the following unbounded, unordered, asynchronous buffer:

$$\begin{array}{l}\mathsf{obj}\ abuffer = \mathsf{self}(z)\\ \quad set(a,r\ ) \triangleright\ r.reply()\ \&\ z.Some(a)\\ \mathsf{or}\ get(r)\ \&\ Some(a) \triangleright r.reply\ (a)\end{array}$$

The object $abuffer$ can react in two different ways: a message (a, r) on set may be consumed by storing the value a in a self-inflicted message on $Some$; alternatively, a message on get and a message on $Some$ may be jointly consumed, and then the value stored on $Some$ is sent to the continuation received on get. The indirection through $Some$ makes $abuffer$ behave asynchronously: messages on set are never blocked, even if no message is ever sent on get. As regards the syntax, the prefix $\mathsf{self}(z)$ explicitly binds the name z to the defined object.

In the example above, the messages on label $Some$ encode the state of $abuffer$. The following definition illustrates a tighter management of state that implements a one-place buffer:

$$\begin{array}{l}\mathsf{obj}\ buffer = \mathsf{self}(z)\\ \quad set(a,r)\ \&\ None() \triangleright r.reply()\ \&\ z.Some(a)\\ \mathsf{or}\ get(r)\ \ \&\ Some(a) \triangleright r.reply(a)\ \&\ z.None()\\ \mathsf{init}\ \ buffer.None()\end{array}$$

Such a buffer can either be empty or contain one element. The state is encoded as a message pending on $None$ or $Some$, respectively. Object $buffer$ is created empty, by sending a first message on $None$ in the (optional) init part of the obj construct. As opposed to $abuffer$ above, a set message is blocked when the buffer is not empty.

To keep the *buffer* object consistent, there should be a single message pending on either *None* or *Some*. This invariant holds as long as external users cannot send messages on these labels directly. In appendix B, we will describe a refined semantics and a type system that distinguishes private labels such as *None* and *Some* from public labels, and we will restrict access to private labels. In the examples, private labels conventionally bear an initial capital letter.

Once private labels are hidden, each variant of our buffer provides the same interface to the outside world (two methods labeled *get* and *set*) but their concurrent behaviors are very different.

2.2 Syntax

We use two disjoint countable sets of identifiers for object names $x, z, u \in \mathcal{O}$ and for labels $\ell \in \mathcal{L}$. Tuples of names are written ${x_i}^{i \in I}$ or simply $\widetilde{x}$. The grammar of the *objective join calculus* (without classes) is given in Figure 1; it has syntactic categories for processes P, definitions D, and patterns M. We abbreviate $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ by omitting $\mathsf{self}(z)$ when z does not occur free in D and omitting $\mathsf{init}\ P$ when P is $\mathsf{0}$.

Figure 1: Syntax for the core object calculus

$P ::=$		**Processes**
	$\mathsf{0}$	null process
	$x.M$	message sending
	$P_1\ \&\ P_2$	parallel composition
	$\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$	object definition
$D ::=$		**Definitions**
	$M \triangleright P$	reaction rule
	$D_1\ \mathsf{or}\ D_2$	disjunction
$M ::=$		**Patterns**
	$\ell(\widetilde{u})$	message
	$M_1\ \&\ M_2$	synchronization

A reaction rule $M \triangleright P$ associates a pattern M with a guarded process P. Every message pattern $\ell(\widetilde{u})$ in M binds the names $\widetilde{u}$ with scope P. The object names occurring in M are called *received names*, in notation $rn(M)$. We require that every pattern M guarding a reaction rule be linear, that is, each label and each name appears at most once in M. This will be enforced by typing.

In addition, an object definition $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$ binds two names x and z to D. The scope of x is the processes P_1 and P_2; the scope of z is every guarded process in D. Free names in processes and definitions are defined according to these binders. Terms are taken modulo renaming of bound names (or α-conversion).

2.3 Chemical semantics

The operational semantics is defined in terms of a *reflexive chemical abstract machine* [8]. Each rewrite rule applies to configurations of objects and processes, called *chemical solutions*. A solution $\mathcal{D} \Vdash \mathcal{P}$ consists of a set of named object definitions $\mathcal{D}$ and of a multiset of processes $\mathcal{P}$ running in parallel. We write $x.D$ for a named definition in $\mathcal{D}$, and always assume that there is at most one definition for x in $\mathcal{D}$. We use two families of rewrite rules between solutions: *structural rules* $\equiv$ represent the syntactical rearrangement of terms; *reduction rules* $\longrightarrow$ represent the basic computation steps for objects.

Figure 2: Chemical semantics

$$\begin{array}{lll} \textsc{Par} & \textsc{Nil} & \textsc{Obj} \\ \Vdash P \ \& \ Q \ \equiv \ \Vdash P, \ Q & \Vdash \mathsf{0} \ \equiv \ \Vdash & \Vdash \mathsf{obj}\ x = \mathsf{self}(x)\ D\ \mathsf{init}\ P\ \mathsf{in}\ Q \ \equiv \ x.D \Vdash P, \ Q \end{array}$$

$$\begin{array}{ll} \textsc{Join} & \textsc{Red} \\ \Vdash x.(M \ \& \ M') \ \equiv \ \Vdash x.M \ , \ x.M' & x.[M \triangleright P] \Vdash x.M\sigma \ \longrightarrow \ x.[M \triangleright P] \Vdash P\sigma \end{array}$$

The rules for the objective join calculus are given in Figure 2; they have the following side conditions: for Obj, that x is a fresh name in the solution; for Red, that $x.[M \triangleright P]$ abbreviates a named definition $x.D$ in which D contains the reaction rule $M \triangleright P$ and that σ is a substitution with domain $rn(M)$. $M\sigma$ and $P\sigma$ denote the results of applying the substitution σ to M and P, respectively.

The rules Par and Nil make parallel composition of processes associative and commutative, with unit $\mathsf{0}$. The rule Obj describes the introduction of an object; its side condition enforces a lexical scoping discipline for object names. (Preliminary α-conversion may be required to pick a fresh name x.) The rule Join gathers messages sent to the same object. The reduction rule Red states how messages can be jointly consumed and replaced by a copy of a guarded process, in which the contents of these messages are substituted for the formal parameters appearing in the pattern.

As usual with chemical semantics, each rule mentions only the components that participate to the rewriting, but the rewriting applies to every chemical solution that contains them. More explicitly, every rewrite rule of Figure 2 comes with a simple context rule. For rules Obj and Red, for instance, we use

$$\textsc{Chemistry-Obj}\ \frac{\Vdash P \equiv x.D \Vdash \mathcal{P} \qquad x \notin \mathit{fn}(\mathcal{D}' \Vdash \mathcal{P}')}{\mathcal{D}' \Vdash P, \mathcal{P}' \equiv \mathcal{D}', x.D \Vdash \mathcal{P}, \mathcal{P}'} \qquad \textsc{Chemistry-Red}\ \frac{x.D \Vdash P \longrightarrow x.D \Vdash P'}{\mathcal{D}, x.D \Vdash P, \mathcal{P} \longrightarrow \mathcal{D}, x.D \Vdash P', \mathcal{P}}$$

where the set $\mathit{fn}(\mathcal{D} \Vdash \mathcal{P})$ collects the free names in solution, *i.e.*, $\mathit{fn}(\mathcal{D} \Vdash \mathcal{P}) = (\bigcup_{x.D \in \mathcal{D}} \{x\} \cup \mathit{fn}(D)) \cup \bigcup_{P \in \mathcal{P}} \mathit{fn}(P)$. The side conditions prevent name capture. (The context rules for Par, Join and Nil are similar to Chemistry-Red.)

The objective join calculus is a variant of the plain join calculus [8]. The join calculus can be encoded in the objective calculus by passing single-label forwarder objects instead of channel names. Conversely, objects can be passed as records of channel names in a join calculus extended with primitive records.

3 Inheritance and concurrency

We now extend the calculus of concurrent objects with classes and inheritance. The behavior of objects in the join calculus is statically defined: once an object is created, it cannot be extended with new labels or with new reaction rules synchronizing existing labels. Instead, we provide this flexibility at the level of classes. Our operators on classes can express various object paradigms, such as method overriding (with late binding) or method extension. As regards concurrency, these operators are also suitable to define synchronization policies in a modular manner.

3.1 Deriving a concurrent class

We introduce the syntax for classes in a series of simple examples. We begin with a class *buffer* behaving as the one-place buffer of Section 2.1:

$$\begin{array}{l} \mathsf{class}\ \mathit{buffer} = \mathsf{self}(z) \\ \quad \mathit{get}(r)\ \&\ \mathit{Some}(a) \triangleright r.\mathit{reply}(a)\ \&\ z.\mathit{None}() \\ \mathsf{or}\ \mathit{set}(a,r)\ \&\ \mathit{None}() \triangleright r.\mathit{reply}()\ \&\ z.\mathit{Some}(a) \end{array}$$

The class *buffer* can be used to create objects:

$$\mathsf{obj}\ b = \mathit{buffer}\ \mathsf{init}\ b.\mathit{None}()$$

Assume that, for debugging purposes, we want to log the buffer content on the terminal. Our first solution uses an explicit *log* label.

$$\begin{array}{l} \mathsf{class}\ \mathit{logged\_buffer} = \mathsf{self}(z) \\ \quad \mathit{buffer} \\ \mathsf{or}\ \mathit{log}()\ \&\ \mathit{Some}(a) \triangleright \mathit{out.print\_int}(a)\ \&\ z.\mathit{Some}(a) \\ \mathsf{or}\ \mathit{log}()\ \&\ \mathit{None}() \triangleright \mathit{out.print\_string}(\text{"None"})\ \&\ z.\mathit{None}() \end{array}$$

The class body above is a disjunction of an inherited class and of additional reaction rules. The intended meaning of disjunction is that reaction rules are cumulated, yielding competing behaviors for messages on labels that appear in several disjuncts. The order of the disjuncts does not matter. The programmer that writes *logged_buffer* must have some knowledge of the parent class *buffer*, namely the use of private labels *Some* and *None* for representing the state.

Another, perhaps more useful, kind of debugging information is a synchronized log of all messages consumed on *set*. This is done by selecting the patterns in which *set* occurs and adding a printing message to the corresponding guarded processes:

$$\begin{array}{l} \mathsf{class}\ \mathit{logged\_buffer\_bis} = \\ \quad \mathsf{match}\ \mathit{buffer}\ \mathsf{with} \\ \qquad \mathit{set}(a,r) \Rightarrow \mathit{set}(a,r) \triangleright \mathit{out.print\_int}(a) \\ \quad \mathsf{end} \end{array}$$

The match construct can be understood by analogy with pattern matching *à la* ML, applied to the reaction rules of the parent class. Here, any reaction rule from the parent *buffer* whose pattern contains the label *set* is replaced in the derived *logged_buffer_bis* by a rule with the same pattern (*set* appears on both sides of $\Rightarrow$) and with the original guarded process in parallel with a printing message (the parent action and the & are left implicit in syntax). Any other parent rule remains unchanged. Hence, the above definition behaves as the following definition:

$$\begin{array}{l} \mathsf{class}\ \mathit{logged\_buffer\_bis} = \\ \quad \mathit{get}(r)\ \&\ \mathit{Some}(a) \triangleright r.\mathit{reply}(a)\ \&\ z.\mathit{None}() \\ \mathsf{or}\ \mathit{set}(a,r)\ \&\ \mathit{None}() \triangleright r.\mathit{reply}()\ \&\ z.\mathit{Some}(a)\ \&\ \mathit{out.print\_int}(a) \end{array}$$

Yet another kind of debugging information is a log of *set* attempts:

$$\begin{array}{l}\mathsf{class}\ \mathit{logged\_buffer\_ter} = \mathsf{self}(z)\\ \quad \mathsf{match}\ \mathit{buffer}\ \mathsf{with}\\ \quad\quad \mathit{set}(a,r) \Rightarrow \mathit{Parent\_set}(a,r) \triangleright 0\\ \quad \mathsf{end}\\ \mathsf{or}\ \mathit{set}(a,r) \triangleright \mathit{out.print\_int}(a)\ \&\ z.\mathit{Parent\_set}(a,r)\end{array}$$

In this case, the match construct performs a renaming of *set* into *Parent_set* in the patterns of selected rules, without affecting their guarded processes. The net effect is similar to parent method overriding, with the new *set* calling the parent one and a late-binding semantics for guarded processes.

The examples above illustrate that the very idea of class refinement is less abstract in a concurrent setting than in a sequential one. In the first *logged_buffer* example, logging the buffer state requires knowledge of how this state is encoded; otherwise, some states might be forgotten or, worse, logging might lead the buffer into deadlock. The other two examples expose another subtlety: in a sequential language, the distinction between logging *set* attempts and *set* successes is irrelevant. Thinking in terms of sequential object invocations, one may be unaware of the concurrent behavior of the object, and thus write *logged_buffer_ter* instead of *logged_buffer_bis*.

3.2 Syntax

The language with classes extends the core calculus of section 2; its grammar is given in Figure 3. We refer to sections 2.1, 3.1, and to appendix A for explanations and further examples. Classes are taken up to the associative-commutative laws for disjunction or. We use two additional sets of identifiers for class names $c \in \mathcal{C}$ and for sets of labels $L \in 2^{\mathcal{L}}$. Such sets L are used to represent abstract classes that declare all labels in L but do not define them.

Join patterns J generalize the syntactic category of patterns M given in Figure 1 with an operator or that represents alternative synchronization patterns. Join patterns are taken up to simple equivalence laws: & and or are associative-commutative, and & distributes over or. Hence, every join pattern J can be written as a non-empty alternative of patterns $\mathsf{or}_{i\in I} M_i$.

Selection patterns K are either join patterns or the empty pattern 0. Their normal forms are of the form above, except that I can be empty. We sometimes use the notation K_1 & K_2 for decomposing patterns M; we then use the additional law $K\ \&\ 0 = K$. We always assume that patterns J and K meet the following well-formed conditions:

1. All conjuncts M_i in the normal form of K are linear (as defined in section 2.2) and bind the same names. By extension, we say that K binds the names $rn(M_i)$ bound in every M_i, and write $rn(K)$ for these names.

2. In a refinement clause $K_1 \Rightarrow K_2 \triangleright P$, the pattern K_1 is either M or 0, and the pattern K_2 binds at least the names of K_1 ($rn(K_1) \subseteq rn(K_2)$).

As described in section 2.2, binders for object names include object definitions (binding the defined object to name x and self name z) and patterns (binding the received names). In a reaction rule $J \triangleright P$, the join pattern J binds $rn(J)$ with scope P. In a refinement clause $K_1 \Rightarrow K_2 \triangleright P$, the selection pattern K_1 binds $rn(K_1)$ with scope K_2 and P; the modification pattern K_2 binds $rn(K_2) \setminus rn(K_1)$ with scope P. Finally, the self renaming $\mathsf{self}(x)\, C$ binds the

Figure 3: Syntax for classes

$P ::=$		**Processes**
	$\ldots$	(as in figure 1)
	$\mathsf{obj}\ x = \mathsf{self}(z)\ C\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$	object definition
	$\mathsf{class}\ c = \mathsf{self}(z)\ C\ \mathsf{in}\ P$	class definition
$C ::=$		**Classes**
	c	class variable
	L	abstract class
	$J \triangleright P$	reaction rule
	$C_1\ \mathsf{or}\ C_2$	disjunction
	$\mathsf{self}(x)\ C$	self binding
	$\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$	selective refinement
$S ::=$		**Refinement clauses**
	$(K_1 \Rightarrow K_2 \triangleright P) \mid S$	refinement sequence
	$\emptyset$	empty refinement
$J ::=$		**Join patterns**
	$\ell(\widetilde{u})$	message
	$J_1\ \&\ J_2$	synchronization
	$J_1\ \mathsf{or}\ J_2$	alternative
$K ::=$		**Selection patterns**
	0	empty pattern
	J	join pattern

object name x with scope C. Class definitions $\mathsf{class}\ c = C\ \mathsf{in}\ P$ are the only binders for class names c, with scope P. Processes, classes, and reaction rules are taken up to α-conversion.

Labels don't have scopes. Join patterns J declare the labels appearing in their message, *i.e.* $dl(\ell(\widetilde{u})) = \{\ell\}$, $dl(J_1\ \&\ J_2) = dl(J_1) \cup dl(J_2)$, and $dl(J_1\ \mathsf{or}\ J_2) = dl(J_1) \cup dl(J_2)$. Classes C declare the labels of their reaction rules, *i.e.* $dl(J \triangleright P) = dl(J)$ and $dl(C_1\ \mathsf{or}\ C_2) = dl(C_1) \cup dl(C_2)$. Abstract classes trivially declare their labels, *i.e.* $dl(L) = L$. Finally, selective refinements declare labels appearing either in the parent class or in a refinement clause, *i.e.* $dl(\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}) = dl(C) \cup dl(S)$ where $dl(S)$ is defined as $dl(\emptyset) = \emptyset$ and $dl((K \Rightarrow K' \triangleright P) \mid S)) = dl(K) \cup dl(K') \cup dl(S)$.

3.3 Class rewriting semantics

The semantics of classes is given by reduction to the core calculus without classes. Reduction rules are described below, first for class bodies, then for processes.

Reductions for class bodies. Class reduction $\stackrel{x}{\longmapsto}$ carries an object name that stands for the (topmost) name of self. This reduction is defined in Figure 4.

Figure 4: Semantics for classes

Rules for classes

Class-Abstract
$$C \;\mathsf{or}\; \emptyset \xmapsto{x} C$$

Abstract-Cut
$$\frac{L \cap dl(C) \neq \emptyset}{C \;\mathsf{or}\; L \xmapsto{x} C \;\mathsf{or}\; (L \setminus dl(C))}$$

Self
$$\frac{x \notin fn(\mathsf{self}(z)\, C)}{\mathsf{self}(z)\, C \xmapsto{x} C\{x/z\}}$$

Or-Pat
$$J \;\mathsf{or}\; J' \triangleright P \xmapsto{x} J \triangleright P \;\mathsf{or}\; J' \triangleright P$$

Match
$$\frac{C \;\mathsf{with}\; S \;\longmapsto\; C' \qquad fresh(S) \subseteq dl(C')}{\mathsf{match}\; C \;\mathsf{with}\; S \;\mathsf{end} \;\xmapsto{x}\; C'}$$

Class-Context
$$\frac{C \xmapsto{x} C'}{E[C] \xmapsto{x} E[C']}$$

Rules for filters

Filter-Apply
$$K_1 \,\&\, K \triangleright P \;\mathsf{with}\; K_1 \Rightarrow K_2 \triangleright Q \mid S \;\longmapsto\; K_2 \,\&\, K \triangleright P \,\&\, Q \;\mathsf{or}\; dl(K_1) \setminus dl(K_2)$$

Filter-Next
$$\frac{dl(K_1) \not\subseteq dl(M) \qquad M \triangleright P \;\mathsf{with}\; S \longmapsto C'}{M \triangleright P \;\mathsf{with}\; K_1 \Rightarrow K_2 \triangleright Q \mid S \;\longmapsto\; C'}$$

Filter-Abstract
$$L \;\mathsf{with}\; S \;\longmapsto\; L$$

Filter-End
$$C \;\mathsf{with}\; \emptyset \;\longmapsto\; C$$

Filter-Or
$$\frac{C_1 \;\mathsf{with}\; S \;\longmapsto\; C_1' \qquad C_2 \;\mathsf{with}\; S \;\longmapsto\; C_2'}{(C_1 \;\mathsf{or}\; C_2) \;\mathsf{with}\; S \;\longmapsto\; C_1' \;\mathsf{or}\; C_2'}$$

Evaluation contexts for class bodies

$$E[C] \;::=\; C \;\mid\; \mathsf{match}\; E[C] \;\mathsf{with}\; S \;\mathsf{end} \;\mid\; E[C] \;\mathsf{or}\; C' \;\mid\; C' \;\mathsf{or}\; E[C]$$

Auxiliary definitions of $fresh(\cdot)$

$$fresh(|^{i\in I}(K_i \Rightarrow K_i' \triangleright P_i)) = \bigcup_{i\in I} dl(K_i') \setminus dl(K_i)$$

The first two rules simplify abstract classes: rule CLASS-ABSTRACT discards empty abstract classes; rule ABSTRACT-CUT discards labels that are declared elsewhere. Rule SELF removes inner bindings for the name of self. Rule OR-PAT lifts or constructs from patterns to classes.

Selective refinement is the most complex operation. The expression match C with S end evaluates by means of an auxiliary reduction $\longmapsto$ that computes *filters* of the form C with S. The auxiliary relation applies only to disjunctions of reaction rules and abstract classes (more complex classes must be reduced beforehand by rule CLASS-CONTEXT). Every reaction rule $J \triangleright P$ in C is rewritten by the leftmost clause of S that filters J, that is, whose selection pattern K is a sub-pattern of J (rules FILTER-NEXT and FILTER-APPLY). If no clause of S filters J, then the reaction rule is left unchanged (rule FILTER-END). Every abstract class L in C is left unchanged (rule FILTER-ABSTRACT).

Technically, the pattern K_2 & K introduced by rule FILTER-APPLY is well formed only if (1) K_2 and K are not both empty, and (2) K_2 and K declare disjoint sets of labels. Condition (1) is simply enforced by a syntactic restriction on refinement clauses: if K_2 is empty, then K_1 must also be empty. This constraint does not hamper expressiveness. Condition (2) will be checked by the type system, using the types of the class being refined and of the selector.

When we have C with $S \longmapsto C'$, some labels in the reactions rules of C may not appear anymore in those of C'. More specifically, rule FILTER-APPLY removes all the labels in $dl(K_1) \setminus dl(K_2)$ from the matched reaction rule K_1 & $K \triangleright P$. However, lost labels are kept as abstract labels. Our type system (defined in appendix B) computes an approximation of abstract names. Therefore, we statically ensure that lost labels are redefined (by providing suitable reaction rules) before an instance of the class can be taken.

Conversely, some labels in the reaction rules of C' may not appear in those of C. More specifically, rule FILTER-APPLY introduces fresh labels $dl(K_2) \setminus dl(K_1)$ into the refined reaction rule K_2 & $K \triangleright P$ & Q. The set $fresh(S)$ in figure 4 collects the potential fresh labels of S and is statically computable. However, some clauses of S may not be used, hence some label of $fresh(S)$ may not actually appear in the patterns of C'. This situation often corresponds to a programming error; its detection usually requires knowledge of the synchronization patterns of C. To retain expressiveness of selective refinement and keep types simple, we detect this situation dynamically. This is expressed by the premise $fresh(S) \subseteq dl(C')$ of rule Match.

With the rules in Figure 4, our class rewriting semantics is deterministic. Moreover, the final class is a definition in the core object calculus, provided that it does not contain free class names, undefined declared labels, or unresolved refinements. The first two cases are ruled out by the type system in appendix B. The last case occurs only when the rule MATCH does not apply because of the condition $fresh(S) \subseteq dl(C')$.

Reductions for processes. Reduction $\longmapsto$ for processes is defined in Figure 5. An irreducible processs either belongs to the core objective join-calculus, or it contains an irreducible class construct, as described above.

3.4 Implementation issues

Filter rewriting hinders the abstract compilation of patterns, because the compiler must access the patterns of the parent class to build the rules of the derived class. By contrast, the actions of reaction rules can be compiled into closures abstracted with respect to their free names (which include formal message parameters), since the match construct never alters them. This approach

Figure 5: Reduction for processes

CLASS-RED

$$\frac{C \xmapsto{z} C'}{\mathsf{obj}\ x = \mathsf{self}(z)\, C\ \mathsf{init}\ P\ \mathsf{in}\ P' \longmapsto \mathsf{obj}\ x = \mathsf{self}(z)\, C'\ \mathsf{init}\ P\ \mathsf{in}\ P'}$$

CLASS-VAR

$$\mathsf{class}\ c = \mathsf{self}(z)\, C\ \mathsf{in}\ P \longmapsto P\{\mathsf{self}(z)\, C/_c\}$$

PROCESS-CONTEXT

$$\frac{P \longmapsto P'}{E[P] \longmapsto E[P']}$$

where $E[\cdot]$ is any process context (*i.e.*, classes under guards are also reducible.)

is compatible with the JoCaml implementation, which represents join patterns as vectors of bits; the control of the synchronization of messages and the activation of guarded processes is then realized by interpreting these bit vectors at runtime [15].

The patterns in the core objective join calculus do not contain or constructs, and this explains the presence of rule OR-PAT in the class semantics. Instead, we could supplement the object calculus with or-patterns. We view this as an optimization issue. Perhaps more importantly, the unsharing of reaction rules with common sub-patterns performed by rule OR-PAT does not mean that guarded processes are duplicated by the compiler, which would preclude separate compilation. Since guarded processes are compiled as closures, duplicating P in the semantics means duplicating an indirection to the closure that implements P.

4 Static semantics

Our type system for the objective join calculus appears in appendix B. It is inspired by the type systems of the plain join calculus for concurrency and synchronization [10] and of Objective Caml for the class-layer [19]. Class and object types are records of labels paired with their types. Types are polymorphic: they include variables, whose scope is controlled at the class level.

As regards polymorphism, the generalization rule (rules OBJECT and CLASS) corresponds to the one implemented in JoCaml [14]. It is more expressive than the generalization rule used in [10]. It seems equivalent to the generalization rule independently proposed in [5], even though the two formalizations differ. Our type system has been carefully designed so as to allow type inference. We conjecture that well-typed programs have principal types. However, this remains to be checked formally.

The type system prevents runtime errors, such as ill-typed messages and messages-not-understood. The type system also enforces object encapsulation. To this aim, labels are partitioned into *public* and *private* labels. Private labels should be visible only from their defining objects (including their sub-objects). We show that a well-typed program never violate privacy, *i.e.*, never access a private label outside of its visibility scope. In order to express our result, we refine the chemical semantics of section 2.3 to keep track of access privileges for every message along the reduction.

5 Related and future works

There are many works on supplementing object calculi with concurrent primitives [17, 3, 12], or on supplementing process calculi with objects (usually by the mean of an encoding in the original calculus) [21, 20, 13, 6]. Our work follows the latter tradition. However, to our knowledge, it is the only one to address safe object composition in a process calculus.

In [18], Martin Odersky proposes an object-oriented extension of a language based on the join calculus. His proposal amounts to adding some record structure to join definitions, as we do in section 2. However, Odersky does not consider the problem of inheritance and refinement of synchronization. A technical difference is that Odersky's calculus is designed with pattern matching on values.

Since our type system abstracts from the shape of synchronization patterns in classes, it is blind to a number of relevant properties of concurrency, such as the presence of race conditions or deadlock freedom. The design of a sophisticated analyzer that is sensitive to synchronizations is a promising research direction.

6 Conclusions

We have proposed a simple, object-based variant of the join calculus. Every object is defined as a fixed set of reaction rules that describe its synchronization behavior. The expressiveness of the language is significantly increased by adding classes—a form of open definitions that can be incrementally assembled before object instantiation. Thereby, we partially recover the ability of the pi calculus to dynamically define the receptive behavior for messages. Our layered design confines this capability to classes. From a programming-language viewpoint, this seems a good compromise between flexibility and simplicity of the model. Indeed, our proposal still enables efficient compilation of synchronization and type inference.

References

[1] G. Agha, P. Wegner, and A. Yonezawa. *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.

[2] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[3] P. D. Blasio and K. Fisher. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, LNCS 1119, pages 406–421, 1996.

[4] L. Cardelli. Obliq A language with distributed scope. SRC Research Report 122, Digital Equipment, June 1994.

[5] G. Chen, M. Odersky, C. Zenger, and M. Zenger. A functional view of join. Technical Report ACRC-99-016, University of South Australia, 1999.

[6] S. Dal-Zilio. Quiet and bouncing objects: Two migration abstractions in a simple distributed blue calculus. In H. Hüttel and U. Nestmann, editors, *Proceedings of the Worshop on Semantics of Objects as Proceedings (SOAP '98)*, pages 35–42, June 1998.

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

[8] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385, Jan. 1996.

[9] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, LNCS 1119, pages 406–421, 1996.

[10] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory*, LNCS 1243, pages 196–212, 1997. Available electronically at `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/typing-join.ps.gz`.

[11] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join-calculus. Full version. Available electronically at `http://cristal.inria.fr/~remy/work/ojoin.ps.gz`, June 2000.

[12] A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *entcs*, Nice, France, Sept. 1998. To appear.

[13] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. To appear, June 1998.

[14] F. Le Fessant. The JoCAML system prototype. Software and documentation available from `http://pauillac.inria.fr/jocaml`, 1998.

[15] F. Le Fessant and L. Maranget. Compiling join-patterns. *Electronic Notes in Computer Science*, 16(2), 1998.

[16] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. The MIT Press, 1993.

[17] O. Nierstrasz. Towards an object calculus. In O. N. M. Tokoro and P. Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 1–20, 1992.

[18] M. Odersky. Functional nets. In *European Symposium on Programming 2000*, LNCS. Springer Verlag, 2000.

[19] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension to ML. In *Proceedings of POPL '97*, Jan. 1997. Available electronically at `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!tapos98.ps.gz`.

[20] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998.

[21] D. J. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.

[22] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices*, 21(11):258–268, Nov. 1986. Proceedings of OOPSLA '86.

A Inheritance anomaly

As remarked by many authors, the classical point of view on class abstraction—method names and signatures are known, method bodies are abstract—does not mix well with concurrency. More specifically, concurrency and class-based inheritance are not orthogonal. This well-known problem is often referred to as the *inheritance anomaly*. Unfortunately, inheritance anomaly is not defined formally, but by means of examples as in [16] who define three kind of inheritance anomalies.

These examples are small but useful in practice programs. They demonstrate that extending a base class by a new capacity has a impact on the (desirable) concurrent behavior of the capacities that are inherited from the base class. Straightforward extensions to concurrency of sequential languages, such as implementing synchronization in method bodies or providing simple locking policies (cf. `synchronized` from Java) prove unsufficient here. The former because synchronization code is not accessible, the latter because the provided synchronization policies are not expressive enough. [16] (partially) solve inheritance anomalies by making concrete some parts of the parent class (such as "concurrency-control"). It is to be noticed that they do so by considering a new extension for each anomaly.

Our approach is different: starting from a concurrent language we are more concerned with the expressive power of our inheritance operators. Solving the three categories of inheritance anomaly, as we do, appears to be a valuable test.

To this aim, we consider the same running example as Matsuoka and Yonezawa: a FIFO buffer with two methods *put* and *get* to store and retrieve items. We also adopt their taxonomy of inheritance anomaly: inheritance induces desirable modifications of "acceptable states" [of objects], and a solution is a way to express these modifications.

We extend our programming language with booleans and integers, with usual primitive operations. Arrays are created by $\mathsf{create}(n)$, that gives an uninitialized array of size n. The size of array A is retrieved by $A.size$. Finally $A[i]{\leftarrow}v$ is array A where the i-th item has been replaced with v.

$$
\begin{array}{l}
\mathsf{class}\ \mathit{buff} = \mathsf{self}\ (z) \\
\quad \mathit{put}(v,r)\ \&\ (\mathit{Empty}(A,\ i,\ n)\ \mathsf{or}\ \mathit{Some}(A,\ i,\ n)) \triangleright \\
\qquad r.\mathit{reply}()\ \&\ z.\mathit{Check}(A[(i{+}n)\ \mathsf{mod}\ A.\mathit{size}] \leftarrow v,\ i,\ n{+}1) \\
\mathsf{or}\ \mathit{get}(r)\ \&\ (\mathit{Full}(A,\ i,\ n)\ \mathsf{or}\ \mathit{Some}(A,\ i,\ n)) \triangleright \\
\qquad r.\mathit{reply}(A[i])\ \&\ z.\mathit{Check}(A,\ (i{+}1)\ \mathsf{mod}\ A.\mathit{size},\ n{-}1) \\
\mathsf{or}\ \mathit{Check}(A,i,n) \triangleright \\
\qquad \mathsf{if}\ n = A.\mathit{size}\ \mathsf{then}\ z.\mathit{Full}(A,\ i,\ A.\mathit{size}) \\
\qquad \mathsf{else}\ \ \mathsf{if}\ n = 0\ \mathsf{then}\ z.\mathit{Empty}(A,\ 0,\ 0) \\
\qquad \mathsf{else}\ z.\mathit{Some}(A,\ i,\ n) \\
\mathsf{or}\ \mathit{Init}(\mathit{size}\ ) \triangleright\ z.\mathit{Empty}(\mathsf{create}(\mathit{size}),0,0)
\end{array}
$$

The state of the buffer is encoded as a message with label *Empty*, *Some*, or *Full*. The buffer may react to messages on *put* when non-full, and to messages on *get* when non-empty; this is expressed in a concise manner using the or operator in patterns. Once a request is accepted, the state of the buffer is recomputed by sending an internal message on *Check*. As *Check* appears alone in a join pattern, message sending on *Check* acts like a function call, which can be in-lined by an optimizing compiler.

Partitioning of acceptable states. The class *buff2* supplements *buff* with a new method *get2* that atomically retrieves two items from the buffer. For simplicity, we assume here $size > 2$.

Since *get2* succeeds when the buffer contains two elements or more, the buffer state needs to be refined. Furthermore, since for instance, a successful *get2* may disable *get* or enable *put*, the addition of *get2* has an impact on the "acceptable states" of *get* and *put*, which are inherited from the parent *buff*. Therefore, label *Some* is no longer pertinent and is replaced with two labels *One* and *Many*. *One* models the state when the buffer holds exactly one item; *Many* defines a state with two items or more in the buffer.

```
class buff2 = self(z)
   get2(r) & (Full(A,i,n) or Many(A, i, n)) ▷
        r.reply(A[i], A[(i+1) mod A.size])
     & z.Check(A, (i+2) mod A.size, n−2)
or match buff with
   Some(A, i, n) ⇒ (One(A, i, n) or Many(A, i, n)) ▷ 0 end
or Some(A, i, n) ▷
     if n > 1 then z.Many(A, i, n) else z.One(A, i, n)
```

In the program above, a new method *get2* is defined, with its own synchronization condition. The new reaction rule is cumulated with those of *buff*, using a selective refinement that substitutes "*One*(...) or *Many*(...)" for every occurrence of "*Some*(...)" in a join pattern. The refinement eliminates *Some* from any inherited pattern, but it does not affect occurrences of *Some* in inherited guarded processes: the parent code is handled abstractly, so it cannot be modified. Instead, the new class provides an adapter rule that consumes any message on *Some* and issues a message on either *One* or *Many*, depending on the value of n.

History-sensitiveness for acceptable states. The class *gget_buff* alters *buff* as follows: the new method *gget* returns one item from the buffer (like *get*), except that a request on *gget* can be served only immediately after serving a request on *put*. More precisely, a *put* transition enables *gget*, while *get* and *gget* transitions disable it. This condition is reflected in the code by introducing two labels *AfterPut* and *NotAfterPut*. Then, messages on *gget* are synchronized with messages on *AfterPut*.

```
class gget_buff = self (z)
   gget(r) & AfterPut()
             & (Full(A, i, n) or Some(A, i, n)) ▷
         r.reply(A[i]) & z.NotAfterPut()
     & z.Check(A, (i+1) mod A.size, n−1)
or match buff with
   Init(size) ⇒ Init(size ) ▷  z.NotAfterPut()
 | put(v, r) ⇒
```

$$\begin{array}{l}
\quad\quad put(v, r) \ \& \ (AfterPut() \ \mathsf{or} \ NotAfterPut()) \triangleright \\
\quad\quad\quad z.AfterPut() \\
\quad | \ get(r) \Rightarrow \\
\quad\quad get(r) \ \& \ (AfterPut() \ \mathsf{or} \ NotAfterPut()) \triangleright \\
\quad\quad\quad z.NotAfterPut() \\
\mathsf{end}
\end{array}$$

The first clause in the match construct refines initialization, which now also issues a message on *NotAfterPut*. The two other clauses refine the existing methods *put* and *get*, which now consume any message on *AfterPut* or *NotAfterPut* and produce a message on *AfterPut* or *NotAfterPut*, respectively.

Modification of acceptable states. A general-purpose lock may be defined as:

$$\begin{array}{l}
\mathsf{class} \ locker = \mathsf{self} \ (z) \\
\quad suspend(r) \ \& \ Free() \triangleright r.reply() \ \& \ z.Locked() \\
\mathsf{or} \ resume(r) \ \& \ Locked() \triangleright r.reply() \ \& \ z.Free()
\end{array}$$

The class *locker* can be used to create locks, but it can also be combined with some other class c to control message processing on the labels of c. To this end, a simple disjunction of c and *locker* is not enough and some refinement of the parent c is required.

$$\begin{array}{l}
\mathsf{class} \ locked_buff = \mathsf{self} \ (z) \\
\quad locker \\
\mathsf{or} \ \mathsf{match} \ buff \ \mathsf{with} \\
\quad\quad Init(size) \Rightarrow Init(size) \triangleright z.Free() \\
\quad | \ 0 \Rightarrow Free() \triangleright z.Free() \\
\quad \mathsf{end}
\end{array}$$

The first clause in the match construct supplements the initialization of *buff* with an initial *Free* message for the lock. The second clause matches every other rule of *buff*, and requires that the refined clause consume and produce a message on *Free*. (The semantics of clause selection follows the textual priority scheme of ML pattern-matching, where a clause applies to all reaction rules that are not selected by previous clauses, and where the empty selection pattern acts as a default case.)

As a consequence of these changes, parent rules are blocked between a call to *suspend* and the next call to *resume*, and parent rules leave the state of the lock unchanged. In contrast with previous examples, the code above is quite general; it applies to any class following the same convention as *buff* for initialization.

B Types and privacy

The static semantics of our calculus is inspired by those of the core join calculus for concurrency and synchronization [10] and of Objective Caml for the class-layer [19].

We develop a polymorphic type system that prevents various dynamic errors. First, it guarantees *type safety*, that is, no label will ever carry a tuple of messages of the wrong type. Then, it avoids *message-not-understood* errors, that is, no message can be sent to an object with a label that is undefined at that object. (Of course, this does not ensure any processing of messages sent on defined labels, which may be deadlocked.) Finally, the type system checks object encapsulation.

Figure 6: Syntax for types

$\tau ::= \theta \mid [\rho]$	**Types**	$\alpha ::= \theta \mid \varrho$	**Type variables**
$\rho ::= \emptyset \mid \varrho \mid m : \widetilde{\tau}; \rho$	**Row types**	$\widetilde{\tau} ::= ({\tau_i}^{i \in I})$	**Tuple types**
$\sigma ::= \forall X.\tau$	**Type schemes**	$B ::= \emptyset \mid \ell : \widetilde{\tau}; B$	**Internal types**

For that purpose, we partition labels $\ell \in \mathcal{L}$ into *private labels* $f \in \mathcal{F}$ and *public labels* $m \in \mathcal{M}$. A message on a private label can only be sent internally, that is, from within a reaction rule of the object to which the message is sent. (A precise definition can be found in section B.3.) Conversely, a message on a public label can be sent from any context that has access to the object name.

B.1 Syntax for Types

We build types out of a countable set of variable types ranged over by θ. We also assume a countable collection of row-variables, ranged over by ϱ. The grammar for type expressions is given in figure 6. In the sequel, we write α for variables, regardless of their kinds, and γ for either object types τ or row types ρ. We write X and Y for sets of variables. We also abbreviate $\forall\emptyset.\tau$ as τ.

The operator $\ell : \_; \_$ associates to the left. We often skip the trailing $\emptyset$, *i.e.* we abbreviate $\ell_1 : \widetilde{\tau}_1; \ldots \ell_n : \widetilde{\tau}_n; \emptyset$ by $\ell_1 : \widetilde{\tau}_1; \ldots \ell_n : \widetilde{\tau}_n$; and we abstract away from the order of labels $\ell_1, \ldots$ ℓ_n. Object types $[\rho]$ are modeled as records of the types of public labels. Types B are used to describe private labels of objects and class types.

Class types have the form $\forall X.\zeta(\rho)B^V$. The two main components are B and ρ. The expression B specifies the public and private labels that are actually declared in the class. The row ρ is used to collect all the constraints on the type $[\rho]$ of self, *i.e.* an object of the class being defined. These constraints usually originate from recursive calls, but may also come by using self as argument of a message. It is possible that $[\rho]$ has more labels than actually declared in B. Those labels are undefined in the class; a class containing undefined labels cannot be instantiated into an object. Finally, the set V lists virtual labels. These labels are necessarily declared (*i.e.* in the domain of B) but not necessarily defined. Abstract labels are virtual, unless defined elsewhere.

B.2 Typing judgments

An environment A binds object names x to type schemes σ or to internal type schemes $\forall X.B$ and class names c to class type schemes. More precisely, each object x usually has two bindings in A: $x : \forall X.[\rho]$ *and* $x : \forall X.B$, where labels in the domain of B are private. The binder $x : \forall X.B$ defines the type of private labels of x.

Typing judgments are described in figure 7: The last three judgments are of the form $(\_ \vdash \_ :: Y.\_)$ where Y is a set of type variables that cannot be generalized. This constraint is due to synchronization for join patterns (*cf.* rule SYNCHRONIZATION); similar constraints are due to the mutable store in ML.

The typing rules use the following auxiliary definitions:

Figure 7: Typing Judgments

$A \vdash P$	the process P is well-typed in environment A;
$A \vdash x : \tau$	the object x has type τ in environment A;
$A \vdash x.\ell : \widetilde{\tau}$	the label ℓ conveys messages of type $\widetilde{\tau}$ for object x in environment A;
$A \vdash K :: Y.B$	the pattern K receives variables well-typed in A, joins labels in B, and prevents generalization on type variables Y.
$A \vdash C :: Y.\zeta(\rho)B^V$	the class C is well-typed in environment A, declares the labels of B (including virtual labels V, *i.e.* $V \subseteq \mathit{dom}(B)$), and prevents generalization on type variables Y.
$A \vdash S :: Y.\zeta(\rho)(B \Rightarrow B')^V$	the selection S is well-typed in environment A, refines patterns with labels in B (including virtual labels V, *i.e.* $V \subseteq \mathit{dom}(B)$) into patterns with labels in B', and prevents generalization on type variables Y.

- $\mathit{ftv}(\_)$ is the set of free type variables occurring in a type, a tuple of types, a type scheme, or a typing context.
- $B \upharpoonright \mathcal{M}$ and $B \upharpoonright \mathcal{F}$ are the restrictions of B on public labels and private labels, respectively;
- $B_1 \oplus B_2$ is the union of B_1 and B_2, with the statement that B_1 and B_2 coincide on their common labels.
- $A + A'$ is $(A \setminus \mathit{dom}(A')) \cup A'$;

Typing rules for names

Figure 8: Typing rules for names

Object-Var
$$\frac{x : \forall X.\tau \in A}{A \vdash x : \tau\{\gamma_\alpha/\alpha^{\alpha \in X}\}}$$

Private-Message
$$\frac{x : \forall X.(f : \widetilde{\tau}; B) \in A}{A \vdash x.f : \widetilde{\tau}\{\gamma_\alpha/\alpha^{\alpha \in X}\}}$$

Message
$$\frac{A \vdash x : [m : \widetilde{\tau}; \rho]}{A \vdash x.m : \widetilde{\tau}}$$

Class-Var
$$\frac{c : \forall X.\zeta(\rho)B^V \in A \qquad \mathit{ftv}(B) \setminus X \subseteq Y}{A \vdash c :: Y.(\zeta(\rho)B^V)\{\gamma_\alpha/\alpha^{\alpha \in X}\}}$$

Class and object names are typed by instantiating their types taken from the environment. For classes, variables of the class type scheme that were not polymorphic become non-generalizable in the type of the class name. Rules Private-Message and Message check well-typing of messages to labels.

Typing rules for processes

Figure 9: Typing rules for processes

$$\textsc{Null}\quad A \vdash 0$$

$$\textsc{Send}\quad \frac{A \vdash x.\ell : ({\tau_i}^{i\in I}) \qquad (A \vdash x_i : \tau_i)^{i\in I}}{A \vdash x.\ell({x_i}^{i\in I})}$$

$$\textsc{Join}\quad \frac{A \vdash x.M_1 \qquad A \vdash x.M_2}{A \vdash x.(M_1 \,\&\, M_2)}$$

$$\textsc{Par}\quad \frac{A \vdash P \qquad A \vdash Q}{A \vdash P \,\&\, Q}$$

$$\textsc{Object}\quad \frac{\begin{array}{c} A \vdash \mathsf{self}(z)\, C :: Y.\zeta(\rho)B^{\emptyset} \qquad \rho = B \upharpoonright \mathcal{M} \\ X = \mathit{ftv}(B) \setminus (\mathit{ftv}(A) \cup Y) \qquad A + x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F}) \vdash P \qquad A + x : \forall X.[\rho] \vdash Q \end{array}}{A \vdash \mathsf{obj}\ x = \mathsf{self}(z)\, C\ \mathsf{init}\ P\ \mathsf{in}\ Q}$$

$$\textsc{Class}\quad \frac{A \vdash \mathsf{self}(z)\, C :: Y.\zeta(\rho)B^{V} \qquad X = \mathit{ftv}(B) \setminus (\mathit{ftv}(A) \cup Y) \qquad A + c : \forall X.\zeta(\rho)B^{V} \vdash P}{A \vdash \mathsf{class}\ c = \mathsf{self}(z)\, C\ \mathsf{in}\ P}$$

The interesting rule is Object. The object body C is typechecked first and yields a class type $Y.\zeta(\rho)B^{\emptyset}$. Note the constraint banning virtual labels—under-specified classes cannot be instantiated. The object type is the restriction of labels defined in B to public ones. The constraint $\rho = B \upharpoonright \mathcal{M}$ enforces the compatibility of labels that are common to the domains of ρ and B. (Alternatively, this compatibility could be enforced globally.) The process Q is typed in the environment extended with the object bound to a generalized $[\rho]$. The generalization considers those type variables that appear in neither Y nor A. The process P is typed as Q, except that P can access the private labels of x.

Typing rules for class bodies

Rule Reaction checks that join patterns and guarded processes agree on the typing environment extended with the received variables. Rule Self-Binding folds two bindings for self, accounting for public and private bindings, respectively. Rules Disjunction and Refinement merge virtual-label informations, as a disjunct or a parent class may effectively define a virtual label.

Rule Refinement types $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$, out of typings of C and S. It uses the auxiliary judgment for selection clauses that keeps track of labels that are defined in the left hand side and right hand side of patterns. The premise $B_2 \subseteq B_1$ checks that labels in the selection patterns are all defined in B_1, hence declared in C. On the contrary, the premise $\mathit{fresh}(S) \cap \mathit{dom}(B_1) = \emptyset$ implies that names in $\mathit{fresh}(S)$ are not already declared in C. In particular, this ensures that patterns of reaction rules resulting from a refinement are linear (see section 3.3, condition (2)).

Figure 10: Typing rules for class bodies

Reaction

$$\frac{A' \vdash J :: Y.B \qquad A + A' \vdash P \qquad dom(A') = rn(J)}{A \vdash J \triangleright P :: Y.\zeta(\rho)B^V}$$

Self-Binding

$$\frac{A + x : [\rho],\, x : (B \upharpoonright \mathcal{F}) \vdash C :: Y.\zeta(\rho)B^V}{A \vdash \mathsf{self}(x)\; C :: Y.\zeta(\rho)B^V}$$

Abstract

$$\frac{dom(B) = L}{A \vdash L :: Y.\zeta(\rho)B^L}$$

Disjunction

$$\frac{\begin{array}{c} A \vdash C_1 :: Y.\zeta(\rho)B_1^{V_1} \\ A \vdash C_2 :: Y.\zeta(\rho)B_2^{V_2} \qquad (I_k = dom(B_k) \setminus V_k)^{k=1,2} \qquad (V_1 \setminus I_2) \cup (V_2 \setminus I_1) \subseteq V \end{array}}{A \vdash C_1 \;\mathsf{or}\; C_2 :: Y.\zeta(\rho)(B_1 \oplus B_2)^V}$$

Refinement

$$\frac{\begin{array}{c} A \vdash C :: Y.\zeta(\rho)B_1^{V_1} \\ A \vdash S :: Y.\zeta(\rho)(B_2 \Rightarrow B_2')^{V_2} \qquad B_2 \subseteq B_1 \qquad fresh(S) \cap dom(B_1) = \emptyset \qquad V_1 \cup V_2 \subseteq V \end{array}}{A \vdash \mathsf{match}\; C \;\mathsf{with}\; S \;\mathsf{end} :: Y.\zeta(\rho)(B_1 \oplus B_2')^V}$$

Typing rules for selection and join patterns

Typing rules for join patterns define a typing environment for labels therein and the set of type variables that cannot be generalized. More specifically, two labels must agree on the type of their arguments if one argument can be sent along the other one in the guarded process. For instance, in the synchronous channel in section 2.1, defined by $get(r)\ \&\ set(a,s) \triangleright r.reply(a)\ \&\ s.reply()$, the types of first arguments of *get* and *set* cannot be generalized. See rule Synchronization and rule Reaction for comparison. Rules Empty-Pattern and Message-Pattern show that one can always safely add extra variables to the set of non-generalizable variables. Indeed, forcing Y to be $\emptyset$ in those two rules would be correct, but too restrictive, and subject reduction would fail.

Typing rules for refinement clauses

Refinement clauses are typed much like reaction rules and class disjuncts. Rule Modifier checks that labels in the pattern K_i' agree with those K_i of the parent class; the set of virtual labels accounts for labels potentially eliminated by the clause.

Example of typing

For example, we infer a type for the class *buffer* from section 3.1,

$$\begin{array}{ll} \mathsf{self}\,(z) & \\ & get(r)\ \ \&\ Some(a) \triangleright r.reply(a)\ \&\ z.None() \\ \mathsf{or} & set(a,r)\ \&\ None() \triangleright r.reply()\ \&\ z.Some(a) \end{array}$$

of the form $\mathsf{self}(z)\,(J_1 \triangleright P_1 \;\mathsf{or}\; J_2 \triangleright P_2)$.

Figure 11: Typing rules for patterns

$$\textsc{Empty-Pattern}\quad A \vdash \mathsf{0} :: Y.\emptyset$$

$$\textsc{Message-Pattern}\quad \frac{(A \vdash x_i :: \tau_i)^{i\in I}}{A \vdash \ell({x_i}^{i\in I}) :: Y.(\ell : ({\tau_i}^{i\in I}))}$$

$$\textsc{Synchronization}\quad \frac{A \vdash K_1 :: Y.B_1 \qquad A \vdash K_2 :: Y.B_2 \qquad dom(B_1) \cap dom(B_2) = \emptyset \qquad ftv(B_1) \cap ftv(B_2) \subseteq Y}{A \vdash K_1 \ \& \ K_2 :: Y.B_1 \oplus B_2}$$

$$\textsc{Alternative}\quad \frac{A \vdash K_1 :: Y.B_1 \qquad A \vdash K_2 :: Y.B_2 \qquad rn(K_1) = rn(K_2)}{A \vdash K_1 \ \mathsf{or}\ K_2 :: Y.B_1 \oplus B_2}$$

Figure 12: Typing rules for refinement clauses

$$\textsc{Modifier}\quad \frac{\begin{pmatrix} A'_i \vdash K'_i :: Y.B'_i \qquad A + A'_i \vdash P_i \qquad dom(A'_i) = rn(K'_i) \\ A'_i \vdash K_i :: Y.B_i \qquad ftv(B'_i) \setminus ftv(B_i) \subseteq Y \qquad dom(B_i) \setminus dom(B'_i) \subseteq V \end{pmatrix}^{i\in I}}{A \vdash |^{i\in I}(K_i \Rightarrow K'_i \triangleright P_i) :: Y.\zeta(\rho)(\oplus^{i\in I} B_i \Rightarrow \oplus^{i\in I} B'_i)^V}$$

First, let us consider the typing of pattern J_1. By rules Message-Pattern and Synchronization we have:

$$A_1 \vdash get(r) \ \& \ Some(a) :: Y.{B_1}^{\emptyset} \tag{1}$$

where set Y is left unspecified, whereas type environment A_1 and internal type B_1 are of the general forms $A_1 = (r : \theta_1, a : \theta_2)$ and $B_1 = (get : (\theta_1), Some : (\theta_2))$, reflecting label arity.

However pattern J_1 is not typed in isolation but as the pattern of a reaction rule whose guarded process P_1 includes $r.reply(a)$. Rule Reaction requires that P_1 is typed in an environment that subsumes A_1. Moreover, $r.reply(a)$ connects the types for r and a (rule Send). Thus, we get: $\theta_1 = [reply : (\theta); \varrho]$ (*i.e.* r is an object with at least a *reply* label) and $\theta_2 = \theta$. Then, the type variable θ occurs free in the types of both *Some* and *get* which are joined in the same pattern J_1. Hence, set Y in (1) should at least include θ (*cf.* rule Synchronization). Skipping some details, rule Reaction yields the following typing of $J_1 \triangleright P_1$:

$$A \vdash J_1 \triangleright P_1 :: \{\theta\}.\zeta(\rho){B_1}^{\emptyset} \tag{2}$$

where ρ is the public type for self, A is described below, and B_1 is $(get : ([reply : (\theta); \varrho]), Some : (\theta))$. Similarly, the second reaction rule is typed as:

$$A \vdash J_2 \triangleright P_2 :: \emptyset.\zeta(\rho){B_2}^{\emptyset} \tag{3}$$

where $B_2 = (set : (\theta', [reply : (); \varrho']).None : ())$.

Environment A must be the same in both (2) and (3) because these two judgments are premises of a DISJUNCTION rule:

$$A \vdash J_1 \triangleright P_1 \text{ or } J_2 \triangleright P_2 :: \{\theta\}.\zeta(\rho)B^{\emptyset} \tag{4}$$

The internal type B is $B_1 \oplus B_2$. Here, this amounts to $B_1 \cup B_2$ since patterns J_1 and J_2 have no label in common. Hence $B = (get : ([reply : (\theta); \varrho]), Some : (\theta), set : (\theta', [reply : (); \varrho']), None : ())$.

Rule SELF-BINDING implies that environment A contains two bindings for the self name z, namely, $x : [\rho]$ and $x : B_\rho$. The internal type of z, B_ρ is the restriction of B to private labels.

We can now detail the typing for P_2 (which is an hypothesis for (3)). Listing only pertinent parts of the typing environment $A + A_2$, we have:

$$\ldots a : \theta', z : (Some : (\theta), \ldots) \vdash P_2 :: \emptyset.\zeta(\rho){B_2}^{\emptyset} \tag{5}$$

Now, observe that P_2 includes the message $z.Some(a)$, which requires the types for a and for any message on $Some$ to be equal. Thus, $\theta = \theta'$.

Finally, the type for class variable *buffer* is:

$$\forall\{\varrho, \varrho'\}.\zeta(\rho)B^{\emptyset} \tag{6}$$

where B is as before (after equating θ and θ'). Observe that only ϱ and ϱ' are generalized, and not θ (*cf.* rule CLASS). As a consequence, *buffer* is not polymorphic with respect to stored values. The row type ρ is yet unconstrained, but rule OBJ (for typing object creation) would later enforce that ρ contains exactly the types of the public labels of the class.

B.3 Semantics with privacy

Our typing system carefully checks that private labels are accessed from class bodies defining them or from the initialization clauses. However, this is a static notion. So far, our semantics looses the static origin of messages. For instance, rule OBJ in the semantics of section 2.3 mixes the process from the initialization clause, which has access to private names, and the main process in the scope of the definition, which should not access private names.

In this section, we refine the chemical semantics with a notion of privacy for each runtime state. The new semantics simply keeps track of the origin of messages. Informally, private labels may be used to form messages within the static scope of the object, possibly including sub-objects.

In the refined chemical solution, every running process is thus equipped with a prefix $\psi\,\#$, in which ψ is a string of names describing the static environment of the process. Similarly, the definition for every object x is equipped with a prefix $\psi x\,\#$, in which ψ is of the form $y_1 \ldots y_n$; the prefix indicates that x is in the scope of y_1, y_2, …, and y_n. A well-formed refined chemical solution is such that all prefixes agree on object nesting, *i.e.*, if ψx and φx appear in prefixes, then $\psi = \varphi$. This property is preserved by chemical rewriting. We use the refined rules of Figure 13.

There are some side conditions: for OBJ, x' is a fresh name in the solution; for RED, $\psi x\,\#[M \triangleright P]$ abbreviates $\psi x\,\# D$ for some definition D containing the rule $M \triangleright P$ and σ is a substitution on the names bound in M.

Figure 13: Chemical semantics with privacy

PAR
$$\Vdash \psi \,\#\, (P \,\&\, Q) \;\equiv\; \Vdash \psi \,\#\, P \,,\, \psi \,\#\, Q$$

NIL
$$\Vdash \psi \,\#\, 0 \;\equiv\; \Vdash$$

JOIN
$$\Vdash \psi \,\#\, x.(M \,\&\, M') \;\equiv\; \Vdash \psi \,\#\, x.M \,,\, \psi \,\#\, x.M'$$

OBJ
$$\Vdash \psi \,\#\, \mathsf{obj}\ x = \mathsf{self}(x)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q \;\equiv\; \psi x \,\#\, D \Vdash \psi x \,\#\, P \,,\; \psi \,\#\, Q$$

PUBLIC-COMM
$$\psi x \,\#\, D \Vdash \psi' \,\#\, x.m(\widetilde{u}) \;\longrightarrow\; \psi x \,\#\, D \Vdash \psi x \,\#\, x.m(\widetilde{u})$$

PRIVATE-COMM
$$\psi x \,\#\, D \Vdash \psi x \psi' \,\#\, x.f(\widetilde{u}) \;\longrightarrow\; \psi x \,\#\, D \Vdash \psi x \,\#\, x.f(\widetilde{u})$$

RED
$$\psi x \,\#\, [M \triangleright P] \Vdash \psi x \,\#\, x.(M\sigma) \;\longrightarrow\; \psi x \,\#\, [M \triangleright P] \Vdash \psi x \,\#\, (P\sigma)$$

Except for the bookkeeping on environments, rules NIL, PAR, JOIN, OBJ, and RED are the same as in section 2. Note that RED consumes only messages with a prefix that matches the object definition, and triggers a process in the same environment as the object definition. Since messages may be sent from other objects, an intermediate *routing step* is called for. Such steps are modeled by rules PUBLIC-COMM and PRIVATE-COMM that carry messages from their emitter to their receiver.

Our privacy policy states that a message sent from object y to object x on a private label is valid inasmuch as y is a sub-object of x (*cf.* rule PRIVATE-COMM). We take the presence of non-routable messages as our definition of a privacy error. Formally:

Definition 1 (Violation of privacy) *A chemical solution* $\mathcal{D} \Vdash \mathcal{P}$ has violated privacy *when* $\mathcal{D}$ *contains a definition* $\psi x.D$ *and* $\mathcal{P}$ *contains a private message* $\psi' \,\#\, x.f(\widetilde{u})$ *while* ψ *is not a prefix of* ψ'*.*

The new semantics is a refinement of the previous one (formally, we could show that any reduction in the new semantics correspond a reduction in the old semantics). In particular, runtime errors can still be detected:

Definition 2 (Runtime-Failure) *A chemical solution* $\mathcal{D} \Vdash \mathcal{P}$ has failed *when* $\mathcal{P}$ *contains a message* $\psi \,\#\, x.\ell(\widetilde{u})$ *such that one of the following statement holds:*

1. x *is not defined in* $\mathcal{D}$*;*

2. $\psi' x \,\#\, D \in \mathcal{D}$ *and label* ℓ *is not defined in* D*;*

3. $\psi' x \,\#\, D \in \mathcal{D}$*, label* ℓ *is defined in* D *in the message pattern* $\ell(\widetilde{y})$*, and* $\widetilde{y}$ *and* $\widetilde{u}$ *have different arities.*

B.4 Type checking solutions

Typing of programs can be easily extended to chemical solutions. This is useful in our proofs because the subject reduction must be checked on the chemistry.

We introduce two typing judgments to state that the chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is well-typed in the environment A. The main judgment $A \vdash \mathcal{D} \Vdash \mathcal{P}$ holds when all object definitions and all processes are independently well-typed in A:

$$\textsc{Soup}\quad \frac{(A^{\psi} \vdash P)^{(\psi \,\#\, P) \in \mathcal{P}} \qquad (A \vdash \psi \,\#\, D)^{(\psi \,\#\, D) \in \mathcal{D}} \qquad \mathit{dom}(A) \subseteq \{x \mid (\psi x \,\#\, D) \in \mathcal{D}\}}{A \vdash \mathcal{D} \Vdash \mathcal{P}}$$

where A^{ψ} is A where the internal bindings $z : B_i$ have been removed for all names z that do not appear in ψ.

This uses an auxiliary judgment $A \vdash \psi \,\#\, D$ that states that object definition D is well-typed in A at path ψ:

$$\textsc{Multi}\quad \frac{\begin{array}{c} A^{\psi} + x : [\rho], x.(B \restriction \mathcal{F}) \vdash D :: Y.\zeta(\rho)B^{\emptyset} \\ \rho = B \restriction \mathcal{M} \qquad X = \mathit{ftv}(B) \setminus (\mathit{ftv}(A^{\psi}) \cup Y) \qquad x : \forall X.[\rho] \in A \qquad x : \forall X.(B \restriction \mathcal{F}) \in A \end{array}}{A \vdash \psi x \,\#\, D}$$

B.5 Subject reduction with privacy

We define the relation $\subset$ between chemical soups as the preservation of typing. That is, $\mathcal{D} \Vdash \mathcal{P} \subset \mathcal{D}' \Vdash \mathcal{P}'$ if whenever there exists an environment A such that $A \vdash \mathcal{D} \Vdash \mathcal{P}$ then there also exists a typing environment A' such that $A' \vdash \mathcal{D}' \Vdash \mathcal{P}'$.

B.5.1 Chemical reduction

Theorem 1 (Chemistry) *Chemical relations $\equiv$ and $\longrightarrow$ preserve typing (*i.e.* there are sub-relations of $\subset$).*

Theorem 2 (Progress) *A well-typed chemical machine cannot fail.*

(Privacy) *A well-typed chemical solution cannot violate privacy.*

B.5.2 Process Reduction

Theorem 3 (Process Reduction) *Process rewriting $\longmapsto$ preserves typing.*

Theorem 4 (Process-Progress) *A process that is well-typed in the empty environment and that cannot be rewritten either is in the core object calculus or it contains an unresolved refinement (a selection* $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$ *such that* $\mathit{fresh}(S) \not\subseteq C'$*).*