Inheritance in the Join Calculus

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

May 26, 2000

Abstract

We propose an object-oriented calculus with internal concurrency and class-based inheritance that is built upon a process calculus—the *join calculus*. In our calculus, method calls, locks, and states are handled in a uniform manner, using labeled messages. Classes are partial definitions that can be combined and transformed. We design operators for behavioral and synchronization inheritance, and we give a type system that statically enforces basic safety properties. Our approach to inheritance and synchronization is compatible with the present 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, 24, 1, 4], has recently prompted the investigation of the theoretical foundations of concurrent objects. Several works provide encodings of objects in process calculi [23, 21, 13, 6] or, alternatively, supplement objects with concurrent primitives [17, 3, 12]. Those works mostly promote a unified framework for reasoning about objects and processes; none of them address the composition of object definitions or its typechecking.

To this aim, 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.

In order to adapt the join calculus to objects, we apply the well-known objects-as-records paradigm and 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 all can be encoded using continuation messages after CPS). Our language—the *objective join calculus*—allows fine-grain internal concurrency, as each object may send and often treat 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 in the source language. This calls for some compile-time mechanism for assembling partial definitions. To this end, we promote partial definitions into *classes*. Classes can be combined and transformed to create new classes. They can also be *closed* to form objects. Thus, we retain strong static properties for all objects at run-time.

On top of the core objective join calculus, we therefore design a class language and its semantics for reducing classes into plain object definitions. Some operations are imported from sequential languages and adapted to a concurrent setting; for instance, we subsume multiple inheritance as a disjunction of join definitions with an additive behavior that has 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 pattern. Thus, selective refinement treats synchronization concretely, but it handles the parent processes abstractly. Our approach to inheritance and synchronization is compatible with the JoCaml implementation of the join calculus [14], which already singles out synchronization patterns into a concrete compile-time representation, and, on the contrary, compiles behaviors into functional closures.

Our design of the class language, in general, and of selective refinement, in particular, 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 also 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. 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 only accessed internally, from the body of the class of which the object is an instance. 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 a reduction semantics for the class language, which can be seen as a compilation down to the basic language. In appendix A, we provide more involved examples of inheritance and concurrency. In appendix 4, we provide a static semantics for our calculus; we state the results in section B. We refer the reader to [11] for detailed proofs. In section 5, we discuss related works and possible

[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 provide privacy; on the other hand, large tuples of public names must be explicitly passed instead of single objects.

extensions.

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 contents n to an object named *out*, meant to print integers n on the terminal.

Accordingly, the definition of an object describes how messages on particular labels can be used to trigger particular processes of the object. For example,

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

defines an object that reacts to messages on *reply* by printing their contents on the terminal. Another example is the rendez-vous (aka synchronous buffer):

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

The object *sbuffer* has two labels *get* and *set*; it reacts to the *simultaneous presence* of messages on each of these labels by passing the value a to the continuation object r. (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:

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

The object *abuffer* can react in two different ways: a message (a, r) on *set* is consumed by storing the value a in a self-inflicted message on *Some*; a message on *get* is consumed together with one message on *Some* and returns a stored value. The indirection through *Some* makes *abuffer* behave asynchronously. As regards the syntax, self binding is made explicit by the prefix $\mathsf{self}(z)$.

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:

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

Such a buffer can either be empty or contain one element. Each 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.

To keep a *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 Section 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 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, (a *set* label and a *get* label), 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 joins of messages M. We abbreviate $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ 0\ \mathsf{in}\ Q$ into $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{in}\ Q$ and we usually omit $\mathsf{self}(z)$ when z does not occur free in D.

Figure 1: Syntax for the core object calculus

$P ::=$		**Processes**
	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
$M ::=$		**Joins of messages**
	$\ell(\widetilde{u})$	message
	$M_1\ \&\ M_2$	synchronization

In a reaction rule $M \triangleright P$, M is the pattern while P is the guarded process. Every message pattern $\ell(\widetilde{u})$ in M binds the formal parameters $\widetilde{u}$, with scope P. *Formal parameters*, or *received names*, of a pattern are defined by $rv(\ell({u_i}^{i \in I})) = \{{u_i}^{i \in I}\}$ and $rv(M_1\ \&\ M_2) = rv(M_1) \cup rv(M_2)$. We require that every pattern M guarding a reaction rule be linear, that is, each label and each formal parameter appears at most once in M.

In addition to message patterns, object definitions $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$ bind two object 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 definitions and processes are defined according to these binding constructs.

Labels don't have scopes. We define the set $dl(M)$ of labels declared in M as the set of labels appearing in the message patterns in M. By extension, we define $dl(D)$ as the set of labels declared by the reaction rules of D.

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 $\mathcal{D} \Vdash \mathcal{P}$ consist 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 rearrangements of terms; *reduction rules* $\longrightarrow$ represent basic computation steps for objects, consuming messages in the solution and triggering guarded processes.

Figure 2: Chemical semantics

$$\textsc{Par} \quad \Vdash P \ \&\ Q \ \equiv\ \Vdash P,\ Q \qquad \textsc{Nil} \quad \Vdash \mathsf{0} \ \equiv\ \Vdash \qquad \textsc{Join} \quad \Vdash x.(M \ \&\ M') \ \equiv\ \Vdash x.M\ ,\ x.M'$$

$$\textsc{Obj} \quad \Vdash \mathsf{obj}\ x = \mathsf{self}(x)\ D\ \mathsf{init}\ P\ \mathsf{in}\ Q \ \equiv\ x.D \Vdash P,\ Q$$

$$\textsc{Red} \quad x.[M \triangleright P] \Vdash x.M\sigma \ \longrightarrow\ x.[M \triangleright P] \Vdash P\sigma$$

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 $rv(M)$. ($M\sigma$ 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 Join gathers messages sent to the same object. 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 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 context rule. For the structural rule Obj, for instance, we use

$$\textsc{Chemistry-Obj} \quad \frac{\mathcal{D} \Vdash \mathcal{P}_1 \equiv \mathcal{D}, x.D \Vdash \mathcal{P}_2 \qquad x \notin \mathit{fv}(\mathcal{D}') \cup \mathit{fv}(\mathcal{P}')}{\mathcal{D} \cup \mathcal{D}' \Vdash \mathcal{P}_1 \cup \mathcal{P}' \equiv \mathcal{D} \cup \mathcal{D}', x.D \Vdash \mathcal{P}_2 \cup \mathcal{P}'}$$

where $\mathit{fv}(\mathcal{D})$ and $\mathit{fv}(\mathcal{P})$ stand for the free names of objects definitions $\mathcal{D}$ and processes $\mathcal{P}$ (with $\mathit{fv}(x.D) = \{x\} \cup \mathit{fv}(D)$).

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 patterns on existing labels. Instead, we provide flexibility at the level of classes. In contrast with objects, classes can be extended, and objects are finally created by instantiating classes. At the base level, classes are definitions D, *i.e.*, disjunctions of reaction rules. Additionally, classes can be named, they can be combined using the or operator to gather their reaction rules, and they can incorporate parent classes in a controlled manner using *selective refinement*. More precisely, selective refinement allows the modification of the parent rules when they are inherited, and this modification can be selectively applied to the rules consuming messages that match a given pattern.

Our operators on classes can express various object-oriented paradigms, such as method overriding (with late binding) or method extension (methods are just labels in our calculus). 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. In order to write *logged_buffer*, the programmer 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:

```
class logged_buffer_bis =
  match buffer with
    set(a,r) → set(a,r) ▷ out.print_int(a)
  end
```

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:

```
class logged_buffer_bis =
   get(r)  & Some(a) ▷ r.reply(a) & z.None()
or set(a,r) & None() ▷ r.reply() & z.Some(a)
                             & out.print_int(a)
```

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

```
class logged_buffer_ter = self(z)
  match buffer with
    set(a,r) → Parent_set(a,r) ▷ 0
  end
or set(a,r) ▷ out.print_int(a) & z.Parent_set(a,r)
```

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 much like parent method overriding, with the new *set* calling the parent one, and a late-binding semantics.

Apart from introducing the syntax for classes presented in the next section, 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*.

Arguably, the compilation of classes is more involved than in the sequential case. Patterns cannot be compiled in a very abstract way, since the compiler must be given access to 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 formal message parameters, since the match construct never alters them. This approach is compatible with the JoCaml implementation, which represents join patterns as vectors of bits and uses generic code to interpret these vectors at run-time, in order to control the synchronization of messages and the activation of guarded processes [15].

3.2 Syntax

The language with classes extends the core calculus of section 2. We refer to the informal presentations of Sections 2.1, 3.1, and A for explanations and examples. The formal syntax of Figure 3 uses two additional sets of identifiers for class names $c \in \mathcal{C}$ and for sets of labels $L \in 2^{\mathcal{L}}$.

A class L declares all labels in L but does not defined them. It is usually known as an abstract class. Operator or on classes is commutative and associative, and classes are considered up to this simple equivalence law.

Figure 3: Syntax for classes

$P ::=$	**Processes**
$\quad \ldots$	as in figure 1
$\quad \mathsf{obj}\ x = \mathsf{self}(z)\, C\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$	object definition
$\quad \mathsf{class}\ c = \mathsf{self}(z)\, C\ \mathsf{in}\ P$	class definition
$C ::=$	**Class bodies**
$\quad c$	class variable
$\quad L$	abstract class
$\quad J \triangleright P$	reaction rule
$\quad C_1\ \mathsf{or}\ C_2$	disjunction
$\quad \mathsf{self}(x)\, C$	self binding
$\quad \mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$	selective refinement
$J ::=$	**Join Patterns**
$\quad \ell(\widetilde{u})$	message
$\quad J_1\ \&\ J_2$	synchronization
$\quad J_1\ \mathsf{or}\ J_2$	alternative
$S ::=$	**Refinement clauses**
$\quad (K_1 \Rightarrow K_2 \triangleright P) \mid S$	modifier
$\quad \emptyset$	no change
$K ::=$	**Selection patterns**
$\quad 0$	empty pattern
$\quad J$	join pattern

Join patterns J generalize the syntactic category of join of messages M of Figure 1 with an operator or for representing alternative synchronization patterns. In the whole paper, 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 joins of messages $\mathsf{or}_{i\in I}\, M_i$. Selection patterns K are either join patterns or the empty patterns, written 0, and obtained for $I = \emptyset$ in the normal form above; although the pattern expression $K_1\ \&\ K_2$ never occurs in programs, we may use this notation, with additional laws $K\ \&\ 0 = K$ and $0\ \&\ K = K$. Remark that the function $dl(\cdot)$ is not defined on class bodies containing class variables or selective refinements. In the whole paper, we carefully apply this function to class bodies in its domain.

We always assume that patterns J and K meet the following well-formed conditions:

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

2. In a modifier $K_1 \Rightarrow K_2 \triangleright P$, the pattern K_1 is either M or $\mathsf{0}$, and the pattern K_2 binds at least the formal parameters of K_1 ($rv(K_1) \subseteq rv(K_2)$).

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 their formal parameters). In the reaction rule $J \triangleright P$, the pattern J binds $rv(J)$ with scope P. In the modifier $K_1 \Rightarrow K_2 \triangleright P$, the selection pattern K_1 binds $rv(K_1)$ with scope K_2 and P; the modification pattern K_2 binds $rv(K_2) \setminus rv(K_1)$ with scope P. In addition, the self renaming $\mathsf{self}(x)\, C$ binds the object name x with scope C. Class definitions $\mathsf{class}\ c = C\ \mathsf{in}\ P$ are the binders for class names c, with scope P. Processes, class bodies, modifiers, and reaction rules are taken up to alpha-conversion on bound names.

Patterns J declare the labels of all the message patterns in them, *i.e.* we first state $dl(\ell(\widetilde{u})) = \{\ell\}$ and $dl(J_1\ \&\ J_2) = dl(J_1) \cup dl(J_2)$ (as we did for patterns M in section 2.2), and then $dl(J_1\ \mathsf{or}\ J_2) = dl(J_1) \cup dl(J_2)$. Classes C declare the labels declared by its reaction rules, *i.e.* we first state $dl(J \triangleright P) = dl(J)$ and $dl(C_1\ \mathsf{or}\ C_2) = dl(C_1) \cup dl(C_2)$. Finally, we state $dl(\mathsf{0}) = \emptyset$ and, since the purpose of abstract classes L is to declare labels, we state $dl(L) = L$. Declared labels of other class constructs are undefined.

3.3 Class rewriting semantics

The semantics of classes is given as a translation to the core calculus without classes. The main intent of the reduction semantics that follows is to represent operations on the run-time representation of classes. However, part of it, such as safety with respect to abstract labels or assigning slots in the run-time representation of classes can be performed as compile-time and link-time operations. Reduction rules are given in two groups: those for class bodies and those for processes.

Reductions for class bodies. Class reductions $\xmapsto{x}$ carry an object name that stands for the name of self. They are defined in Figure 4, as the closure of axioms $\xmapsto{x}$ for class evaluation contexts E.

Selective refinement is the most complex operation. The refinement $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$ evaluates into a class body by means of an auxiliary reduction $\longmapsto$ that simplifies *filters* C with S. (For the moment don't care about the premise $\mathit{fresh}(S) \subseteq dl(C')$.)

Rules of $\longmapsto$ are organized so that effective refinement is performed only on simple reaction rules (*i.e.* reaction rules of the core calculus as described in figure 1). Therefore, let us consider $M \triangleright P$ (M does not contain any or-pattern) and $K_1 \Rightarrow K_2 \triangleright Q$. The modifier *selects* the reaction rule if $dl(K_1)$ is a subset of $dl(M)$, then M can be written as $K_1\ \&\ K$ (rule Filter-Success), where K is either $\mathsf{0}$ or a join of messages. Once selected, the reaction rule is rewritten by replacing K_1 by K_2 in its pattern on the one hand, and by supplementing Q to its guarded process on the other hand. As a consequence of a successful refinement, some labels declared by K_1 may not be present in K_2. To keep track of these labels, rule Filter-Success makes names $dl(K_1) \setminus dl(K_2)$ abstract in the refined class. These names will be traced by the type system so as to ensure that they are redefined before an instance of the class can be taken.

Observe that the pattern $K_2\ \&\ K$ is not necessarily a well-formed join pattern J. More specifically, we have to check the following two extra conditions: (1) patterns K_2 and K cannot both be empty, (2) patterns K_2 and K declare different labels. Condition (1) can be enforced by the restriction on the syntax of modifiers that whenever K_1 is not empty (which allows K to be empty), then K_2 is also not empty. Such a light constraint does not hamper expressiveness. By contrast, condition (2) will be checked while typing classes, at a time when the labels of the class being refined are known.

In examining refinement of simple reaction rules, it remains to see what happens when a modifier does not select a reaction rule. Multiple refinement clauses separated by "|" are successively tried, subsequent clauses being considered only for reaction rules that are not selected by any former clause (rule FILTER-NEXT). By default, rule FILTER-END keeps unchanged any reaction rule that is not selected. Observe that, as a consequence of these two rules, our semantics for selective refinement is deterministic.

Let us now focus on the premise $\mathit{fresh}(S) \subseteq \mathit{dl}(C')$ of rule MATCH. Observe that the definition of $\mathit{dl}(C')$ requires C' to be a disjunct of reaction rules and abstract classes (see the end of section 3.2). Indeed, our rules produce a class C' of that restricted form. More importantly, computing $\mathit{dl}(C')$ needs the knowledge of all the patterns in C. Indeed, $\mathit{dl}(C')$ may or may not include some of the labels in $\mathit{fresh}(S)$, depending on which modifiers of S have actually selected reaction rules of C.

As $\mathit{fresh}(S)$ contains all the labels that may be introduced in C' by refinement, the runtime check of $\mathit{fresh}(S) \subseteq \mathit{dl}(C')$ certainly rules out the application of a selective refinement to a class for which the selection was not designed. (Continuing with C' would likely introduce unexpected behaviors, such as a message sending on a label that does not actually occur in C', while refinement was designed to introduce it.) The condition $\mathit{fresh}(S) \subseteq \mathit{dl}(C')$ could be checked statically for selection patterns restricted to a single label, which would overly limit the expressiveness of selective refinement. For general selection patterns, this would require to statically trace synchronization, using verbose types and sophisticated typing rules. The type system in section B keeps track of the set of labels defined in each class, but it ignores all synchronization information. Hence, in this framework, the condition $\mathit{fresh}(S) \subseteq \mathit{dl}(C')$ can only be checked at runtime. The possible runtime failure could be compared, for instance and very roughly, with exceptions in ML.

Reduction for processes. Reduction $\longmapsto$ for processes is defined in Figure 5.

The elimination of all class constructs in a given process proceeds by iterating the rewrite steps above until a normal form is reached.

It thus remains to check that, in the irreducible process expression $\mathsf{obj}\ x = \mathsf{self}(z)\ C\ \mathsf{init}\ P\ \mathsf{in}\ P'$, class C is a valid definition D of the object calculus of section 2 (*i.e.* a disjunct of simple reaction rules). As the rules stand, C may contain an unresolved refinement (case (1)) or an abstract class (case (2)). Case (1) may occur only when rule MATCH fails to apply because $\mathit{fresh}(S) \not\subseteq \mathit{dl}(C')$. We considered this case as an acceptable runtime failure; it is currently modeled as a "blocked" selective refinement. Case (2) means that some of the labels discarded by refinement are not redefined elsewhere. This case will be ruled out by the type system.

We now discuss a few implementation issues. The patterns in D do not contain or constructs. However, one can design an object calculus with or-patterns that would be equivalent to the simple object calculus of section 2. We view this as an optimization issue. Perhaps more importantly, the unsharing of reaction rules with common sub-patterns performed by rule OR-PAT of the class semantics does not mean that guarded processes are duplicated by the compiler, which would preclude separate compilation. Guarded processes should be compiled as closures abstracted with respect to their free names (which include names bound by patterns), then duplicating P in the semantics means duplicating an indirection to the closure that implements P.

4 Static semantics

We developed a typing system for our calculus. By lack of space, the typing system is only described in appendix B. 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]. More specifically, types for both classes and object are extensible lists of labels paired with their types. Types are polymorphic: they include variables, whose scope is controlled at the class level. Furthermore, labels are partitioned into public and private labels. Private labels are visible only from within (i.e. in the right hand side of reaction rules) an object of the class that defines them of one of its subclasses. Public labels are visible from anywhere. As usual types and classe types can be seen as abstractions of objects and classes, *i.e.* of both the code of their reactions rules and the synchronization between their defined labels.

As one would expect, our type system prevents for *runtime errors*. That is, no label will ever carry a tuple of messages of the wrong arity and messages can only be sent to objects that belong to a class that defines the corresponding name, which rules out the so call *message-not-understood* errors. In a concurrent setting, the acceptance of a message on a defined label does not mean that the message will finally get processed. Namely, message processing ultimately depends on the synchronization between all methods of the objects, which is not recorded inside types, intendedly. In our opinion, detecting synchronization errors such as deadlocks, is not the job of a simplertype checker job, but of a more sophisticated static analyzers (that may well use type-based technology). Thus, our type checker simply rules out the trivial programming error of sending a message on a non-existent label.

Our type system also enforces object encapsulation. This is formalized by showing that well-typed programs cannot violate privacy, that is access a private label outside of its visibility scope. In order to state this result formally, we refine the semantics of section 2.3 so as to keep track of access privileges during the reduction.

5 Related and future works

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 definition, as we do in section 2. 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. In our proposal, we avoid this kind pattern matching to keep the calculus simple.

The generalization rule in 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.

Selective refinement also allows to superficially rename methods. For instance, we can define a class C' equal to $\mathsf{match}\ C\ \mathsf{with}\ \ell(x) \Rightarrow \ell'(x) \triangleright \mathsf{0}\ \mathsf{end}$. However, if class C has type $\zeta(\rho)B^{\emptyset}$, then class C' has type $(B, \ell' : B(\ell))^{\ell}$, which forces to redefine ℓ in C' before taking an instance. Consequently, the type system disallows renaming of methods. A similar problem occurs in sequential languages, where the type $(B \setminus \ell, \ell' : B(\ell))$ to C' would allow to redefine ℓ in C' with a type $\widetilde{\tau}$ incompatible with $B(\ell)$.

In sequential languages, deep method renaming, *i.e.* including rewrite of recursive calls, (or, more generally, method hiding) can be solved by introducing dictionaries [20] or views [22]. Views would also give other benefits in a concurrent setting. In particular, combining views with multiple inheritance would allow to build a class with two variants of its parent class, having slightly different synchronization policies. This can be used to distinguish between internal calls and external calls, for example. The integration of views with our calculus is a venue that deserves future investigation.

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.

Our type system could easily be refined to describe synchronization patterns of classes. This should allow to detect blocking refinements at compile time. Additionally, allowing the programmer to specify synchronization would facilitate modular programming and reduce synchronization design errors. However, such types would be so precise that a form of subtyping seems unavoidable in order to forget excessive information. This is another research direction for the future.

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 provide a way to assemble definitions, incrementally, before object instantiation.

Thereby, we partially recover the ability of the pi calculus to define the treatment of synchronizing messages in several parts of the program. Our layered design confines this capability to classes. This seems a good compromise between flexibility and simplicity of the computational model. Indeed, our proposal still enables efficient compilation of synchronization and facilitates 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.

[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.

[20] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Theory and Practice of Object Systems*, 1999. To appear.

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

[22] J. Vouillon. Combining subsumption and binary methods: An object calculus with views. Unpublished, 1999.

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

[24] 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 consider the same idea of inheritance as inducing a desirable modification of "acceptable states" [of objects]. Acceptable states are not explicit in the objective join calculus and serves more as a guide to introduce new labels for representing the states of the objects of the derived class.

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 by v.

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

```
or Check(A,i,n) ▷
      if n = A.size then z.Full(A, i, A.size)
      else if n = 0 then z.Empty(A, 0, 0)
      else z.Some(A, i, n)
or Init(size ) ▷ z.Empty(create(size),0,0)
```

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. Furhermore, 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 Some(A, i, n) ▷
      if n > 1 then z.Many(A, i, n) else z.One(A, i, n)
or match buff with Some(A, i, n) →
      (One(A, i, n) or Many(A, i, n)) ▷ 0 end
```

In the program above, we first define a new label *get2*, with its own activation condition. This reaction rule is cumulated with those of *buff*, where every occurrence of *Some* in join patterns is replaced by "*One* or *Many*" (match construct). Note that, by linearity of patterns, *Some* disappears totally from patterns. Such operation does not affect occurrences of *Some* in guarded processes, since parent behaviors are abstracted out. Hence, we add a new rule that consumes messages on *Some* and issues messages on *One* or *Many*, depending upon the value of n. Remark that the parent's code is not overridden.

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)) ▷
```

$$
\begin{array}{l}
\quad\quad r.reply(A[i]) \ \& \ z.NotAfterPut() \\
\quad \& \ z.Check(A, \ (i{+}1) \ \mathsf{mod} \ A.size, \ n{-}1) \\
\mathsf{or\ match}\ buff\ \mathsf{with} \\
\quad Init(size) \rightarrow Init(size\ ) \triangleright \ z.NotAfterPut() \\
\ \mid put(v,\ r) \rightarrow \\
\quad\quad put(v,\ r)\ \&\ (AfterPut()\ \mathsf{or}\ NotAfterPut()) \triangleright \\
\quad\quad\quad z.AfterPut() \\
\ \mid get(r) \rightarrow \\
\quad\quad get(r)\ \&\ (AfterPut()\ \mathsf{or}\ NotAfterPut()) \triangleright \\
\quad\quad\quad z.NotAfterPut() \\
\mathsf{end}
\end{array}
$$

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. We first implement simple locks by a *locker* class.

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

Although *locker* may be used alone, its intended usage is in combination with some other class c, in order to disable and enable message consumption on the labels of c. In this case, a simple disjunction of c and *locker* is not enough and some refinement of the parent c is required.

$$
\begin{array}{l}
\mathsf{class}\ locked\_buff\ = \mathsf{self}\ (z) \\
\quad locker \\
\mathsf{or\ match}\ buff\ \mathsf{with} \\
\quad\quad Init(size) \rightarrow Init(size\ ) \triangleright \ z.Free() \\
\quad \mid \mathsf{0} \rightarrow Free\ () \triangleright \ z.Free() \\
\quad \mathsf{end}
\end{array}
$$

The changes are quite simple: all parent rules except the one for *Init* must now consume and emit one message on *Free*. Adding *Free* both in patterns and actions is the purpose of the last clause, where the empty selection pattern acts like a default case. Indeed, 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.

As a consequence of these changes, parent rules are inhibited between a call to *suspend* and the next call to *resume*. 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.

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.) By contrast a message on a public label is unrestricted.

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:

- $ftv(\_)$ is the set of free type variables occurring in a type, a tuple of types, a type scheme, or a typing context.
- $B \restriction \mathcal{M}$ and $B \restriction \mathcal{F}$ 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 $(A \setminus dom(A')) \cup A'$;

Typing rules for names

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

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-Rule 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 Selective-Refinement merge virtual-label informations, as a disjunct or a parent class may effectively define a virtual label.

Rule Selective-Refinement types `match` C `with` S `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 $\mathit{fresh}(S) \cap dom(B_1) = \emptyset$ implies that names in $\mathit{fresh}(S)$ are not already declared in C.

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-Rule 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 $\mathsf{self}(z)\,(J_1 \triangleright P_1 \;\mathsf{or}\; J_2 \triangleright P_2)$.

First, let us consider the typing of pattern J_1. By rules MESSAGE-PATTERN and SYNCHRONIZATION we have:

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

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-RULE 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-RULE yields the following typing of $J_1 \triangleright P_1$:

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

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:

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

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:

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

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:

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

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:

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

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, $\ldots$, 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.

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:

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

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:

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

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 Reduction of messages

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.*

Theorem 3 (Privacy) *A well-typed chemical solution cannot violate privacy.*

B.5.2 Reduction of class expressions

Theorem 4 (Class-Reduction) *Class rewriting* $\stackrel{x}{\longmapsto}$ *and process rewriting* $\longmapsto$ *preserve typing.*

Theorem 5 (Class-Progress) *A well-typed program that cannot be rewritten by* $\stackrel{x}{\longmapsto}$ *either is in the core object calculus or it contains a selection* $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$ *such that* $C\ \mathsf{with}\ S \longmapsto C'$ and $\mathit{fresh}(S) \not\subseteq C'$.

Figure 4: Semantics for classes

$$\textsc{Class-Context}\quad \frac{C \xrightarrow{x} C'}{E[C] \xrightarrow{x} E[C']}$$

$$\textsc{Self}\quad \mathsf{self}(z)\, C \xrightarrow{x} C\{x/z\}$$

$$\textsc{Or-Pat}\quad J \;\mathsf{or}\; J' \triangleright P \xrightarrow{x} J \triangleright P \;\mathsf{or}\; J' \triangleright P$$

$$\textsc{Abstract-Cup}\quad L \;\mathsf{or}\; L' \xrightarrow{x} L \cup L'$$

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

$$\textsc{Class-Abstract}\quad C \;\mathsf{or}\; \emptyset \xrightarrow{x} C$$

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

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

$$\textsc{Filter-End}\quad M \triangleright P \;\mathsf{with}\; \emptyset \;\longmapsto\; M \triangleright P$$

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

$$\textsc{Filter-Or}\quad \frac{C_1 \;\mathsf{with}\; S|_{C_1} \longmapsto C_1' \qquad C_2 \;\mathsf{with}\; S|_{C_2} \longmapsto C_2'}{C_1 \;\mathsf{or}\; C_2 \;\mathsf{with}\; S \;\longmapsto\; C_1' \;\mathsf{or}\; C_2'}$$

where evaluation contexts E for class bodies are:

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

and $\mathit{fresh}(\cdot)$, $\cdot|_C$ are defined as follows:

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

$$\begin{array}{rcl} \emptyset|_C & = & \emptyset \\ (K \Rightarrow K' \triangleright P \mid S)|_C & = & \begin{cases} K \Rightarrow K' \triangleright P \mid (S|_C) & \text{if } \mathit{dl}(K) \subseteq \mathit{dl}(C) \\ S|_C & \text{otherwise} \end{cases} \end{array}$$

In Self, $x \notin \mathit{fv}(\mathsf{self}(z)\, C)$.

Figure 5: Reduction for processes

Class-Var

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

Class-Red

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

Process-Context

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

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

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**

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.* $V \subseteq dom(B)$), 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.* $V \subseteq dom(B)$) into patterns with labels in B', and prevents generalization on type variables Y.

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

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

Figure 9: Typing rules for processes

$$\textsc{Null}\quad \dfrac{}{A \vdash 0} \qquad \textsc{Send}\quad \dfrac{A \vdash x.\ell : ({\tau_i}^{i\in I}) \qquad (A \vdash x_i : \tau_i)^{i\in I}}{A \vdash x.\ell({x_i}^{i\in I})} \qquad \textsc{Join-Message}\quad \dfrac{A \vdash x.M_1 \qquad A \vdash x.M_2}{A \vdash x.(M_1 \ \&\ M_2)}$$

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

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

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

Figure 10: Typing rules for class bodies

$$\textsc{Reaction-Rule}\quad \dfrac{A' \vdash J :: Y.B \qquad A + A' \vdash P \qquad \mathit{dom}(A') = \mathit{rv}(J)}{A \vdash J \triangleright P :: Y.\zeta(\rho)B^{V}}$$

$$\textsc{Self-Binding}\quad \dfrac{A + x : [\rho],\, x : (B \upharpoonright \mathcal{F}) \vdash C :: Y.\zeta(\rho)B^{V}}{A \vdash \mathsf{self}(x)\, C :: Y.\zeta(\rho)B^{V}} \qquad \textsc{Abstract}\quad \dfrac{\mathit{dom}(B) = L}{A \vdash L :: Y.\zeta(\rho)B^{L}}$$

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

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

Figure 11: Typing rules for patterns

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

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

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

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

Figure 12: Typing rules for refinement clauses

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

Figure 13: Chemical semantics with privacy

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

Nil
$$\Vdash \psi \,\# \mathsf{0} \;\equiv\; \Vdash$$

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