Inheritance in the Join Calculus

Cédric Fournet* Cosimo Laneve† Luc Maranget‡
Didier Rémy‡

February 21, 2003

Abstract

We design an extension of the join calculus with class-based inheritance. Method calls, locks, and states are handled in a uniform manner, using asynchronous messages. Classes are partial message definitions that can be combined and transformed by means of operators for behavioral and synchronization inheritance. We also give a polymorphic type system that statically enforces basic safety properties. Our language and its type system are compatible with the JoCaml implementation of the join calculus.

*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.

Contents

1 Introduction

The *join calculus* is a simple name-passing calculus, related to the pi calculus but with a functional flavor [10, 9]. In this calculus, communication channels are statically defined: channels are created together with a set of *reaction rules* that specify, once and for all, how messages sent on these names will be synchronized and processed. These design choices favor the integration of concurrency and distribution within a programming language. Indeed, a typed language based on join calculus has been implemented as an extension of OCaml [17], called JoCaml [11, 15]. However, the above integration does not address object-oriented features. Precisely, JoCaml objects are just imported from OCaml and therefore they are sequential. The main contribution of this work is an extension of the join calculus with object oriented features that is compatible with the JoCaml implementation.

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-grained 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 [16] and simplifies reasoning on objects. However, the complete definition of object 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.

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 a typing discipline. In a concurrency setting, such promises can be rather hard to achieve.

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. The design of our class language follows from common programming patterns in the join calculus. We also illustrate this design by coding some standard problematic examples that mix synchronization and inheritance.

Our approach to computing classes is compatible with the JoCaml implementation of the join calculus [15], which relies on runtime representation of synchronization patterns as bitfields [16] and, on the contrary compiles processes into functional closures. As a consequence, synchronization patterns can be scanned and produced at runtime, while processes cannot be scanned (but still can be produced), as required by our operators on classes.

We then define a type system for objects and classes. We introduce types at the end of the paper only, to separate design issues and typing issues. The type system improves on our previous work on polymorphism in the join calculus [12]. As discussed in [12], message synchronization potentially weakens polymorphism. With classes, however, message synchronization may not be entirely determined as we type partial definitions. In order to preserve polymorphism, we thus rely on synchronization information in class types.

In addition to standard 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, we distinguish *public* and *private* labels. Then, the type system guarantees that private labels are accessed only from the body of a class used to create the object. The correctness of the type system is established for an operational semantics supplemented with privacy information.

As concern class operators, there is a trade-off between expressiveness and simplicity of both the static and dynamic semantics. In this paper, we favor expressiveness, while preserving type soundness and compositionality (with respect to compilation). As a consequence, we get a better understanding of inheritance and concurrency, and a touchstone for other design choices. Most of the complexity of our class operators stays within selective refinement. However, selective refinement is often used in rather simple patterns. This leaves place for simpler design choices (some of them will be discussed in Section 5).

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 rewriting semantics for the class language. In Section 4 we present more involved examples of inheritance and concurrency. In Section 5, we discuss other design choices. In Section 6, we provide a static semantics for our calculus and state its correctness. In Section 7, we discuss related works and possible extensions. Appendix A presents cross-encodings between the plain and objective join calculus. Appendix B gathers the main typing proofs.

2 The objective join calculus

We first focus on a core calculus dealing with objects. This calculus is a variant of the join calculus [10]. We illustrate the operations of the calculus, then we define its syntax and semantics.

[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 (see Appendix A.)

2.1 Getting started

The basic operation of our calculus is asynchronous message passing. For instance, the process $out.print\_int(n)$ sends a message with label $print\_int$ and 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 instance,

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

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:

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

The object $sbuffer$ has two labels get and put; 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 content n, 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, message synchronization and concurrent execution are expressed in a symmetric manner, on either side of $\triangleright$, using the same infix operator &.

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:

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

The object $abuffer$ can react in two different ways: a message (n, r) on put may be consumed by storing the value n 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 put are never blocked, even if no message is ever sent on get.

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:

$$\begin{array}{l}\mathsf{obj}\ buffer = \\ \quad put(n,r)\ \&\ Empty() \triangleright r.reply()\ \&\ buffer.Some(n) \\ \mathsf{or}\ get(r)\ \ \&\ Some(n) \triangleright r.reply(n)\ \&\ buffer.Empty() \\ \mathsf{init}\ \ buffer.Empty()\end{array}$$

Such a buffer can either be empty or contain one element. The state is encoded as a message pending on $Empty$ or $Some$, respectively. Object $buffer$ is created empty, by sending a first message on $Empty$ in the (optional) init part of the obj construct. As opposed to $abuffer$ above, a put message is blocked when the buffer is not empty.

To keep the $buffer$ object consistent, there should be a single message pending on either $Empty$ or $Some$. This invariant holds as long as external users

Figure 1: Syntax for the core objective join calculus

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

cannot send messages on these labels directly. In Section 6, we describe a refined semantics and a type system that distinguishes private labels (such as *Empty* and *Some*) from public labels and restricts access to private labels. In the examples, private labels conventionally bear an initial capital letter.

Once private labels are hidden, each of the three variants of *buffer* provides the same interface to the outside world (two methods labeled *get* and *put*) 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 labels $\ell \in \mathcal{L}$. Tuples 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 obj $x = D$ init P_1 in P_2 by omitting init P_1 when P_1 is $\mathsf{0}$.

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 object names $\widetilde{u}$ with scope P. As in join calculus, we require that every pattern M guarding a reaction rule be linear, that is, labels and names appear at most once in M. In addition, the object definition obj $x = D$ init P_1 in P_2 binds the name x to D. The scope of x is every guarded process in D (here x means "self") and the processes P_1 and P_2. Free names in processes and definitions, written $\mathit{fn}(\cdot)$, are defined accordingly; a formal definition of free names appears in Figure 4. Terms are taken modulo renaming of bound names (or α-conversion).

2.3 Chemical semantics

The operational semantics is given as a *reflexive chemical abstract machine* [10]. Each rewrite rule of the machine applies to configurations of objects and processes, called *chemical solutions*. A solution $\mathcal{D} \Vdash \mathcal{P}$ consists of a set of named definitions $\mathcal{D}$ (representing objects in solution) 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 $\mathcal{D}$. Chemical reductions are obtained by composing rewrite rules of two kinds: *structural rules* $\equiv$ represent the syntactical rearrangement of terms; *reduction rules* $\longrightarrow$ represent the basic computation steps.

Figure 2: Chemical semantics

Par
$$\Vdash P \;\&\; Q \;\equiv\; \Vdash P\,,\; Q$$

Nil
$$\Vdash \mathsf{0} \;\equiv\; \Vdash$$

Obj
$$\Vdash \mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q \;\equiv\; x.D \Vdash P\,,\; Q$$

Join
$$\Vdash x.(M \;\&\; M') \;\equiv\; \Vdash x.M\,,\; x.M'$$

Red
$$x.[M \triangleright P] \Vdash x.M\sigma \;\longrightarrow\; x.[M \triangleright P] \Vdash P\sigma$$

Chemistry
$$\frac{\mathcal{D}_0 \Vdash \mathcal{P}_1 \;\Rrightarrow\; \mathcal{D}_0 \Vdash \mathcal{P}_2}{\mathcal{D}\,, \mathcal{D}_0 \Vdash \mathcal{P}_1\,, \mathcal{P} \;\Rrightarrow\; \mathcal{D}\,, \mathcal{D}_0 \Vdash \mathcal{P}_2\,, \mathcal{P}}$$

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

The rules for the objective join calculus are given in Figure 2, with side conditions for rule Red: $[M \triangleright P]$ abbreviates a definition D that contains the reaction rule $M \triangleright P$; σ is a substitution with domain $\mathit{fn}(M)$; the processes $M\sigma$ and $P\sigma$ denote the results of applying σ to M and P, respectively.

Rules Par and Nil make parallel composition of processes associative and commutative, with unit $\mathsf{0}$. Rule Obj describes the introduction of an object. (Preliminary α-conversion may be required to pick a fresh name x.) Here, according to Obj and Par, expressions $\mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ and $\mathsf{obj}\ x = D\ \mathsf{in}\ P \;\&\; Q$ are equivalent. However, P and Q have different meaning in the semantics with privacy, where Q cannot access private names of D (see Section 6.1). Rule Join gathers messages sent to the same object. 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 of the pattern.

In chemical semantics, each rule usually mentions only the components that participate to the rewriting, while the rewriting applies to every chemical solution that contains them. More explicitly, we provide two context rules Chemistry and Chemistry-Obj. In rule Chemistry, the symbol $\Rrightarrow$ stands for either $\equiv$ or $\longrightarrow$. In rule Chemistry-Obj, the side condition $x \notin \mathit{fn}(\mathcal{D}) \cup \mathit{fn}(\mathcal{P})$ prevents name capture when introducing new objects (the sets $\mathit{fn}(\mathcal{D})$ and $\mathit{fn}(\mathcal{P})$ are defined in Figure 4).

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 Refining synchronization

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

$$\begin{array}{l} \mathsf{class}\ \ buffer = \mathsf{self}(z) \\ \quad get(r)\ \ \&\ Some(n) \triangleright r.reply(n)\ \&\ z.Empty() \\ \mathsf{or}\ \ put(n,r)\ \&\ Empty() \triangleright r.reply()\ \&\ z.Some(n) \end{array}$$

As regards the syntax, the prefix $\mathsf{self}(z)$ explicitly binds the name z to self. The class *buffer* can be used to create objects:

$$\mathsf{obj}\ \ b = buffer\ \mathsf{init}\ \ b.Empty()$$

Assume that, for debugging purposes, we want to log the buffer content on the terminal. We first add an explicit *log* method:

$$\begin{array}{l} \mathsf{class}\ \ logged\_buffer = \mathsf{self}(z) \\ \quad buffer \\ \mathsf{or}\ \ log()\ \&\ Some(n) \triangleright out.print\_int(n)\ \&\ z.Some(n) \\ \mathsf{or}\ \ log()\ \&\ Empty() \triangleright out.print\_string(\text{"Empty"})\ \&\ z.Empty() \end{array}$$

The class 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 who writes *logged_buffer* must have some knowledge of the parent class *buffer*, namely the use of private labels *Some* and *Empty* for representing the state.

Some other useful debugging information is the synchronous log of all messages that are consumed on *put*. This log can be produced by selecting the patterns in which *put* occurs and adding a printing message to the corresponding guarded processes:

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

The match construct can be understood by analogy with pattern matching *à la* ML, applied to the reaction rules of the parent class. In this example, every reaction rule from the parent *buffer* whose synchronization pattern contains the label *put* is replaced in the derived *logged_buffer_bis* by a rule with the same synchronization pattern (since *put* appears on both sides of $\Rightarrow$) and with the original guarded process in parallel with the new printing message (the original guarded process is left implicit in the match syntax). Every other parent rule is kept unchanged. Hence, the class above behaves as the definition:

$$
\begin{array}{l}
\mathsf{class}\ \ \mathit{logged\_buffer\_bis}\ = \mathsf{self}(z) \\
\quad \mathit{get}(r)\ \ \&\ \mathit{Some}(n) \triangleright\ \ r.\mathit{reply}(n)\ \&\ z.\mathit{Empty}() \\
\mathsf{or}\ \ \mathit{put}(n,r)\ \&\ \mathit{Empty}() \triangleright\ \ r.\mathit{reply}()\ \&\ z.\mathit{Some}(n)\ \&\ \mathit{out}.\mathit{print\_int}(n)
\end{array}
$$

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

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

In this case, the match construct performs a renaming of *put* into *Parent_put* in every pattern of class *buffer*, without affecting their guarded processes. The overall effect is similar to the overriding of the method *put* of *buffer* with a late-binding semantics. Namely, should there be a message $z.put(\widetilde{u})$ in a guarded process of the parent class, then, at runtime, $z.put(\widetilde{u})$ would reach the new definition of *put*.

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 logging might lead the buffer into deadlock. The other two examples expose another subtlety: in a sequential language, the distinction between logging *put* attempts and *put* 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 4 for explanations and examples. Classes are taken up to the associative-commutative laws for disjunction. 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 the labels in L but do not necessarily define them.

Join patterns J generalize the syntactic category of patterns M given in Figure 1 with an or operator that represents alternative synchronization patterns. Selection patterns K are either join patterns or the empty pattern $\mathsf{0}$. All patterns are taken up to equivalence laws: & and or are associative-commutative, & distributes over or, and $\mathsf{0}$ is the unit for &. Hence, every pattern K can be written as an alternative of patterns $\mathsf{or}_{i\in I}\, M_i$. We sometimes use the notation $K_1\ \&\ K_2$ for decomposing patterns M.

We always assume that processes meet the following well-formedness conditions:

Figure 3: Syntax for the objective join calculus

$P ::=$		**Processes**
	$\mathsf{0}$	null process
	$x.M$	message sending
	$P_1 \,\&\, P_2$	parallel composition
	$\mathsf{obj}\ x = C\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$	object definition
	$\mathsf{class}\ c = C\ \mathsf{in}\ P$	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**
	$\mathsf{0}$	empty pattern
	J	join pattern

1. All conjuncts M_i in the normal form of K are linear (as defined in Section 2.2) and binds the same names (*i.e.* have the same set of free names, as defined in Figure 4). By extension, we say that K binds the names $\mathit{fn}(M_i)$ bound in each M_i, and write $\mathit{fn}(K)$ for these names.

2. In a refinement clause $K_1 \Rightarrow K_2 \triangleright P$, the pattern K_1 is either M or $\mathsf{0}$, the pattern K_2 binds at least the names of K_1 ($\mathit{fn}(K_1) \subseteq \mathit{fn}(K_2)$), and K_1 is empty whenever K_2 is empty (so as to avoid the generation of empty patterns).

Binders for object names include object definitions (binding the defined object) and patterns (binding the received names). In a reaction rule $J \triangleright P$, the join pattern J binds $\mathit{fn}(J)$ with scope P. In a refinement clause $K_1 \Rightarrow K_2 \triangleright P$, the selection pattern K_1 binds $\mathit{fn}(K_1)$ with scope K_2 and P; the modification pattern K_2 binds $\mathit{fn}(K_2) \setminus \mathit{fn}(K_1)$ with scope P. Finally, $\mathsf{self}(x)\, C$ binds the object name x to the receiver (self) with scope C.

Class definitions $\mathsf{class}\ c = C\ \mathsf{in}\ P$ are the only binders for class names c, with scope P. The scoping rules appear in Figure 4; as usual, $\uplus$ means disjoint union. Processes, classes, and reaction rules are taken up to α-conversion.

Figure 4: Free names $fn(\cdot)$ and declared labels $dl(\cdot)$

In patterns M, join patterns J, and selection patterns K:

$$\begin{aligned} fn(0) &= \emptyset \\ fn(\ell(\widetilde{u})) &= \widetilde{u} \\ fn(J \mathbin{\&} J') &= fn(J) \cup fn(J') \\ fn(J \mathbin{\mathsf{or}} J') &= fn(J) \qquad\qquad (\text{also} = fn(J')) \end{aligned}$$

$$\begin{aligned} dl(0) &= \emptyset \\ dl(\ell(\widetilde{u})) &= \{\ell\} \\ dl(J \mathbin{\&} J') &= dl(J) \uplus dl(J') \\ dl(J \mathbin{\mathsf{or}} J') &= dl(J) \cup dl(J') \end{aligned}$$

In refinement clauses S:

$$\begin{aligned} fn(\big|^{i \in I}(K_i \Rightarrow K'_i \triangleright P_i)) &= \textstyle\bigcup_{i \in I} fn(P_i) \setminus fn(K'_i) \\ dl(\big|^{i \in I}(K_i \Rightarrow K'_i \triangleright P_i)) &= \textstyle\bigcup_{i \in I} dl(K'_i) \setminus dl(K_i) \end{aligned}$$

In classes C:

$$\begin{aligned} fn(c) &= \{c\} \\ fn(L) &= \emptyset \\ fn(J \triangleright P) &= fn(P) \setminus fn(J) \\ fn(C_1 \mathbin{\mathsf{or}} C_2) &= fn(C_1) \cup fn(C_2) \\ fn(\mathsf{self}(z)\, C) &= fn(C) \setminus \{z\} \\ fn(\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}) &= fn(C) \cup fn(S) \end{aligned}$$

$$\begin{aligned} dl(c) &= \emptyset \\ dl(L) &= L \\ dl(J \triangleright P) &= dl(J) \\ dl(C_1 \mathbin{\mathsf{or}} C_2) &= dl(C_1) \cup dl(C_2) \\ dl(\mathsf{self}(z)\, C) &= dl(C) \\ dl(\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}) &= dl(C) \cup dl(S) \end{aligned}$$

In processes P:

$$\begin{aligned} fn(0) &= \emptyset \\ fn(x.M) &= \{x\} \cup fn(M) \\ fn(P \mathbin{\&} Q) &= fn(P) \cup fn(Q) \\ fn(\mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ Q) &= (fn(C) \cup fn(P) \cup fn(Q)) \setminus \{x\} \\ fn(\mathsf{class}\ c = C\ \mathsf{in}\ P) &= fn(C) \cup (fn(P) \setminus \{c\}) \end{aligned}$$

In solutions:

$$\begin{aligned} fn(\mathcal{D}) &= \textstyle\bigcup_{x.D \in \mathcal{D}} (\{x\} \cup fn(D)) \\ fn(\mathcal{P}) &= \textstyle\bigcup_{P \in \mathcal{P}} fn(P) \end{aligned}$$

Labels don't have scopes. Labels declared in patterns and classes, written $dl(K)$ and $dl(C)$, are specified in Figure 4. We say that a label is *declared but undefined* when it is declared only in abstract classes.

3.3 Rewriting semantics of the class language

The semantics of classes is defined by reduction to the core calculus. The rules are given in Figure 5. Rules CLASS-SUBST and CLASS-RED describe class rewritings for processes. Rule CLASS-RED lifts an auxiliary reduction on classes $\overset{x}{\longmapsto}$ to processes. This auxiliary reduction is parameterized by the name of the object and replaces the (outermost) self names with the reduction parameter.

Rule SELF removes inner bindings for the name of self. Rule OR-PAT lifts or constructs from patterns to classes. The next two rules for classes simplify abstract classes, rule CLASS-ABSTRACT discards empty abstract classes; rule ABSTRACT-CUT discards abstract labels that are declared elsewhere. Rule MATCH reduces selective refinements $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$ by means of an auxiliary relations, described below. Rule CLASS-CONTEXT applies rewriting under disjunctions and selective refinements.

The auxiliary reduction $\longmapsto$ computes *filters* of the form $C\ \mathsf{with}\ S$. Every reaction rule $M \triangleright P$ in C is rewritten according to the leftmost clause of S that matches M, that is, whose selection pattern K is a sub-pattern of M (rules FILTER-NEXT and FILTER-APPLY). If no clause of S matches M, then the reaction rule is left unchanged (rule FILTER-END). Abstract classes L in C are also left unchanged (rule FILTER-ABSTRACT).

Note that rule FILTER-APPLY can be used only if the pattern K_2 & K introduced by rule FILTER-APPLY is well-formed, *i.e.*, (1) K_2 and K are not both empty, (2) $fn(K_2) \cap fn(K) = \emptyset$, and (3) $dl(K_2) \cap dl(K) = \emptyset$. Condition (1) is enforced by syntactic restriction 2 of Section 3.2. Condition (2) can be enforced by α-conversion. Condition (3) will be checked by the type system.

The rewriting $C\ \mathsf{with}\ S \longmapsto C'$ may erase every reaction rule defining a label of C. Specifically, rule FILTER-APPLY removes the labels $dl(K_1) \setminus dl(K_2)$. To prevent those labels from being undeclared in C', rule FILTER-APPLY records them as abstract labels. The type system of Section 6 will compute an approximation of abstract names to statically ensures that every erased label is actually redefined before the class is instantiated.

Conversely, the rewriting $C\ \mathsf{with}\ S \longmapsto C'$ may introduce labels in C' that are undeclared in C. Specifically, rule FILTER-APPLY introduces the labels $dl(K_2) \setminus dl(K_1)$. The set $dl(S)$ defined in Figure 4 statically collects all such labels. However, some clauses of S may not be used, hence some label of $dl(S)$ may not be declared in C'. This situation often corresponds to a programming error. To prevent it, we supplement rule MATCH with the premise $dl(S) \subseteq dl(C')$ that blocks the rule, and we interpret this situation as a dynamic *refinement error*.

Next, we summarize the outcome of class reduction for processes. One can easily check that the rewriting semantics is deterministic and always terminates. Using rule CLASS-SUBST, any class construct can be eliminated, so we focus on object creations:

Lemma 1 (Rewriting) *Let P be a process of the form $\mathsf{obj}\ x = C\ \mathsf{init}\ Q\ \mathsf{in}\ Q'$ such that rule* CLASS-RED *does not apply to P. One of the following holds:*

Figure 5: Rewriting semantics of the class language

Rules for processes ($\longmapsto$)

Class-Subst

$$\mathsf{class}\ c = C\ \mathsf{in}\ P \longmapsto P\{C/c\}$$

Class-Red

$$\frac{C \xrightarrow{x} C'}{\mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ P' \longmapsto \mathsf{obj}\ x = C'\ \mathsf{init}\ P\ \mathsf{in}\ P'}$$

Rules for classes ($\xrightarrow{x}$)

Self

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

Or-Pat

$$J\ \mathsf{or}\ J' \triangleright P \xrightarrow{x} J \triangleright P\ \mathsf{or}\ J' \triangleright P$$

Class-Abstract

$$C\ \mathsf{or}\ \emptyset \xrightarrow{x} C$$

Abstract-Cut

$$\frac{L' = L \setminus dl(C) \qquad L \neq L'}{C\ \mathsf{or}\ L \xrightarrow{x} C\ \mathsf{or}\ L'}$$

Match

$$\frac{C\ \mathsf{with}\ S \ \longmapsto\ C' \qquad dl(S) \subseteq dl(C')}{\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}\ \xrightarrow{x}\ C'}$$

Class-Context

$$\frac{C \xrightarrow{x} C'}{E[C] \xrightarrow{x} E[C']}$$

Evaluation contexts for classes

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

Rules for filters ($\longmapsto$)

Filter-Apply *(Side conditions in the text)*

$$K_1 \mathbin{\&} K \triangleright P\ \mathsf{with}\ K_1 \Rightarrow K_2 \triangleright Q \mid S \ \longmapsto\ K_2 \mathbin{\&} K \triangleright P \mathbin{\&} Q\ \mathsf{or}\ dl(K_1) \setminus dl(K_2)$$

Filter-Next

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

Filter-End

$$C\ \mathsf{with}\ \emptyset \ \longmapsto\ C$$

Filter-Or

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

Filter-Abstract

$$L\ \mathsf{with}\ S \ \longmapsto\ L$$

Completion: *C is a disjunction of reaction rules ($C = \mathsf{or}_{i=1}^{n} M_i \triangleright P_i$).*

Failure: *For some evaluation context E, we have:*

1. *$C = E[c]$ and c is free (undefined class).*
2. *$C = E[L]$ and L is not under a* match *(undefined label).*

Refinement error: *C contains a blocked refinement in evaluation context:*
$C = E[\mathsf{match}\ C'\ \mathsf{with}\ S\ \mathsf{end}]$, $C'\ \mathsf{with}\ S \longmapsto C''$, *and* $dl(S) \not\subseteq dl(C'')$.

The distinction between failures and refinement errors matters only in the typed semantics, which prevents all failures but not necessarily refinement errors.

Like in Section 2, we call "definition" and write D instead of C for any class of the form given in the Completion case.

3.4 Implementation issues

In order to compile disjunctions and selective refinements, one must access the patterns of parent classes. This hinders the abstract compilation of patterns, but does not otherwise preclude separate compilation. As in a functional setting, the guarded processes attached to individual reaction rules can be immediately compiled into closures abstracted with respect to their free names, including formal message parameters. This approach is compatible with the JoCaml implementation, which keeps a concrete representation of 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 [16]. In an implementation of the object calculus with classes, such vectors of bits would serve as a basis of a data structure for representing classes at runtime.

In the core objective join calculus, patterns do not contain alternatives "or". To eliminate them, rule Or-Pat duplicates reaction rules whose patterns contain alternatives. Alternatively, we could have supplemented the object calculus with or-patterns, but we view this as an optimization issue. Perhaps more importantly, the unsharing of reaction rules performed by rule Or-Pat does not mean that guarded processes are duplicated by the compiler. Since guarded processes are compiled as closures, duplicating P in the semantics means duplicating an indirection to the closure that implements P.

4 Solving challenging examples

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, the signature of a parent class does not usually convey any information on its synchronization behavior. As a result, it is often awkward, or even impossible, to refine a concurrent behavior using inheritance. (More conservatively, object-oriented languages with plain concurrent extensions usually require that the synchronization properties be invariant through inheritance, e.g., that all method calls be synchronized. This strongly constrains the use of concurrency.) This well-known problem is often referred to as the *inheritance anomaly*. Unfortunately, inheritance anomaly is not defined formally, but by means of problematic examples.

In [18] for instance, Matsuoka and Yonezawa identify three patterns of inheritance anomaly. For each pattern, they propose a refinement of the class language that suffices to express the particular synchronization property at hand: they identify the parts of the code that control synchronization in the parent class (which are otherwise hidden in the body of the inherited methods); they express this "concurrency control" in the interface of the class; and they rely on the extended interface to refine synchronization in the definition of subclasses.

In principle, it should be possible to fix any particular anomaly by enriching the class language in an *ad hoc* manner. However, the overall benefits of this approach are unclear. Our approach is rather different: we start from a core calculus of concurrency, rather than programming examples, and we are primarily concerned with the semantics of our inheritance operators. Tackling the three patterns of inheritance anomaly of [18], as we do in this section, appears to be a valuable test of our design. Indeed, the issue of inheritance anomaly is out of the scope of this paper. We refer to other studies, such as [20], for analyses of this issue, which, however, concern *different* and *untyped* frameworks.

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.

In the following examples, we use a language extended with basic datatypes. Booleans and integers are equipped with their usual operations. Arrays are created by $\mathsf{create}(n)$, which gives an uninitialized array of size n. The size of an array A is given by $A.size$. Finally, the array $A[i] \leftarrow v$ is obtained from A by overwriting its i-th entry with value v.

The FIFO buffer of [18] can then be written as follows:

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

The state of the buffer is encoding a circular array represented by a message with label *Empty*, *Some*, or *Full*. These labels carry three arguments. The first one, A is the array; the second one, i is the index of the first element of the buffer; the third one, n is the number of elements in the buffer. 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*. Since *Check* appears alone in a join pattern, message sending on *Check* acts like a function call.

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 *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 methods *get* and *put*, which are inherited from the parent *buff*. Therefore, label *Some* is not detailed enough and is replaced with two labels *One* and *Many*. *One* represents a state with exactly one item in the buffer; *Many* represents 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-dependent 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) ⇒
      put(v, r) & (AfterPut() or NotAfterPut()) ▷ z.AfterPut()
 | get(r) ⇒
      get(r) & (AfterPut() or NotAfterPut()) ▷ z.NotAfterPut()
end
```

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 define a general-purpose lock with the following *locker* class:

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

This class can be used to create locks, but it can also be combined with some other class such as *buff* to temporarily prevent message processing in *buff*. To this end, a simple disjunction of *buff* and *locker* is not enough and some refinement of the parent class *buff* is required:

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

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.

Further anomalies Dealing with the examples above does not mean that we *solved* the inheritance anomaly problem. Indeed, most limitations of expressiveness can be interpreted as inheritance anomalies. We conclude this section with a more difficult example, for which we only have a partial solution. The difficulty arises when we try to delegate privileged access to an object.

Consider a class with some mutable state, such as the one-place buffer of Section 3.1:

$$\begin{array}{l} \mathsf{class}\ \ buffer = \mathsf{self}(z) \\ \quad get(r)\ \ \&\ Some(n) \triangleright r.reply(n)\ \&\ z.Empty() \\ \mathsf{or}\ \ put(n,r)\ \&\ Empty() \triangleright r.reply()\ \&\ z.Some(n) \end{array}$$

We want to supplement *buffer* with an *incr* method that increments the buffer content. One might also require *incr* to be performed by using *get* and *put* from another object *server*:

$$\begin{array}{l} \mathsf{obj}\ server = \\ \quad do\_incr(x,r) \triangleright \mathsf{obj}\ s = reply(n) \triangleright x.put(n{+}1,r)\ \mathsf{in}\ x.get(s) \end{array}$$

Furthermore, we require that the call to *put* from inside *do_incr* never block. Thus, any other call to *put* should be blocked during the execution of *do_incr*. To enforce this partial exclusion, we introduce an *Exclusive* flag, we take two copies of the parent class, and we specialize their definitions of *put* for external calls (disallowed during an increment) and privileged calls (performed only from the server). In the latter refinement clause, the conflicting method *put* is renamed to *Put_priv*, and a "proxy object" that forwards *put* calls to *Put_priv* is passed to the server.

$$
\begin{array}{l}
\mathsf{class}\ \ counter = \mathsf{self}(z) \\
\qquad \mathsf{match}\ buffer\ \mathsf{with} \\
\qquad\quad put(n,r) \Rightarrow put(n,r)\ \&\ Exclusive() \triangleright z.Exclusive() \\
\qquad \mathsf{end} \\
\quad \mathsf{or}\ \mathsf{match}\ buffer\ \mathsf{with} \\
\qquad\quad put(n,r) \Rightarrow Put\_priv(n,r) \triangleright 0 \\
\qquad \mathsf{end} \\
\quad \mathsf{or}\ incr(r)\ \&\ Exclusive() \triangleright \\
\qquad\quad \mathsf{obj}\ s = reply() \triangleright r.reply()\ \&\ z.Exclusive()\ \mathsf{in} \\
\qquad\quad \mathsf{obj}\ proxy = get(r) \triangleright z.get(r) \\
\qquad\qquad\qquad \mathsf{or}\ put(n,r) \triangleright z.Put\_priv(n,r)\ \mathsf{in} \\
\qquad\quad server.do\_incr(proxy,s) \\
\quad \mathsf{or}\ Init() \triangleright z.Empty()\ \&\ z.Exclusive()
\end{array}
$$

Our solution is not entirely satisfactory. In particular, the duplication of method *put* forces further refinements of counter to treat the two methods *put* and *Put_priv* in a consistent manner. For instance, if we refine *counter* in order to log successful puts, as we do in example *logged_buffer_bis* of Section 3.1, then the puts from *server* are not logged.

We can improve this by keeping a single copy of *put* —the private one— and make the public method *pub* synchronously forwarding its call to the private version (synchrony is required to ensure that the *Exclusive* lock is only released when the forwarded call to the private version of *put* has returned.)

$$
\begin{array}{l}
\mathsf{class}\ \ counter = \mathsf{self}(z) \\
\qquad \mathsf{match}\ buffer\ \mathsf{with} \\
\qquad\quad put(n,r) \Rightarrow Put\_priv(n,r) \triangleright 0 \\
\qquad \mathsf{end} \\
\quad \mathsf{or}\ put(n,r)\ \&\ Exclusive() \triangleright \\
\qquad\quad \mathsf{obj}\ s = reply(x) \triangleright r.reply(x)\ \&\ z.Exclusive()\ \mathsf{in} \\
\qquad\quad z.Put\_priv(n,s) \\
\quad \mathsf{or}\ incr(r)\ \&\ Exclusive() \triangleright \\
\qquad\quad \mathsf{obj}\ s = reply() \triangleright r.reply()\ \&\ z.Exclusive()\ \mathsf{in} \\
\qquad\quad \mathsf{obj}\ proxy = get(r) \triangleright z.get(r) \\
\qquad\qquad\qquad \mathsf{or}\ put(n,r) \triangleright z.Put\_priv(n,r)\ \mathsf{in} \\
\qquad\quad server.do\_incr(proxy,s) \\
\quad \mathsf{or}\ Init() \triangleright z.Empty()\ \&\ z.Exclusive()
\end{array}
$$

The class can not be inherited as long as the clients operates on the private version *Put_private* instead of the public name *put* that is just used as an entry point.

A more elegant approach is to stick to views in the style of [27, 30]. A view is a map from method names to method slots. Method slots are positions for method implementations in a class, which are used for self-referencing other methods of the same object. Overriding of a method amounts to change the method implementation, thus affecting the other methods that reference the overridden method slot. Overriding must preserve method slot types. Therefore method slots cannot be hidden in classes, since otherwise, they could be redefined with another incompatible type. On the contrary, a method name can be safely erased from a view, leaving its slot unreachable (in that view). A later redefinition of the same name will simply allocate a new unrelated slot. In the traditional approach—the one that we followed—names and slots are in a one-to-one correspondence. Hence, methods cannot be forgotten in classes (they can only become abstract and will have to be redefined later before taking an instance of the class). Views are a much more powerful mechanism, but they require a significant complication of the type system.

5 Variations on the class language

Our language of classes favor expressiveness while retaining type soundness and modularity. Thus, several other design choices, which could allow significant simplifications or enforce stronger invariants, can be derived from our proposal either by small variations or restrictions.

5.1 Object initialization

Usually, object initialization is defined at the class level, rather than at the object level, which is hidden to users. Instead, in our proposal objects are visible because they provide the basic semantics. Regarding the object initialization, we enriched the object language as little as needed by splitting names into private and public. Hereafter, we illustrate a more standard approach to object initialization by means of straightforward translation of a higher-level language. (Indeed, examples of Section 4 conform with this approach.)

In the user-language we replace class and obj declarations by the following ones:

- Class definitions are written $\mathsf{class}\ c(\widetilde{x}) = I\ C\ \mathsf{in}\ P$ where
  - the name c of the class takes a list of arguments $(\widetilde{x})$;
  - I is a list of inheritance clauses of the form $(\mathsf{inherits}\ c_i(\widetilde{u}_i)\ \mathsf{as}\ d_i)^{i\in I}$ where each c_i is a previsouly defined class and d_i a local name for the body of c_i.
  - C is as before, except that it contains exactly one rule of the form $c^{init}(\widetilde{x}) \triangleright Q$. The names c^{init} are private and special in the sense that they cannot occur (in the user-language) anywhere else. Indeed, c^{init} play the role of class constructors.
  - P is as before.
- Object definitions are written $\mathsf{obj}\ x = c(\widetilde{u})\ \mathsf{in}\ P$ where c has been defined as above.

The translation of the above declarations are, respectively:

- class $c(\widetilde{x})$ = match C with $c^{init}(\widetilde{x}) \Rightarrow c^{init}(\widetilde{x})\ \&^{i \in I}\ {c_i}^{init}(\widetilde{u}_i)$ end in P
- obj $x = c$ init $c^{init}(\widetilde{u})$ in P.

In this user language, the class is responsible for object initialization. Moreover, by constructions, subclasses always invoke the initialization methods of its parent classes. Indeed, other design choices are possible. For instance, this design easily generalizes to allow multiple class constructors.

5.2 Restriction of selective refinements

The examples in Section 4 only use selective refinement in a restrictive and simple form. In particular, refinement clauses $K \Rightarrow K' \triangleright P$ always uses K with 0 or 1 message. Restricting to such cases simplifies rule FILTER-APPLY in the rewriting semantics (Figure 5) and the static semantics of classes given below in Section 10.

A different approach has been taken in Polyphonic $C^\sharp$ [3]. This language extends $C^\sharp$ with reaction rules *a la* join calculus (called *chords*). As inheritance is concerned, Polyphonic $C^\sharp$ is quite severe with respect to join patterns: if a method is overridden, then all join patterns concerning that method must be overridden, and so on, transitively. This apporach is significantly different than ours. In particular, while we allow overriding of as few methods as possible during inheritance, they instead require overriding of too many methods. For instance, in the common pattern where every method synchronizes with a global object state, overriding any method would require overriding all of them.

6 Types and privacy

The static semantics of our calculus extends those of the core join calculus for concurrency and synchronization [12] and of OCaml for the class-layer [26], respectively. As regards polymorphism, the type system supports ML parametric polymorphism and uses row variables to enable some form of subtyping [32, 26]. It also improves on [12], so as to match at least the implementation [15] and avoid the limitation pointed out in [24]. As regards classes, we supplement the typing of [26] in order to deal with the new operator of selective refinement and to collect some synchronization information.

6.1 A semantics with privacy

In this section, we specify the dynamic errors that are detected by typing. For instance, the type system detects *message-not-understood* errors: no message can be sent to an object with a label that is undefined at that object. (Of course, the type system does not ensure any processing of messages.) In addition, the type system enforces object encapsulation; this is stated for a chemical semantics extended with a notion of privacy.

We partition labels $\ell \in \mathcal{L}$ into *private labels* $f \in \mathcal{F}$ and *public labels* $m \in \mathcal{M}$. Informally, a message on a private label can only be sent internally, that is, from within either a reaction rule or the init part of the object. Conversely, a message on a public label can be sent from any context that has access to the object name. However, the origin of a message is a static notion, which is not preserved in the original chemical semantics given in Section 2. For instance, rule OBJ in Figure 2, used to create new objects, immediately mixes its privileged init process with all other running processes.

In order to express subject reduction and type safety with privacy, we thus supplement our chemical semantics with privacy annotations at runtime. In the state of the refined machine, every running process P and active definition D is prefixed by a *string of object names* ψ that records the nesting of objects. Precisely, the string $y_1 \ldots y_n x$ indicates that object x was created within the definition (or the init process) of objects $y_1, \ldots, y_n$ and thereby can access their private labels. The chemical state, or *solution*, is written $\mathcal{D} \Vdash \mathcal{P}$. It consists of a set $\mathcal{D}$ of prefixed definitions $\psi x \,{}_{\#} D$ and a multiset of prefixed processes $\psi \,{}_{\#} P$. A solution is well-formed when all prefixes agree on object nesting, *i.e.*, if ψx and φx appear in prefixes, then $\psi = \varphi$. As before, we also assume that there is a single definition for every object in the solution. These properties are preserved by chemical rewriting.

We use the rules of Figure 6, with the following side conditions: for OBJ, D is a definition, i.e., a class of the form $\mathsf{or}_{i=1}^{n} M_i \triangleright P_i$; for RED, $[M \triangleright P]$ abbreviates a definition that contains the rule $M \triangleright P$ and σ is a substitution on the names bound in M.

Except for the bookkeeping on static environments, rules NIL, PAR, JOIN, OBJ, RED, CHEMISTRY, and CHEMISTRY-OBJ 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 can 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. This semantics is a refinement of the previous semantics. (Formally, every reduction in this semantics can be mapped into zero or one reduction in the previous semantics after removing all prefixes.)

Our privacy policy states that a message sent from object y to object x on a private label is valid as long as y has been created by a process of x (*cf.* rule PRIVATE-COMM). We take the presence of non-routable messages as our definition of a privacy error. We also give a formal definitions for other errors, which do not depend on privacy information.

Definition 1 *A solution $\mathcal{D} \Vdash \mathcal{P}$* fails *when one of the following holds:*

Free variables: *the solution contains a free class variable, or a free object name that is not defined in $\mathcal{D}$.*

Runtime Failure: *for some $\psi \,{}_{\#}\, x.\ell(\widetilde{u}) \in \mathcal{P}$ and $\psi' x \,{}_{\#}\, D \in \mathcal{D}$, we have*

1. Failed privacy: $\ell \in \mathcal{F}$ *and ψ' is not a prefix of ψ.*
2. Undeclared label: $\ell \notin dl(D)$.
3. Arity mismatch: *$\ell(\widetilde{y})$ appears in a pattern of D with different arities for $\widetilde{y}$ and $\widetilde{u}$.*

Class rewriting failure: *for some $\psi \,{}_{\#}\, P \in \mathcal{P}$, the process P is a failure, as defined in Lemma 1 of Section 3.3.*

Figure 6: Chemical semantics with privacy

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

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

Join
$$\Vdash \psi\,\#x.(M \mathbin{\&} M') \;\equiv\; \Vdash \psi\,\#x.M\,,\psi\,\#x.M'$$

Obj
$$\Vdash \psi\,\#\mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q \;\equiv\; \psi x\,\#D \Vdash \psi x\,\#P,\ \psi\,\#Q$$

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

Chemistry
$$\frac{\mathcal{D}_0 \Vdash \mathcal{P}_1 \;\Rrightarrow\; \mathcal{D}_0 \Vdash \mathcal{P}_2}{\mathcal{D}\,,\mathcal{D}_0 \Vdash \mathcal{P}_1\,,\mathcal{P} \;\Rrightarrow\; \mathcal{D}\,,\mathcal{D}_0 \Vdash \mathcal{P}_2\,,\mathcal{P}}$$

Chemistry-Obj
$$\frac{\Vdash P \equiv \psi x\,\#D \Vdash \mathcal{P}' \qquad x \notin \mathrm{fn}(\mathcal{D}) \cup \mathrm{fn}(\mathcal{P})}{\mathcal{D} \Vdash P\,,\mathcal{P} \equiv \mathcal{D}\,,\psi x\,\#D \Vdash \mathcal{P}'\,,\mathcal{P}}$$

6.2 Type expressions

The grammar for type expressions is given in Figure 7. We build types out of a countable set of type variables, ranged over by θ and a countable set of row variables, ranged over by ϱ. 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 type variables. We also abbreviate type schemes $\forall\emptyset.\tau$ as τ.

Object types $[\rho]$ collect the types of public labels. For instance the type of the object *continuation* from Section 2.1 is $[\mathit{reply} : (\mathsf{int})]$. Object types may end with a row variable (open object types), as in OCaml [26]. For instance, consider a simple rendez-vous object *join*, with an internal counter:

$$\begin{array}{l}\mathsf{obj}\ \mathit{join} = \mathit{sync1}(\mathit{r1}) \mathbin{\&} \mathit{sync2}(\mathit{r2}) \mathbin{\&} \mathit{Count}(x) \triangleright \\ \quad \mathit{r1}.\mathit{reply}() \mathbin{\&} \mathit{r2}.\mathit{reply}() \mathbin{\&} \mathit{join}.\mathit{Count}(x{+}1)\end{array}$$

The type of the object *join* is $[\mathit{sync1} : ([\mathit{reply} : (); \varrho]); \mathit{sync}2 : ([\mathit{reply} : (); \varrho'])]$, telling that the *sync* labels accept messages composed of any object with a *reply* label which in turn accepts empty messages. We assumed that the label *Count* is private, hence it does not appear in the type of *join*.

Internal types B are used to describe both public and private labels. Such internal types appear in class types (see below). They are also used to describe the internal type of self while typechecking class bodies where sending messages

Figure 7: Syntax for type expressions

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

on private names is allowed (Section 6.4). We observe that B is a partial function from labels to tuple types; therefore we will address types of labels by function application.

We use the following standard notations. 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 $\ell_1, \ldots \ \ell_n$. For a given set of labels L, we write $B \upharpoonright L$ for the restriction of B to the labels of L. We also write $B_1 \oplus B_2$ for the union of B_1 and B_2, with the statement that B_1 and B_2 coincide on their common labels, and state $B_1 \subseteq B_2$ when there is B_1' such that $B_1 \oplus B_1' = B_2$. We write $dom(B)$ for the set of labels listed in B.

Class types have the form $\forall X.\zeta(\rho)B^{W,V}$. The set X collects all object type variables and row type variables appearing in ρ or B that are polymorphic. The row type ρ collects all the constraints on the type $[\rho]$ of self, *i.e.* an object of the class being defined. (These constraints originate from recursive calls, and also from passing self as a parameter in messages.) The internal type B lists the types for all public and private labels declared in the class. The consistency between ρ and B is checked only when objects are created. The set W collects the *coupled labels* of the class, as explained below. The set $V \subseteq dom(B)$ contains labels that are declared but undefined; we call these labels "virtual labels"; classes with virtual labels cannot be instantiated.

Given the sophistication of class types, we delay the presentation of a complete example until section 6.7. However, if the previous definition of the *join* object is lifted into a class definition, then the B component of its type is $sync1 : ([reply : (); \varrho]); sync2 : ([reply : (); \varrho']); Count : (\mathsf{int})$, there are no virtual nor coupled labels, and both variables ϱ and ϱ' can be made polymorphic. Moreover, the internal type for the objet *join* should also contain $Count : (\mathsf{int})$.

6.3 Polymorphism and inheritance

We now discuss the interaction between synchronization, inheritance, and polymorphism, in order to define the generalization conditions for type variables. (The reader not interested in polymorphism may skip these definitions and their usage in the type system.)

In contrast with functional method types, the types of messages sent on labels appearing in the same pattern must agree on the instantiation of any shared type variables. Consider, for instance, the *sbuffer* of Section 2.1:

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

The types of *get* and *put* are $([\mathit{reply} : (\theta); \varrho])$ and $(\theta, [\mathit{reply} : (); \varrho'])$, respectively. In order to retain type consistency for messages on $r.\mathit{reply}$, the two occurrences of θ in *get* and *put* must be instantiated to the same type. Hence, variable θ cannot be generalized. Conversely, type variables ϱ and ϱ' appearing in the type of a single method can be generalized; this is the main source of polymorphism in the objective join calculus[2].

We introduce auxiliary definitions to capture the sharing of messages and type variables in patterns. Let K be a pattern. The pattern $\widehat{K}$ is obtained from K by erasing every message that carries an empty tuple. The set of *coupled labels* of K, written $\mathit{cl}(K)$, collects the labels whose contents are effectively synchronized in K: we let $\mathit{cl}(M) = \mathit{dl}(\widehat{M})$ when the pattern $\widehat{M}$ contains at least two messages, and $\mathit{cl}(M) = \emptyset$ otherwise. For more complex patterns of the form $K = \mathsf{or}_{i \in I}\, M_i$, we let $\mathit{cl}(K) = \bigcup_{i \in I} \mathit{cl}(M_i)$.

Similarly for types, $\widehat{B}$ is obtained from B by removing every label with an empty tuple type. We write $\mathit{ftv}(\_)$ for the the set of free type variables occurring in a type, a tuple type, a type scheme, or a typing environment (defined in Section 6.4). We let $\mathit{ctv}(B)$ be the subset of type variables in $\mathit{ftv}(B)$ that occur in at least two labeled entries of B:

$$\mathit{ctv}(B) = \bigcup_{\ell \neq \ell'} \mathit{ftv}(B(\ell)) \cap \mathit{ftv}(B(\ell'))$$

Assuming that B gathers the types for all messages that can be sent to an object, the set $\mathit{ctv}(B)$ contains any variable that cannot be generalized because of synchronization, independently of the patterns for that object. When the synchronization patterns are known, however, one can usually compute a smaller set of such variables.

Since objects and classes can refine other classes, we compute a safe approximation of non-generalizable variables in contexts where the patterns for the objects are still unsettled. To this end, the type of each class carries a set W of *coupled labels*, such that $\mathit{cl}(J) \subseteq W$ for all patterns J that may appear in an object of a class of that type. Eventually, the typing rule for object definition will generalize all type variables except those that appear in $\mathit{ctv}(B \restriction W)$, where B gathers the types for all messages of the object and W collects all potentially-coupled labels.[3]

The main issue is to compute the coupled labels for a refined class, of the form $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$. Instead of the patterns for C, we only know B and W from its class type. Since the refinement may leave unchanged some rules of C, the refined class retains at least the coupled labels of W. In addition, for every filter $K \Rightarrow K' \triangleright P$ of S, some labels may become coupled as the filter matches a pattern $K \mathbin{\&} K''$ in C (for some K'') and produces a rule with pattern $K' \mathbin{\&} K''$. By definition of $\mathit{cl}(\_)$, the new coupled labels are:

$$\mathit{cl}(K' \mathbin{\&} K'') \quad = \quad \mathit{cl}(K') \cup \mathit{cl}(K'') \cup \begin{cases} \emptyset \text{ when } \widehat{K'} = \mathsf{0} \text{ or } \widehat{K''} = \mathsf{0} \\ \mathit{dl}(\widehat{K'}) \cup \mathit{dl}(\widehat{K''}) \text{ otherwise} \end{cases}$$

[2]In Ocaml, objects are kept monomorphic for simplicity, and polymorphic functions are usually defined outside of objects.

[3]We could use more general approximations, e.g., we could discard labels whose types have no free variable as we compute $\widehat{K}$ from K. However, such generalizations complicate type inference, which becomes sensitive to the ordering of type variable instantiations.

The three subsets correspond to the labels that appear in distinct pairs in $\widehat{K'} \times \widehat{K'}$, $\widehat{K''} \times \widehat{K''}$ (with $cl(\widehat{K''}) \subseteq W$), and $\widehat{K'} \times \widehat{K''}$, respectively. We define a safe approximation of the union of $cl(K' \mathbin{\&} K'')$ for all well-typed $K \mathbin{\&} K''$, written $cls(B^W, K \Rightarrow K')$. The definition is by cases:

1. If $\widehat{K'} = 0$, we use $cls(B^W, K \Rightarrow K') = W$.
   (In this case, no new message with arguments is introduced.)

2. If $dl(K) \cap W = \emptyset$ and $\widehat{K} \neq 0$, we use $cls(B^W, K \Rightarrow K') = cl(K')$.
   (In this case, $\widehat{K}$ is a single message and $\widehat{K''}$ is empty.)

3. If $dl(K) \cap W \neq \emptyset$, we use $cls(B^W, K \Rightarrow K') = dl(\widehat{K'}) \cup W$.
   (In this case, all labels in $\widehat{K''}$ already appear in W.)

4. Otherwise ($\widehat{K} = 0$ and $\widehat{K'} \neq 0$), we use $cls(B^W, K \Rightarrow K') = dl(\widehat{K'}) \cup dom(\widehat{B})$.

Note that $cls(B^W, K \Rightarrow K')$ only depends on the domain of B and not on its type assignment.

For example, consider the selective refinement of the class *locked_buff* of Section 4. Informally, we may assume that the type of *buff* is of the form B^W where $dom(B) = dom(\widehat{B}) = \{put, get, Empty, Some, Full, Check, Init\}$ and $W = \{put, get, Empty, Some, Full\}$ (every label of W synchronizes with at least another one). The refinement involves the two rewriting clauses $Init(size) \Rightarrow Init(size) \triangleright z.Free()$ and $0 \Rightarrow Free() \triangleright z.Free()$. By item 2 of the definition of *cls*, we get $cls(B^W, Init(size) \Rightarrow Init(size)) = cl(Init(size)) = \emptyset$, which means that this refinement will not introduce any new synchronization. By item 1 of the definition of *cls*, since $\widehat{Free()}$ is 0, we obtain $cls(B^W, 0 \Rightarrow Free()) = W$ (label *Free* may synchronize with other labels but it cannot exchange values with them). These two independent results may be combined together by taking their union to get an approximation of the type of the whole refinement.

6.4 Type checking processes and classes

The typing judgments are described in Figure 8. They rely on *type environments* A that bind class names c to class type schemes and bind object names x to (external) type schemes σ or to (internal) type schemes $\forall X.B$ with $dom(B) \subseteq \mathcal{F}$. In a given environment A, an object x can have two complementary bindings $x : \forall X.[\rho]$ and $x : \forall X.B$. The binding $x : \forall X.[\rho]$ represents the typing of public labels, $x : \forall X.B$ is the binding of private labels.

Since X is a set of type and row variables, we write $\{\gamma_\alpha/\alpha^{\alpha \in X}\}$ for replacing variables in X with object and row types, correspondingly.

We write $dom(A)$ for the set of names bound in A. We let $A + A'$ be $(A \setminus dom(A')) \cup A'$, where $A \setminus X$ removes from A all the bindings of names in X and let $A + x : \forall X.[\rho], x : \forall X.B$ be $A \setminus \{x\} \cup x : \forall X.[\rho] \cup x : \forall X.B$.

The typing rules appear in figures 9 and 10. Generalization in objects and classes relies on a standard auxiliary definition: $\mathsf{Gen}\ (\rho, B, A)$ is the set of free type variables of ρ or B that are not free in A.

Figure 8: Typing judgments

$A \vdash x : \tau$	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;
$A \vdash P$	the process P is well-typed in environment A;
$A \vdash K :: B$	the pattern or selection pattern K binds variables well-typed in A and joins labels in B.
$A \vdash C :: \zeta(\rho)B^{W,V}$	the class C is well-typed in environment A, declares the labels of B, has coupled labels in W, and has virtual labels V (with $V \subseteq dom(B)$ and $W \subseteq dom(\widehat{B})$).
$A \vdash S :: B^W \Rightarrow B'^{W',V}$	the refinement clauses S are well-typed in environment A, refine patterns with labels in B and coupled labels W into patterns with labels in B', coupled labels in W', and virtual labels V (with $W' \subseteq dom(\widehat{B}) \cup dom(\widehat{B'})$ and $V \subseteq dom(B)$).

Processes. In rule Class, all type variables can be generalized, regardless of synchronization. This is safe because classes are templates for object definitions: the set W in the class type of c is used to restrict polymorphism, but only at object instantiation.

In rule Object, the class C is first typechecked and yields a class type $\zeta(\rho)B^{W,\emptyset}$. The shape of this type excludes virtual labels, thus preventing the instantiation of a partially-defined object. The object type is the restriction of labels declared in B to public ones. The constraint $\rho = B \upharpoonright \mathcal{M}$ checks, for each public label m of B, that the type given to m in B and in ρ are the same. The process Q is typed in an environment extended with the object x bound to a generalized $[\rho]$. The process P is typed as Q, except that P can also use the private labels of x.

Patterns. Typing rules for join patterns check that the patterns are well-formed, collect their typed labels, and check that the environment agrees with received objects.

Classes. 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 `match` C `with` S `end`, out of typings of C and S. It uses the auxiliary judgment for selection clauses $A \vdash S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2}$; This premise will ensure that labels in the selection patterns are all defined in B_1, hence declared in C. On the contrary, the premise $dl(S) \cap dom(B_1) = \emptyset$ implies that names in $dl(S)$ are not already declared in C. In particular, this ensures that the pattern of every refined reaction rule is linear (condition (2) in

Figure 9: Typing rules for names, messages, and processes

Rules for names and messages

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

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

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

Rules for processes

Null
$$A \vdash 0$$

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

Join-Parallel
$$\frac{A \vdash x.M_1 \qquad A \vdash x.M_2}{A \vdash x.(M_1 \ \&\ M_2)}$$

Parallel
$$\frac{A \vdash P \qquad A \vdash Q}{A \vdash P \ \&\ Q}$$

Class
$$\frac{\begin{array}{l} A \vdash C :: \zeta(\rho)B^{W,V} \\ A + c : \forall \mathsf{Gen}\ (\rho, B, A).\zeta(\rho)B^{W,V} \vdash P \end{array}}{A \vdash \mathsf{class}\ c = C\ \mathsf{in}\ P}$$

Object
$$\frac{\begin{array}{ll} A \vdash \mathsf{self}(x)\, C :: \zeta(\rho)B^{W,\emptyset} & \\ A + x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F}) \vdash P & \rho = B \upharpoonright \mathcal{M} \\ A + x : \forall X.[\rho] \vdash Q & X = \mathsf{Gen}\ (\rho, B, A) \setminus ctv(B \upharpoonright W) \end{array}}{A \vdash \mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ Q}$$

Section 3.3).

Refinement clauses. Refinement clauses are typed much like reaction rules and class disjuncts. Rule Modifier types a series of selection clauses and builds a superset of the coupled labels after the refinement, as detailed in Section 6.3. Rule Modifier also 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.

6.5 Type checking solutions

We finally extend typing from programs to chemical solutions. The typing judgment $\vdash (\mathcal{D} \Vdash \mathcal{P})$ states that the chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is well-typed. The auxiliary judgments $A \vdash \mathcal{D} :: A'$ deals with active object definitions. The typing rules appear in Figure 11.

Rule Chemical-Solution uses an additional notation A^ψ. Let N be a set of object names and A be a typing environment of the form

$$A = \bigcup_{x \in N} (x : \sigma_x) \cup \bigcup_{x \in N} (x : \forall Y_x.B_x)$$

Figure 10: Typing rules for patterns, classes, and refinement clauses

Rules for patterns

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

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

Synchronization
$$\frac{A \vdash J_1 :: B_1 \qquad A \vdash J_2 :: B_2}{A \vdash J_1 \;\&\; J_2 :: B_1 \oplus B_2}$$

Alternative
$$\frac{A \vdash J_1 :: B_1 \qquad A \vdash J_2 :: B_2}{A \vdash J_1 \text{ or } J_2 :: B_1 \oplus B_2}$$

Rules for classes

Sub
$$\frac{A \vdash c :: \zeta(\rho)B^{W,V}}{A \vdash c :: \zeta(\rho)B^{W \cup W', V \cup V'}}$$

Class-Var
$$\frac{c : \forall X.\zeta(\rho)B^{W,V} \in A}{A \vdash c :: (\zeta(\rho)B^{W,V})\{\gamma_\alpha / \alpha^{\alpha \in X}\}}$$

Reaction
$$\frac{A' \vdash J :: B \qquad A + A' \vdash P \qquad dom(A') = fn(J)}{A \vdash J \triangleright P :: \zeta(\rho)B^{cl(J),\emptyset}}$$

Self-Binding
$$\frac{A + x : [\rho], x : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V}}{A \vdash \mathsf{self}(x)\, C :: \zeta(\rho)B^{W,V}}$$

Abstract
$$\frac{dom(B) = L}{A \vdash L :: \zeta(\rho)B^{\emptyset,L}}$$

Disjunction
$$\frac{\begin{array}{ll} A \vdash C_1 :: \zeta(\rho)B_1^{W_1,V_1} & V_1' = V_1 \setminus (dom(B_2) \setminus V_2) \\ A \vdash C_2 :: \zeta(\rho)B_2^{W_2,V_2} & V_2' = V_2 \setminus (dom(B_1) \setminus V_1) \end{array}}{A \vdash C_1 \text{ or } C_2 :: \zeta(\rho)(B_1 \oplus B_2)^{W_1 \cup W_2, V_1' \cup V_2'}}$$

Refinement
$$\frac{\begin{array}{ll} A \vdash C :: \zeta(\rho)B_1^{W_1,V_1} & \\ A \vdash S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2} & dl(S) \cap dom(B_1) = \emptyset \end{array}}{A \vdash \text{match } C \text{ with } S \text{ end} :: \zeta(\rho)(B_1 \oplus B_2)^{W_1 \cup W_2, V_1 \cup V_2}}$$

Rules for refinement clauses

Modifier-Clause
$$\frac{\begin{array}{ll} A' \vdash K :: B'' & B'' \subseteq B \\ A' \vdash K' :: B' & dom(A') = fn(K') \\ A + A' \vdash P & W' = cls(B^W, K \Rightarrow K') \end{array}}{A \vdash K \Rightarrow K' \triangleright P :: B^W \Rightarrow B'^{\,W', dl(K) \setminus dl(K')}}$$

Modifier
$$\frac{(A \vdash S_i :: B^W \Rightarrow B_i^{W_i,V_i})^{i \in I}}{A \vdash |^{i \in I} S_i :: B^W \Rightarrow (\oplus^{i \in I} B_i')^{\bigcup_{i \in I} W_i, \bigcup_{i \in I} V_i}}$$

Figure 11: Typing rules for solutions

$$\frac{\begin{array}{l}\text{Chemical-Solution}\\ A = \cup_{\psi x \# D \in \mathcal{D}} A_x \\ (A^{\psi} \vdash D :: A_x)^{\psi x \# D \in \mathcal{D}} \\ (A^{\psi} \vdash P)^{\psi \# P \in \mathcal{P}}\end{array}}{\vdash \mathcal{D} \Vdash \mathcal{P}} \qquad \frac{\begin{array}{l}\text{Definition}\\ \rho = B \upharpoonright \mathcal{M} \\ X = \mathsf{Gen}\ (\rho, B, A) \setminus ctv(B \upharpoonright W) \\ A \vdash \mathsf{self}(x)\ D :: \zeta(\rho) B^{W,\emptyset}\end{array}}{A \vdash D :: x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F})}$$

For any string ψ of names in N, we define the restricted environment:

$$A^{\psi} = \bigcup_{x \in N} (x : \sigma_x) \cup \bigcup_{x \in \psi} (x : \forall Y_x . B_x)$$

Namely, A^{ψ} removes from the environment A the private labels of objects which did not create x. Actually, x cannot access these labels.

Rules Chemical-Solution and Definition are similar to rule Object. The main difference is that, in $A \vdash D :: A'$, the typing environment A' is polymorphic. This allows polymorphic type-checking for solutions.

6.6 Subject reduction with privacy

We are now ready to state our main results on types for the chemical semantics and the class rewriting, respectively. Additional lemmas and the proofs appear in Appendix B.

Theorem 1 (Subject reduction)

1. *Chemical reductions preserve chemical typings:* *if* $\vdash \mathcal{D} \Vdash \mathcal{P}$ *and* $\mathcal{D} \Vdash \mathcal{P} \equiv \mathcal{D}' \Vdash \mathcal{P}'$ *or* $\mathcal{D} \Vdash \mathcal{P} \longrightarrow \mathcal{D}' \Vdash \mathcal{P}'$, *then* $\vdash \mathcal{D}' \Vdash \mathcal{P}'$.
2. *Class rewriting preserve typings: if* $A \vdash P$ *and* $P \longmapsto P'$ *then* $A \vdash P'$.

In combination, any interleaving of chemical reductions and class rewritings preserves chemical typing. Precisely, we can lift class rewriting steps from processes to chemical solutions ($\mathcal{D} \Vdash P, \mathcal{P} \longmapsto \mathcal{D} \Vdash P', \mathcal{P}'$ when $P \longmapsto P'$) and we have that, if $\mathcal{D} \Vdash \mathcal{P}$ is well-typed and $\mathcal{D} \Vdash \mathcal{P}\ (\equiv\!\!\!\rightarrow \cup \longmapsto)^* \ \mathcal{D}' \Vdash \mathcal{P}'$, then $\mathcal{D}' \Vdash \mathcal{P}'$ is also well-typed.

The next theorem guarantees that chemical typing prevents any runtime failure and class rewriting failure, as detailed in Definition 1 (Section 6.1):

Theorem 2 (Safety) *Well-typed chemical solutions do not fail.*

While we do not address type inference in this paper, our type system has been carefully designed to allow type inference. We conjecture that, given a typing environment A and a process P (or a class C), it is decidable whether P (or C) is typable in A; moreover, we conjecture that if C is typable then it has a principal type.

6.7 Example of typing

We infer a type for the class *buffer* of Section 3.1:

$$\begin{array}{l} \mathsf{class}\ \ buffer = \mathsf{self}(z) \\ \qquad get(r)\ \ \&\ Some(a) \triangleright r.reply(a)\ \&\ z.Empty() \\ \quad \mathsf{or}\ \ put(a,r)\ \&\ Empty() \triangleright r.reply()\ \&\ z.Some(a) \end{array}$$

The body of this class definition is of the form $\mathsf{self}(z)\,(J_1 \triangleright P_1$ or $J_2 \triangleright P_2)$. First, consider the typing of pattern J_1. By rules MESSAGE-PATTERN and SYNCHRONIZATION we have:

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

where the type environment A_1 and internal type B_1 are $A_1 = r : \theta_1, a : \theta_2$ and $B_1 = get : (\theta_1); Some : (\theta_2)$, reflecting the presence of labels and their arities.

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 be typed in an environment that subsumes A_1. Moreover, $r.reply(a)$ connects the types for r and a (rule SEND). Thus, we get: $\theta_1 = [reply : (\theta); \varrho]$ (reflecting that r is an object with at least a *reply* label) and $\theta_2 = \theta$. Then, the type variable θ is free in the types of both *Some* and *get* which are joined in the same pattern J_1. Skipping some details, rule REACTION yields the following typing of $J_1 \triangleright P_1$:

$$A \vdash J_1 \triangleright P_1 :: \zeta(\rho) {B_1}^{\{get, Some\}, \emptyset} \tag{2}$$

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:

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

where $B_2 = put : (\theta', [reply : (); \varrho']); Empty : ()$. Note that the set of coupled labels is empty, since pattern J_2 contains only one label of non-zero arity.

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

$$A \vdash J_1 \triangleright P_1 \ \mathsf{or}\ J_2 \triangleright P_2 :: \zeta(\rho) B^{\{get, Some\}, \emptyset} \tag{4}$$

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. (In the general case where a label is declared in several patterns, the "$\oplus$" operator enforces a compatibility check on label types.) Hence $B = get : ([reply : (\theta); \varrho]); Some : (\theta); put : (\theta', [reply : (); \varrho']); Empty : ()$.

Rule SELF-BINDING implies that environment A contains two bindings for the self name z, namely, $z : [\rho]$ and $z : B_\rho$. The internal type of z, B_ρ is the restriction of B to private labels, i.e. we get $B_\rho = Some : (\theta); Empty : ()$.

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:

$$\ldots a : \theta', z : (Some : (\theta), \ldots) \vdash P_2 \tag{5}$$

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, $\theta = \theta'$. Hence, the type for class variable *buffer* finally is:

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

where B is as before (after equating θ and θ'). That is: $B = get : ([reply : (\theta); \varrho]); Some : (\theta); put : (\theta, [reply : (); \varrho']); Empty : ()$.

Observe that, according to CLASS, all the type variables (ϱ, ϱ' and θ) are generalized. As a consequence the type for class *buffer* is as polymorphic as it can be. Also observe that the public type ρ is yet unconstrained.

Nevertheless, polymorphism will be restricted and ρ will be made precise while creating objects from class *buffer* (rule OBJECT).

7 Related and future works

The design and implementation of concurrent object-oriented languages, *e.g.* [2, 33, 1, 6], has recently prompted the investigation of the theoretical foundations of concurrent objects. Several works provide encodings of objects in process calculi [31, 28, 14, 8, 19] or, conversely, supplement objects with concurrent primitives [23, 4, 13, 29]. These works promote a unified framework for reasoning about objects and processes, but they do not address the incremental definition of concurrent objects or its typechecking. (When considered, inheritance is treated as in a sequential language and does not deal with synchronization.) In particular, the TyCO language, developed by Vasconcelos and his collegues, relies on a core calculus which is very close to the objective join calculus in Section 2, with few differences due to name definitions. However the two languages differ when classes are considered. In particular, TyCO [29] only allows simple forms of inheritance, namely extensions of classes with new methods, and updates of old methods with new ones. No mechanism is provided to access methods in the super class and, therefore, to reuse their bodies. As regards the type system, both the languages stick to a predicative polymorphism discipline. However, in our calculus, polymorphism originates from class and obj operations and regards objects and classes as a whole; in TyCO polymorphism originates from a different operator—the let—which allows to define one (polymorphic) function at a time, and regards single methods.

The addition of classes to the join calculus enables a modular definition of synchronization. Different receivers for the same labels can thus be introduced at different syntactic positions in a program. In that respect, we partially recover the ability of the pi calculus to dynamically introduce receivers on channels [21]. However, our layered design confines this modularity to classes, which are resolved at compile time. From a programming-language viewpoint, this strikes a good balance between flexibility and simplicity, and does not preclude type inference or the efficient compilation of synchronization [16].

Odersky independently proposed an object-oriented extension of the join calculus [24, 25]. As in Section 2, they use join patterns to define objects and synchronization between labeled messages. The main difference lies in the encapsulation of methods within objects. In our proposal, a definition binds a single object, with all the labels appearing in the definition, and we rely on types to hide some of those labels as private. In their proposal, a definition may bind any number of objects, and each object explicitly collects some of the declared labels as its methods. As a result, a label that is not collected remains syntactically private. Besides, their synchronization patterns can express matching on the values carried in messages (strings, integers, lists, trees, *etc.*) rather than matching on just the message labels. For instance, a rule

$\ell(h :: t) \triangleright P$ reacts provided ℓ carries a non-empty list. Those design decisions may lead to different implementation strategies. However, they do not deeply affect typing.

As regards polymorphism, our generalization rule OBJECT corresponds to the one currently implemented in JoCaml [15]. It is more expressive than the generalization rule initially proposed in [12] and seems equivalent to the generalization rule of [7]. In [12], non-generalizable type variables were computed altogether for all clauses of a definition, which may be too conservative. In both [15] and [7], non-generalizable type variables are computed one rule at a time, which is more precise. This latter approach is natural in our setting, since recursion is left open till object instantiation. In ML, this amounts to typing let rec $x_1 = a_1$ and $x_2 = a_2$ in a as let $x =$ fix $(\lambda(x_1, x_2).(a_1, a_2))$ in a. The latter term separates type-checking the body of the recursion from type-checking recursion itself.

Our language supports multiple inheritance of classes, but not mixin inheritance [5], which amounts to parameterize classes by other classes. As in OCaml, we can only combine existing classes, but obtain mixin inheritance indirectly through modules and functors.

In sequential languages, deep method renaming, *i.e.* rewriting of recursive calls or method hiding, can be expressed using dictionaries [27] or views [30]. In concurrent languages, views offer additional benefits. For example, one can duplicate the synchronization patterns of a superclass by inheriting several copies of the class, independently refine their synchronization, and use different views to access the copies. For instance, one could distinguish internal and external views in the last example of Section 4. The integration of views in the objective join calculus deserves further investigation.

Since classes are just object templates, our typing system allows polymorphic variables in class types, and defer any monomorphic restriction till object instantiation. For type safety, one must check that, in every join pattern of an object, any variable occurring in the type of several labels is monomorphic. To this aim, our class types collect a superset W of these coupled labels, but other approaches are possible. A plain solution is to assume that all labels are coupled. Then, class types don't convey any synchronization information, and generalization is as in [12]. Conversely, the class types could detail the labels of each join pattern. This would allow us to detect refinement errors at compile time. However, the resulting types would be very precise, and we would also need some form of subtyping to get rid of excessive information. This is another promising direction for research.

8 Conclusion

We have designed 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. In particular, our operators for inheritance can express transformations on the parent class, according to its synchronization patterns. We motivated our design choices using standard, problematic examples that mix inheritance and synchronization. We gave operational semantics for objects and classes, and a type system that prevents standard errors and also enforces privacy.

Acknowledgments. This work benefited from fruitful discussions with Sylvain Conchon, Fabrice Le Fessant, and François Pottier.

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] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for $c^\sharp$. In *Informal Proceedings of the FOOL 9 workshop.* ACM, January 2002.

[4] 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.

[5] G. Bracha and W. Cook. Mixin-Based Inheritance. In N. Meyrowitz, editor, *European Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 303–311, Otawa, 1990. ACM press.

[6] L. Cardelli. Obliq A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

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

[8] 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.

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

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

[11] 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.

[12] 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.

[13] A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In Nestmann and Pierce [22].

[14] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*. North-Holland, 1998.

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

[16] F. Le Fessant and L. Maranget. Compiling join-patterns. In Nestmann and Pierce [22].

[17] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation and user's manual - release 3.04. Technical report, INRIA, December 2001. Documentation distributed with the Objective Caml system.

[18] 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.

[19] M. Merro, J. Kleist, and U. Nestmann. Mobile objects as mobile processes. *Information and Computation*, To appear.

[20] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. M. Nierstrasz, editor, *ECOOP'93*, number 707 in LNCS, pages 220–246. SRI International, Springer Verlag, 1993.

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

[22] U. Nestmann and B. C. Pierce, editors. *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*, Nice, France, Sept. 1998.

[23] 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.

[24] M. Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 1–25. Springer Verlag, Mar. 2000.

[25] M. Odersky. An overview of functional nets. In *APPSEM Summer School, Caminha, Portugal*, LNCS. Springer Verlag, Sept. 2000. To appear.

[26] 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`.

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

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

[29] V. T. Vasconcelos. Typed concurrent objects. In *8th* , volume 821 of *Lecture Notes in Computer Science*, pages 100–117, July 1994.

[30] J. Vouillon. Combining subsumption and binary methods: An object calculus with views. In *Proceedings of POPL '01*, Jan. 2001.

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

[32] M. Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[33] 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 Cross-encodings to the join calculus

In the join calculus of [10], each definition binds one or several *channel names* that can be passed independently whereas, in the objective join calculus of Section 2, each definitions binds a single object. This difference is not very deep; we briefly present cross-encodings between these two variants. We recall a syntax for the join calculus:

$$\begin{array}{rcl} P & ::= & \mathsf{0} \mid x(\widetilde{u}) \mid P \mathbin{\&} P \mid \mathsf{def}\ D\ \mathsf{in}\ P \\ D & ::= & M \triangleright P \mid D\ \mathsf{or}\ D \\ M & ::= & x(\widetilde{u}) \mid M \mathbin{\&} M \end{array}$$

Join calculus processes can be encoded by introducing single-label forwarder objects and passing those object names instead of channel names. (The encoding given below works for non-recursive definitions in the join calculus; recursive definitions can easily be eliminated beforehand [9].)

$$\begin{array}{rcl} [\![\mathsf{0}]\!] & = & \mathsf{0} \\ [\![x(\widetilde{u})]\!] & = & x.send(\widetilde{u}) \\ [\![P_1 \mathbin{\&} P_2]\!] & = & [\![P_1]\!] \mathbin{\&} [\![P_2]\!] \\ [\![\mathsf{def}\ D\ \mathsf{in}\ P]\!] & = & \mathsf{obj}\ o = [\![D]\!]\ \mathsf{in} \\ & & \mathsf{obj}\ x_1 = send(\widetilde{u}_1) \triangleright o.m_1(\widetilde{u}_1)\ \mathsf{in}\ \ldots \\ & & \mathsf{obj}\ x_n = send(\widetilde{u}_n) \triangleright o.m_n(\widetilde{u}_n)\ \mathsf{in}\ [\![P]\!] \end{array}$$

where we assume that D defines the channel names $x_1, \ldots, x_n$. Accordingly, we encode channel reaction rules and patterns as follows:

$$\begin{array}{rclcrcl} [\![D_1\ \mathsf{or}\ D_2]\!] & = & [\![D_1]\!]\ \mathsf{or}\ [\![D_2]\!] & \quad & [\![M_1 \mathbin{\&} M_2]\!]^- & = & [\![M_1]\!]^- \mathbin{\&} [\![M_2]\!]^- \\ [\![M \triangleright P]\!] & = & [\![M]\!]^- \triangleright [\![P]\!] & & [\![x(\widetilde{u})]\!]^- & = & m_x(\widetilde{u}) \end{array}$$

Conversely, one can encode objective join processes into a join calculus enriched with records. Join calculus values then consist of both names and records, written $\{\ell_i = x_i\}^{i\in I}$; we use $_\#$ for record projection. The encoding substitutes explicit records of channels for defined objects.

$$\begin{array}{rcl}
[\![0]\!] & = & 0 \\
[\![x.\ell(\widetilde{u})]\!] & = & x_\#\ell(\widetilde{u}) \\
[\![P_1 \mathbin{\&} P_2]\!] & = & [\![P_1]\!] \mathbin{\&} [\![P_2]\!] \\
[\![\mathsf{obj}\ x = D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2]\!] & = & (\mathsf{def}\ [\![D]\!]_x\ \mathsf{in}\ [\![P_1 \mathbin{\&} P_2]\!])\{\{\ell = x_\ell, \ell \in dl(D)\}/x\}
\end{array}$$

$$\begin{array}{rclcrcl}
[\![D_1\ \mathsf{or}\ D_2]\!]_x & = & [\![D_1]\!]_x\ \mathsf{or}\ [\![D_2]\!]_x & \qquad & [\![M_1 \mathbin{\&} M_2]\!]_x & = & [\![M_1]\!]_x \mathbin{\&} [\![M_2]\!]_x \\
[\![M \triangleright P]\!]_x & = & [\![M]\!]_x \triangleright [\![P]\!] & & [\![\ell(\widetilde{u})]\!]_x & = & x_\ell(\widetilde{u})
\end{array}$$

The encoding above treats all methods as public; it can be refined to preserve the scope of private labels using two records of channels instead of a single one.

B Proofs for typing

B.1 Basic properties

In the following lemmas, we let Δ range over any right hand side of a judgment, except for the chemical judgment.

Lemma 2 (Useless variable) *For any judgment of the form $A \vdash \Delta$, and any name x that is not free in Δ we have:*

$$A \vdash \Delta \Leftrightarrow A + A' \vdash \Delta$$

where A' is either $x : \sigma$ or $x : \sigma, x : \forall X.B$.

Lemma 3 (Renaming of type variables) *Let η be a substitution on type variables. We have:*

$$A \vdash \Delta \Rightarrow \eta(A) \vdash \eta(\Delta)$$

We say that a type scheme $\forall X.\tau$ is *more general* than $\forall X'.\tau'$ if τ' is of the form $\eta(\tau)$ for some substitution η replacing type and row variables by types and rows, respectively. This notion is also lifted to set of assumptions as follows: A' is more general than A if A and A' have the same domain and for each u in their domain, $A'(u)$ is more general than $A(u)$.

Lemma 4 (Generalization) *If $A \vdash \Delta$ and A' is more general than A, then $A' \vdash \Delta$.*

Lemma 5 (Substitution of a name in a term) *If $A + x : \tau \vdash \Delta$ and $A \vdash u : \tau$ then $A \vdash \Delta\{u/x\}$. Similarly, if $A + x : [m_i : \widetilde{\tau}_i{}^{i\in I}], x : (\ell : \widetilde{\tau}_\ell{}^{\ell\in S}) \vdash \Delta$ and $A \vdash y : \tau$, $(A \vdash y.m_i : \widetilde{\tau}_i)^{i\in I}$, and $(A \vdash y.\ell : \widetilde{\tau}_\ell)^{\ell\in S}$ then $A \vdash \Delta\{y/x\}$.*

Lemma 6 (Substitution of a class name in a term)
Let $A + c : \forall X.\zeta(\rho)B^{W,V} \vdash \Delta$ and $A + B \vdash C :: \zeta(\rho)B^{W,V}$ and $X \subseteq \mathsf{Gen}\ (\rho, B, A)$. Then $A \vdash \Delta\{C/c\}$.

B.2 Subject reduction for the chemical semantics (Theorem 1.1)

PROOF Let $\vdash (\mathcal{D} \Vdash \mathcal{P})$ and $\mathcal{D} \Vdash \mathcal{P} \Longrightarrow \mathcal{D}' \Vdash \mathcal{P}'$. We demonstrate that $\vdash (\mathcal{D}' \Vdash \mathcal{P}')$. According to the rules in Figure 6, there are two cases for the proof of $\mathcal{D} \Vdash \mathcal{P} \Longrightarrow \mathcal{D}' \Vdash \mathcal{P}'$:

1. an instance of rule OBJ, followed by a sequence of CHEMISTRY-OBJ;
2. an instance of one of the rules PAR, NIL, PUBLIC-COMM, PRIVATE-COMM, RED, followed by a sequence of CHEMISTRY.

We discuss the two cases separately.

Case 1. The reduction is $\mathcal{D} \Vdash \psi \,\#\, \mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q, \mathcal{P} \equiv \mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, P, \psi \,\#\, Q, \mathcal{P}$, where $x \notin \mathit{fn}(\mathcal{D}) \cup \mathit{fn}(\mathcal{P})$.

On the one hand, if $\vdash (\mathcal{D} \Vdash \psi \,\#\, \mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q, \mathcal{P})$ (1) and rule CHEMICAL-SOLUTION, we obtain $(A^{\varphi} \vdash D' :: A_z)^{\varphi z \,\#\, D' \in \mathcal{D}}$ (2), $A^{\psi} \vdash \mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ and $(A^{\varphi} \vdash P')^{\varphi \,\#\, P' \in \mathcal{P}}$ (3), where $A = \cup_{\varphi z \,\#\, D' \in \mathcal{D}} A_z$. A derivation of $A^{\psi} \vdash \mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ must have the form:

$$\textsc{Object}\ \dfrac{\begin{array}{c} A^{\psi} \vdash \mathsf{self}(x)\, D :: \zeta(\rho) B^{W,\emptyset}\ (4) \\ X = \mathsf{Gen}\ (\rho, B, A^{\psi}) \setminus ctv(B \upharpoonright W)\ (5) \\ A^{\psi} + x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F}) \vdash P\ (7) \\ \rho = B \upharpoonright \mathcal{M}\ (8) \qquad A^{\psi} + x : \forall X.[\rho] \vdash Q\ (9) \end{array}}{A^{\psi} \vdash \mathsf{obj}\ x = D\ \mathsf{init}\ P\ \mathsf{in}\ Q}$$

On the other hand, if $\vdash (\mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, P, \psi \,\#\, Q, \mathcal{P})$ (10) and rule CHEMICAL-SOLUTION, we obtain $(A'^{\varphi} \vdash D' :: A'_z)^{\varphi z \,\#\, D' \in \mathcal{D}}$ (11), $A'^{\psi} \vdash D :: A'_x$, $A'^{\psi x} \vdash P$ (12), $A'^{\psi} \vdash Q$ (13) and $(A'^{\varphi} \vdash P')^{\varphi \,\#\, P' \in \mathcal{P}}$ (14) where $A' = (\cup_{\varphi z \,\#\, D' \in \mathcal{D}} A'_z) \cup A'_x$. A derivation of $A'^{\psi} \vdash D :: A'_x$ must have the form:

$$\dfrac{\begin{array}{c} A'^{\psi} \vdash \mathsf{self}(x)\, D :: \zeta(\rho') B'^{W',\emptyset}\ (15) \qquad \rho' = B' \upharpoonright \mathcal{M}\ (16) \\ X' = \mathsf{Gen}\ (\rho', B', A'^{\psi}) \setminus ctv(B' \upharpoonright W')\ (17) \end{array}}{A'^{\psi} \vdash D :: A'_x}\ \textsc{Definition}$$

where $A'_x = x : \forall X'.[\rho] \oplus x : \forall X'.(B' \upharpoonright \mathcal{F})$.

Note that, by definition, A does not bind x because $x \notin \mathit{fn}(\mathcal{D})$. Therefore, by Lemma 2, if either (1) or (10) hold, then we can always choose A or A' such that $A' = A + A'_x$, the judgments (2) and (11) be equivalent, as well as the judgments (3) and (14). We now focus on the other judgments.

Then, since $A'^{\psi} = A^{\psi} + x : \forall X'.\,[\rho']$, the following premises can be made equivalent to one another:

(4) $\equiv$ (15): By Lemma 2 and identifying B' with B, ρ' with ρ, and W' with W.

(5) $\equiv$ (17): By definition, we now have

$$X = (ftv(B) \cup ftv(\rho)) \setminus (ftv(A^{\psi}) \cup ctv(B \upharpoonright W))$$

$$X' = (ftv(B) \cup ftv(\rho)) \setminus (ftv(A'^{\psi}) \cup ctv(B \upharpoonright W))$$

Since ${A'}^{\psi}$ is equal to $A^{\psi} + x : \forall X.\rho$ and A^{ψ} does not bind x, we have $X' \subseteq X$. Conversely, we have $X \setminus X' \subseteq \mathit{ftv}(\forall X.\rho)$. Since by definition $\mathit{ftv}(\forall X.\rho)$ does not intersect X, it follows that $X \setminus X'$ is empty, thus $X \subseteq X'$.

(7) $\equiv$ (12): Using the previous equivalences, we now have $A_x = A'_x$. Therefore, ${A'}^{\psi x}$ is equal to $A^{\psi x}$.

(9) $\equiv$ (13): Same reasoning as the previous case.

(8) $\equiv$ (16): The two equalities are now identical.

To conclude, if (1) holds, then (10) holds by taking $A + x : \forall X.[\rho], x : \forall X.(B \restriction \mathcal{F})$ for A', and conversely, if (10) holds, then (1) holds by taking $A' \setminus x$ for A.

Case 2. The reduction is $\mathcal{D} \Vdash \mathcal{P}_1, \mathcal{P} \rightleftharpoons \mathcal{D} \Vdash \mathcal{P}_2, \mathcal{P}$. There are several subcases, according to the leaf node in the proof tree.

Case Nil. The equivalence is $\mathcal{D} \Vdash \psi \,\#\, 0, \mathcal{P} \;\equiv\; \mathcal{D} \Vdash \mathcal{P}$. Indeed the judgment $A \vdash \psi \,\#\, 0$ is always true, for any environment A.

Case Par. The equivalence is $\mathcal{D} \Vdash \psi \,\#\, (P \,\&\, Q), \mathcal{P} \equiv \mathcal{D} \Vdash \psi \,\#\, P, \psi \,\#\, Q, \mathcal{P}$. We apply rule Chemical-Solution, then it suffices to prove that the two judgments $A^{\psi} \vdash P \,\&\, Q$ and $A^{\psi} \vdash P$, $A^{\psi} \vdash Q$ are equivalent. This follows by rules Parallel and Chemical-Solution.

Case Join. This is similar to the above case, and relies on rules Chemical-Solution, Parallel and Join-Parallel.

Case Public-Comm. The reduction is $\mathcal{D}, \psi x \,\#\, D \Vdash \psi' \,\#\, x.m(\widetilde{u}), \mathcal{P} \longrightarrow \mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, x.m(\widetilde{u}), \mathcal{P}$. Let us assume that $A^{\psi} \vdash D :: A_x$ and $A^{\psi'} \vdash x.m(\widetilde{u})$ (1). We must show that $A^{\psi x} \vdash x.m({u_i}^{i \in I})$ where $\widetilde{u} = {u_i}^{i \in I}$. A complete derivation of (1) must be of the form:

$$\dfrac{\dfrac{\dfrac{\cdots}{A^{\psi'} \vdash x : [m : ({\tau_i}^{i\in I}); \rho]}\,\textsc{Object-Var}}{A^{\psi'} \vdash x.m : ({\tau_i}^{i \in I})}\,\textsc{Message} \qquad \left(\dfrac{\cdots}{A^{\psi'} \vdash u_i : \tau_i}\,\textsc{Object-Var}\right)^{i \in I}}{A^{\psi'} \vdash x.m({u_i}^{i\in I})}\,\textsc{Send}$$

This derivation, which does not use any internal type assumption of A (any Private-Message rule), is not affected by replacing $A^{\psi'}$ with $A^{\psi x}$. Thus, $A^{\psi x} \vdash x.m({u_i}^{i \in I})$ holds.

Case Private-Comm. The reduction is $\mathcal{D}, \psi x \,\#\, D \Vdash \psi x \psi' \,\#\, x.f(\widetilde{u}), \mathcal{P} \longrightarrow \mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, x.f(\widetilde{u}), \mathcal{P}$. Let us assume that $A^{\psi} \vdash D :: A_x$ and $A^{\psi x \psi'} \vdash x.f(\widetilde{u})$ (1). We must show that $A^{\psi x} \vdash x.f({u_i}^{i \in I})$ where $\widetilde{u} = {u_i}^{i \in I}$. A complete derivation of (1) must be of the form:

$$\dfrac{\dfrac{x : \forall X.(f : ({\tau_i}^{i \in I}); B) \in A^{\psi x \psi'}}{A^{\psi x \psi'} \vdash x.f : ({\tau_i\{\gamma_\alpha/\alpha^{\alpha \in X}\}}^{i \in I})}\,\textsc{Private-Message} \qquad \left(\dfrac{\cdots}{A^{\psi x \psi'} \vdash u_i : \tau_i\{\gamma_\alpha/\alpha^{\alpha \in X}\}}\,\textsc{Object-Var}\right)^{i \in I}}{A^{\psi x \psi'} \vdash x.f({u_i}^{i \in I})}\,\textsc{Send}$$

Note that, the only internal type assumption in the premise is on x, which remains in $A^{\psi x}$. Thus, as in case Public-Comm, we can replace $A^{\psi x \psi'}$ by $A^{\psi x}$ in this derivation and conclude that $A^{\psi x} \vdash x.f({u_i}^{i \in I})$.

Case Red. The reduction is $\mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, x.(M\sigma), \mathcal{P} \longrightarrow \mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, (P\sigma), \mathcal{P}$, where D is of the form $M \triangleright P$ or D' and M is of the form $\&_{i \in I}\ \ell_i(\widetilde{x}_i)$. A derivation of $\vdash (\mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, x.(M\sigma), \mathcal{P})$ must follow from rule Chemical-Solution with the following premises:

$$\begin{array}{cl} A = (\cup_{\varphi z \,\#\, D'' \in \mathcal{D}} A_z) \cup A_x & (1) \\ (A^{\varphi} \vdash D'' :: A_z)^{\varphi z \,\#\, D'' \in \mathcal{D}} & (2) \\ A^{\psi} \vdash D :: A_x & (3) \\ A^{\psi x} \vdash x.(M\sigma) & (4) \\ (A^{\varphi} \vdash P')^{\varphi \,\#\, P' \in \mathcal{P}} & (5) \end{array}$$

The derivation of (3) must end with rule Definition with the premises

$$A_x = x : \forall X.[\rho], x : \forall X.(B \restriction \mathcal{F}) \tag{6}$$
$$\rho = B \restriction \mathcal{M} \tag{7}$$
$$X = \mathsf{Gen}\ (\rho, B, A^{\psi}) \setminus ctv(B \restriction W) \tag{8}$$
$$A^{\psi} \vdash \mathsf{self}(x)\, D :: \zeta(\rho) B^{W,\emptyset} \tag{9}$$

To demonstrate $\vdash (\mathcal{D}, \psi x \,\#\, D \Vdash \psi x \,\#\, (P\sigma), \mathcal{P})$ it suffices to show that $A^{\psi} \vdash P\sigma$. Note that $A^{\psi x} = A^{\psi} + A_x$ follows from (1). The derivation of (4) must end with a rule Join-Parallel. Hence, for each message $\ell_i(\sigma(\widetilde{x}_i))$ of $M\sigma$, we have $A^{\psi} + A_x \vdash x.\ell_i(\sigma(\widetilde{x}_i))$. In turn, this must be derived by rule Send with the premises $A^{\psi} + A_x \vdash x.\ell_i : \widetilde{\tau}_i$ (10) and $A^{\psi} + A_x \vdash \sigma(\widetilde{x}_i) : \widetilde{\tau}_i$ (11). The judgment (10) is derived by either rule Private-Message or Message, depending on whether ℓ is private or public. In both cases, each tuple $\widetilde{\tau}_i$ is an instance of the generic type $\forall X.B(\ell_i)$, with a substitution η_i of domain in $X \cap ftv(B(\ell_i))$.

The derivation of (9) must contain a sub-derivation

$$\dfrac{\dfrac{\text{Reaction}\quad \begin{array}{c} A' \vdash M :: B_0 \ (12) \\ A^{\psi} + x : [\rho], x : (B \restriction \mathcal{F}) + A' \vdash P \ (13) \qquad dom(A') = fn(M) \end{array}}{\text{Sub}\quad A^{\psi} + x : [\rho], x : (B \restriction \mathcal{F}) \vdash M \triangleright P :: \zeta(\rho) B_0^{cl(M),\emptyset}}}{A^{\psi} + x : [\rho], x : (B \restriction \mathcal{F}) \vdash M \triangleright P :: \zeta(\rho) B_0^{W,\emptyset} \ (14)}$$

where $cl(M) \subseteq W$ (15) (the judgment (14) is then combined with other judgments for other reaction rules of D by a sequence of Disjunction rules, followed by a rule Self, to end up with (9)). The internal type B_0 is therefore a subset of B. As regards the premise (12), it is derived by a combination of rules Synchronization, Message-Pattern, and Empty-Pattern. Hence, we have:

$$A' \vdash \ell_i(\widetilde{x}_i) :: \ell_i : B(\ell_i) \ (16)$$

for every $i \in I$. By (15) and the definitions of $cl(B)$ and $ctv(B \restriction W)$, we have $ftv(B(\ell_i)) \cap ftv(B(\ell_j)) \subseteq ctv(B \restriction W)$, hence by (8), $ftv(B(\ell_i)) \cap ftv(B(\ell_j)) \cap X = \emptyset$, for every distinct i and j in I. Thus, the sets $(X \cap ftv(B(\ell_i)))^{i \in I}$, *i.e.* $dom(\eta_i)^{i \in I}$ form a partition of X. Let η be the sum of substitutions $\oplus^{i \in I} \eta_i$.

Observe that the domain of η is included in X and is thus disjoint from free type variables of A^ψ (17).

Applying Lemma 3 to (13), we have $\eta(A^\psi + x : [\rho], x : (B \upharpoonright \mathcal{F})) + \eta(A') \vdash P$, that is, $A^\psi + x : [\eta(\rho)], x : \eta(B \upharpoonright \mathcal{F}) + \eta(A') \vdash P$. By Lemma 4, we can let x be used polymorphically, *i.e.* $A^\psi + A_x + \eta(A') \vdash P$. Similarly, we have $(A^\psi + A_x \vdash \sigma(x_i) : \eta(\tau_i'))^{x_i:\tau_i' \in A'}$ by Lemmas 3 and 4 successively applied to the collection of judgments (11). Thus, by Lemma 5, we derive $A^\psi + A_x \vdash P\sigma$. $\square$

B.3 Subject reduction for the rewriting semantics (Theorem 1.2)

We first show a couple of properties relating typing and the set of declared labels.

Lemma 7 *For any class C such that $A \vdash C : \zeta(\rho)B^{V,W}$, then $dl(C) = dom(B)$.*

The proof of this lemma is omitted because it is a straighforward induction on the depth of $A \vdash C : \zeta(\rho)B^{V,W}$.

Lemma 8 *For any selective refinement clauses S such that $A \vdash S :: B^W \Rightarrow {B'}^{W',V'}$, we have $dom(B') \setminus dom(B) \subseteq dl(S)$.*

PROOF Selective refinement clauses can always be written as $|_{i\in I}\ K_i \Rightarrow K_i' \triangleright P$. A proof of $A \vdash S :: B^W \Rightarrow {B'}^{W',V'}$ can only end with rule MODIFIER in which the derivation of each premise ends with a rule MODIFIER-CLAUSE. Hence, we have at least the judgments

$$A_i \vdash K_i :: B_i \tag{18}$$
$$B_i \subseteq B \tag{19}$$
$$A_i \vdash K_i' :: B_i' \tag{20}$$
$$B' = \oplus^{i \in i} B_i' \tag{21}$$

Hence,

$$\begin{aligned}
dom(B') \setminus dom(B) &\subseteq dom(B') \setminus (\textstyle\bigcup^{i\in I} dom(B_i)) && \text{by (19)}\\
&= (\textstyle\bigcup^{i\in I} dom(B_i')) \setminus (\textstyle\bigcup^{i\in I} dom(B_i)) && \text{by (21)}\\
&= \textstyle\bigcup^{i\in I} (dom(B_i') \setminus (\textstyle\bigcup^{j\in I} dom(B_j)))\\
&\subseteq \textstyle\bigcup^{i\in I} (dom(B_i') \setminus dom(B_i))\\
&= \textstyle\bigcup^{i\in I} (dl(K_i') \setminus dl(K_i))\\
&= dl(S)
\end{aligned}$$

$\square$

We now show that filter rewriting $\longmapsto$ preserves typing. We denote with $B \setminus L$ the set of pairs $\ell : \tilde{\tau}$ that belongs to B and such that $\ell \notin L$.

Lemma 9 (Filter rewriting) *If all the following conditions hold*

$$C \text{ with } S \longmapsto C' \qquad A \vdash C :: \zeta(\rho)B^{W,V} \qquad A \vdash S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2}$$
$$dl(S) \cap dom(B_1) = \emptyset \qquad B \subseteq B_1 \qquad B_2 \upharpoonright dom(B_1) \subseteq B_1$$

then $A \vdash C' :: \zeta(\rho)(B \oplus (B_2 \setminus L))^{W',V'}$ for some $W' \subseteq W \cup (W_2 \cap dom(B \oplus (B_2 \setminus L)))$, $V' \subseteq V \cup (V_2 \cap (dom(B_1) \setminus L))$, and $L \subseteq (dl(S) \setminus dl(C')) \cup (dom(B_1) \setminus dom(B))$.

<u>PROOF</u> In this proof, we abbreviate $dom(B)$ by $\overline{B}$, for sake of conciseness.

Basic cases.

Case FILTER-APPLY. Let us assume that

$$\begin{aligned}
&K_1 \mathbin{\&} K \triangleright P \text{ with } K_1 \Rightarrow K_2 \triangleright Q \mid S \\
&\qquad \longmapsto K_2 \mathbin{\&} K \triangleright P \mathbin{\&} Q \text{ or } dl(K_1) \setminus dl(K_2) && (1)\\
&A \vdash K_1 \mathbin{\&} K \triangleright P :: \zeta(\rho) B^{W,V} && (2)\\
&A \vdash K_1 \Rightarrow K_2 \triangleright Q \mid S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2} && (3)\\
&((dl(K_2) \setminus dl(K_1)) \cup dl(S)) \cap \overline{B_1} = \emptyset && (4)\\
&B \subseteq B_1 && (5)\\
&B_2 \upharpoonright \overline{B_1} \subseteq B_1 && (6)
\end{aligned}$$

The judgments (2) and (3) are respectively derived by

$$\dfrac{\dfrac{\dfrac{A' \vdash K_1 :: B_1' \ (7) \qquad A' \vdash K :: B_0 \ (8)}{A' \vdash K_1 \mathbin{\&} K :: B_1' \oplus B_0 \ (9)}\ \textsc{Synchronization} \quad A + A' \vdash P \ (10) \qquad \overline{A'} = fn(K_1 \mathbin{\&} K) \ (11)}{A \vdash K_1 \mathbin{\&} K \triangleright P :: \zeta(\rho)(B_1' \oplus B_0)^{cl(K_1 \& K),\emptyset}}\ \textsc{Reaction}}{A \vdash K_1 \mathbin{\&} K \triangleright P :: \zeta(\rho)(B_1' \oplus B_0)^{W,V}}\ \textsc{Sub}$$

and,

$$\dfrac{\dfrac{\begin{array}{ll} A'' \vdash K_1 :: B_1' \ (12) & B_1' \subseteq B_1 \ (13) \\ A'' \vdash K_2 :: B_2' \ (14) & \overline{A''} = fn(K_2) \ (16) \\ A + A'' \vdash Q \ (15) & W_2' = cls(B_1^{W_1}, K_1 \Rightarrow K_2) \end{array}}{A \vdash K_1 \Rightarrow K_2 \triangleright Q :: B_1^{W_1} \Rightarrow B_2'^{W_2', dl(K_1) \setminus dl(K_2)}}\ \textsc{Modifier-Clause} \quad \cdots}{A \vdash K_1 \Rightarrow K_2 \triangleright Q \mid S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2}}\ \textsc{Modifier} + \textsc{Sub}$$

where $B = B_1' \oplus B_0$ (17), $cl(K_1 \mathbin{\&} K) \subseteq W$ (18), $B_2' \subseteq B_2$ (19), $W_2' \subseteq W_2$ and $dl(K_1) \setminus dl(K_2) \subseteq V_2$ (20).

From (7) and (12), A' and A'' coincide on $fn(K_1)$ because they assign the same types to $fn(K_1)$. Moreover, due to the scope rules of reaction rules and the selection operator, we can safely assume that $\overline{A'} \cap \overline{A''} = fn(K_1)$. Thus, by Lemma 2 applied to (8), and (14) we derive $A' + A'' \vdash K :: B_0$ (21) and $A' + A'' \vdash K_2 :: B_2'$ (22). Similarly, by lemma 2 applied to (10) and (15), we also derive $A + A' + A'' \vdash P$ (23) and $A + A' + A'' \vdash Q$ (24).

Foremost we prove the linearity of $K_2 \mathbin{\&} K$. Notice that $dl(K_2) = (dl(K_2) \setminus dl(K_1)) \cup (dl(K_2) \cap dl(K_1))$ and both left and right hand-sides of the $\cup$ have an empty intersection with $dl(K)$. This follows from (4) and from the linearity of $K_1 \mathbin{\&} K$.

By rule SYNCHRONIZATION with premises (22) and (21), we derive $A' + A'' \vdash K_2 \mathbin{\&} K :: B_2' \oplus B_0$ (25). Also, combining the judgments (23) and (24) yields $A + A' + A'' \vdash P \mathbin{\&} Q$ (26) using rule PARALLEL. By premises (11) and (16) we have $\overline{A'} \cup \overline{A''} = fn(K) \cup fn(K_1) \cup fn(K_2)$. By (16) and (12), we have $fn(K_1) \subseteq fn(K_2)$. Hence $\overline{A' + A''} = fn(K_2 \mathbin{\&} K)$ (27). Therefore, by REACTION with premises (25), (26), and (27) we derive

$$A \vdash K_2 \mathbin{\&} K \triangleright P \mathbin{\&} Q :: \zeta(\rho)(B_2' \oplus B_0)^{cl(K_2 \& K),\emptyset} \qquad (28).$$

By rule Abstract, we also deduce

$$A \vdash dl(K_1) \setminus dl(K_2) :: \zeta(\rho) B_1''^{\emptyset, dl(K_1)\setminus dl(K_2)} \tag{29}$$

where $B_1'' = B_1' \setminus dl(K_2)$ (30). Hence, Disjunction allows to derive:

$$A \vdash K_2 \mathbin{\&} K \triangleright P \mathbin{\&} Q \text{ or } L' :: \zeta(\rho)(B_2' \oplus B_0 \oplus B_1'')^{cl(K_2 \& K), dl(K_1)\setminus dl(K_2)} \tag{31}$$

and by rule Sub:

$$A \vdash K_2 \mathbin{\&} K \triangleright P \mathbin{\&} Q \text{ or } L' :: \zeta(\rho)(B_2' \oplus B_0 \oplus B_1'')^{W', V'} \tag{32}$$

where $W' = W \cup cl(K_2 \mathbin{\&} K)$ and $V' = V \cup (dl(K_1) \setminus dl(K_2))$.

Let L be $(dl(S) \setminus (\overline{dl(K_2)} \cup \overline{dl(K)} \cup \overline{dl(K_1)})) \cup ((\overline{B_1} \setminus \overline{B}) \setminus dl(K_2))$, or equivalently, $(dl(S) \setminus (\overline{B_2'} \cup \overline{B_0} \cup \overline{B_1'})) \cup ((\overline{B_1} \setminus \overline{B}) \setminus \overline{B_2'})$ (33). Observe that L is choosen so as to satisfy the condition $L \subseteq (dl(S) \setminus dl(C')) \cup (\overline{B_1} \setminus \overline{B})$. To conclude, we verify that other constraints of the lemma are satisfied for the judgment (32). That is,

1. $B_2' \oplus B_0 \oplus B_1'' = \underline{B \oplus (B_2 \setminus L)}$. By (6), (13), (17), (19) it is enough to check the set equality: $\overline{B_2'} \cup \overline{B_0} \cup \overline{B_1''} = \overline{B} \cup (\overline{B_2} \setminus L)$. Since both sides of (33) are restrictions outside of the set $\overline{B_2'}$, we have $L \cap \overline{B_2'} \subset \overline{B}$ (34). Therefore,

$$\begin{array}{lll} \overline{B_2'} \cup \overline{B_0} \cup \overline{B_1''} & & \\ \quad = & \overline{B_2'} \cup \overline{B_0} \cup (\overline{B_1'} \setminus \overline{dl(K_2)}) & \text{by definition of } B_1' \\ \quad = & \overline{B_2'} \cup \overline{B_0} \cup (\overline{B_1'} \setminus \overline{B_2'})) & \text{by (16)} \\ \quad = & \overline{B_2'} \cup \overline{B_0} \cup \overline{B_1'} & \\ \quad = & \overline{B_2'} \cup \overline{B} & \text{by definition of } B \\ \quad = & \overline{B} \cup (\overline{B_2} \setminus L) & \text{by (34)} \end{array}$$

2. $W' \subseteq W \cup (W_2 \cap (\overline{B \oplus (B_2 \setminus L)}))$. Because $W' = W \cup cl(K_2 \mathbin{\&} K)$ and $cl(K_2 \mathbin{\&} K) \subseteq W_2$ by (18), and $cl(K_2 \mathbin{\&} K) \subseteq dl(K_2 \mathbin{\&} K) = \overline{B_2'} \cup \overline{B_0} \subseteq \overline{B \oplus (B_2 \setminus L)}$ by the equality above.

3. $V' \subseteq V \cup (V_2 \cap (\overline{B_1} \setminus L))$. By definition, $V' = V \cup (dl(K_1) \setminus dl(K_2))$ and $dl(K_1) \setminus dl(K_2) \subseteq V_2$ by (20). It remains to prove that $dl(K_1) \setminus dl(K_2) \subseteq \overline{B_1} \setminus L$. Since $dl(K_1) \setminus dl(K_2) \subseteq \overline{B_1}$, it suffices to show that $(dl(K_1) \setminus dl(K_2)) \cap L = \emptyset$. Obviously, $(dl(K_1) \setminus dl(K_2)) \cap (dl(S) \setminus dl(K_2 \mathbin{\&} K \mathbin{\&} K_1)) = \emptyset$, whilst $(dl(K_1) \setminus dl(K_2)) \cap ((\overline{B_1} \setminus \overline{B}) \setminus dl(K_2)) = \emptyset$, since $dl(K_1) = \overline{B_1'} \subseteq \overline{B}$.

Case Filter-End**.** Let us assume

$$\begin{array}{cr} M \triangleright P \text{ with } 0 \longmapsto M \triangleright P & (1) \\ A \vdash M \triangleright P :: \zeta(\rho) B^{W,V} & (2) \\ A \vdash 0 :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2} & (3) \\ dl(0) \cap \overline{B_1} = \emptyset & (4) \\ B \subseteq B_1 & (5) \end{array}$$

Since in (3) $\overline{B_2}$ must be the empty set, we conclude from (2) by choosing $L = \overline{B_1} \setminus \overline{B}$, $W' = W$, and $V' = V$.

Case Filter-Abstract. Let us assume

$$\begin{aligned}
&L' \ \mathsf{with}\ S \longmapsto L' \\
&A \vdash L' :: \zeta(\rho)B^{W,V} && (1)\\
&A \vdash S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2} \\
&dl(S) \cap \overline{B_1} = \emptyset && (2)\\
&B \subseteq B_1 && (3)\\
&B_2 \upharpoonright \overline{B_1} \subseteq B_1 && (4)
\end{aligned}$$

A derivation of (1) must contain an instance of rule Abstract, hence $A \vdash L' :: \zeta(\rho)B^{\emptyset,V}$ and $\overline{B} = V = L'$ (5). Let L be $(dl(S) \setminus L') \cup (\overline{B_1} \setminus \overline{B})$. We show that (1) satisfies the lemma:

1. $B \oplus (B_2 \setminus L) = B$. Since by (4) B_2 is compatible with B_1 and with B by (3), it suffices to show that $\overline{B_2} \setminus L \subseteq \overline{B}$. By (5), it follows that L is equal to $(dl(S) \cup \overline{B_1}) \setminus \overline{B}$ (6). Hence:

$$\begin{aligned}
\overline{B_2} \setminus L &= ((\overline{B_2} \upharpoonright \overline{B_1}) \cup (\overline{B_2} \setminus \overline{B_1})) \setminus L \\
&\subseteq (\overline{B_1} \cup dl(S)) \setminus L && \text{by (4) and Lemma 8}\\
&= (\overline{B_1} \cup dl(S)) \setminus ((\overline{B_1} \cup dl(S)) \setminus \overline{B}) \\
&= (\overline{B_1} \cup dl(S)) \cap \overline{B}
\end{aligned}$$

2. $W \subseteq W \cup (W_2 \cap (\overline{B \oplus B_2} \setminus L))$. Obvious.

3. $V \subseteq V \cup (V_2 \cap (\overline{B_1} \setminus L))$. Obvious.

Inductive cases.

Case Filter-Next. Let us assume

$$\begin{aligned}
&M \triangleright P \ \mathsf{with}\ K_1 \Rightarrow K_2 \triangleright Q \mid S \longmapsto C' && (1)\\
&dl(K_1) \not\subseteq dl(M) && (2)\\
&A \vdash M \triangleright P :: \zeta(\rho)B^{W,V} && (3)\\
&A \vdash K_1 \Rightarrow K_2 \triangleright Q \mid S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2} && (4)\\
&((dl(K_2) \setminus dl(K_1)) \cup dl(S)) \cap \overline{B_1} = \emptyset && (5)\\
&B \subseteq B_1 && (6)\\
&B_2 \upharpoonright \overline{B_1} \subseteq B_1 && (7)
\end{aligned}$$

The selection clauses S are of the form $|_{i\in I}\ K_i' \Rightarrow K_i'' \triangleright Q_i$. A derivation of (4) must contain an instance of Modifier, with premises:

$$\begin{aligned}
&A \vdash K_1 \Rightarrow K_2 \triangleright Q :: B_1^{W_1} \Rightarrow B_2'^{W_2',V_2'} && (8)\\
&(A \vdash K_i' \Rightarrow K_i'' \triangleright Q_i :: B_1^{W_1} \Rightarrow B_i''^{W_i'',V_i''})^{i\in I}
\end{aligned}$$

where

$$\begin{aligned}
B_2'' &= \oplus_{i\in I} B_i'' \\
B_2 &= B_2' \oplus B_2'' && (9)\\
W_2'' &= \textstyle\bigcup_{i\in I} W_i'' \\
W_2 &= W_2' \cup W_2'' && (10)\\
V_2'' &= \textstyle\bigcup_{i\in I} V_i'' \\
V_2 &= V_2' \cup V_2'' && (11)
\end{aligned}$$

Hence, by rule MODIFIER, we derive

$$A \vdash S :: B_1^{W_1} \Rightarrow B_2''^{W_2'',V_2''} \tag{12}$$

A derivation of (1) must end with an instance of rule FILTER-NEXT, hence $M \triangleright P$ **with** $S \longmapsto C'$ (13). By induction hypothesis applied to (13), (3), (5), (12), (6), and (7) there must exist some L', W', and V' such that

$$A \vdash C' :: \zeta(\rho)(B \oplus (B_2'' \setminus L'))^{W',V'} \tag{14}$$
$$L' \subseteq (dl(S) \setminus dl(C')) \cup (\overline{B_1} \setminus \overline{B}) \tag{15}$$
$$W' \subseteq W \cup (W_2'' \cap (\overline{B \oplus (B_2'' \setminus L')})) \tag{16}$$
$$V' \subseteq V \cup (V_2'' \cap (\overline{B_1} \setminus L')) \tag{17}$$

Let us prove that $A \vdash C' :: \zeta(\rho)(B \oplus (B_2 \setminus L))^{W',V'}$ (18), for $L = L' \cup \overline{B_2'} \setminus (\overline{B_1} \cup \overline{B_2''})$ and check that L, W', V' satisfy the conditions of the lemma. We first prove that $L \subseteq (((dl(K_2) \setminus dl(K_1)) \cup dl(S)) \setminus dl(C')) \cup (\overline{B_1} \setminus \overline{B})$ (19). By Lemma 8 applied to (8), we have $\overline{B_2'} \setminus \overline{B_1} \subseteq dl(K_2) \setminus dl(K_1)$ (20). Notice that $dl(C') = \overline{B} \cup \overline{B_2'' \setminus L'}$ by Lemma 7 and (14), hence $dl(C') \subseteq \overline{B} \cup \overline{B_2''}$ (21). Thus, we have:

$$\begin{aligned}
L &= L' \cup \overline{B_2'} \setminus (\overline{B_1} \cup \overline{B_2''}) && \\
&= L' \cup (\overline{B_2'} \setminus \overline{B_1}) \setminus \overline{B_2''} && \\
&\subseteq L' \cup (dl(K_2) \setminus dl(K_1)) \setminus \overline{B_2''} && \text{by (20)} \\
&= L' \cup (dl(K_2) \setminus dl(K_1)) \setminus (\overline{B} \cup \overline{B_2''}) && \text{by (5)} \\
&\subseteq L' \cup (dl(K_2) \setminus dl(K_1)) \setminus dl(C') && \text{by (21)} \\
&= (dl(S) \setminus dl(C')) \cup (\overline{B_1} \setminus \overline{B}) \cup (dl(K_2) \setminus dl(K_1)) \setminus dl(C') && \text{by (15)} \\
&= ((dl(K_2) \setminus dl(K_1)) \cup dl(S)) \setminus dl(C') \cup \overline{B_1} \setminus \overline{B} &&
\end{aligned}$$

To conclude, we check the following properties:

1. $B \oplus (B_2 \setminus L) = B \oplus (B_2'' \setminus L')$. Since by (7) B_2 and B_1 agree, and so do B_2'' and B by (6) and (9), it suffices to check the equality of their domains. By

$$\begin{aligned}
\overline{B} \cup (\overline{B_2} \setminus L) &= \overline{B} \cup (\overline{B_2} \setminus (L' \cup \overline{B_2'} \setminus (\overline{B_1} \cup \overline{B_2''}))) \\
&= \overline{B} \cup (\overline{B_2} \setminus (\overline{B_2'} \setminus (\overline{B_1} \cup \overline{B_2''}))) \setminus L' \\
&= \overline{B} \cup ((\overline{B_2' \oplus B_2''}) \setminus (\overline{B_2'} \setminus (\overline{B_1} \cup \overline{B_2''}))) \setminus L' \\
&= \overline{B} \cup ((\overline{B_2'} \setminus (\overline{B_2'} \setminus (\overline{B_1} \cup \overline{B_2''}))) \cup \\
&\qquad (\overline{B_2'} \setminus (\overline{B_2'} \setminus (\overline{B_1} \cup \overline{B_2''})))) \setminus L' \\
&= \overline{B} \cup ((\overline{B_1} \cup \overline{B_2''}) \upharpoonright \overline{B_2'} \cup B_2'') \setminus L' \\
&= \overline{B} \cup \underbrace{(\overline{B_1} \upharpoonright \overline{B_2'}) \setminus L'}_{\subseteq B} \cup \underbrace{(\overline{B_2''} \upharpoonright \overline{B_2'}) \setminus L'}_{\subseteq B_2'' \setminus L'} \cup (\overline{B_2''} \setminus L') \\
&= \overline{B} \cup (\overline{B_2''} \setminus L')
\end{aligned}$$

2. $W' \subseteq W \cup (W_2 \cap \overline{B \oplus (B_2 \setminus L)})$. This follows from (16), $W_2'' \subseteq W_2$ (by (10)), and $B \oplus (B_2 \setminus L) = B \oplus (B_2'' \setminus L')$.

3. $V' \subseteq V \cup (V_2 \cap (\overline{B_1} \setminus L))$. This follows from (17), $V_2 \subseteq V_2'$ (by (11)) and $L' \subseteq L$ (by definition of L').

Case Filter-Or. Let us assume that:

$$
\begin{array}{cr}
C_1 \text{ or } C_2 \text{ with } S \longmapsto C' & (1)\\
A \vdash C_1 \text{ or } C_2 :: \zeta(\rho)B^{W,V} & (2)\\
A \vdash S :: B_1 \Rightarrow B_2^{W_2,V_2} & (3)\\
dl(S) \cap \overline{B_1} = \emptyset & (4)\\
B \subseteq B_1 & (5)\\
B_2 \upharpoonright \overline{B_1} \subseteq B_1 & (6)
\end{array}
$$

A derivation of (2) must end with an instance of rule Disjunction, followed by a sequence of rules Sub. Hence B is of the form $B_1' \oplus B_2'$ (7) and:

$$
\begin{array}{cr}
A \vdash C_1 :: \zeta(\rho){B_1'}^{W_1',V_1'} & (8)\\
A \vdash C_2 :: \zeta(\rho){B_2'}^{W_2',V_2'} & (9)\\
W_1' \cup W_2' \subseteq W & (10)\\
(V_1' \setminus (\overline{B_2'} \setminus V_2')) \cup (V_2' \setminus (\overline{B_1'} \setminus V_1')) \subseteq V & (11)
\end{array}
$$

The condition (4) implies that $dl(S) \cap \overline{B_i} = \emptyset$ for $i \in \{1,2\}$ (12). The reduction (1) implies that C' is of the form C_1' or C_2' such that C_i with $S \longmapsto C_i'$ for $i \in \{1,2\}$ (13). By induction hypothesis applied to (13), (8) and (9), (3), (12), (5), and (6), it follows that there exist some L_i, W_i'', and V_i'' such that

$$
\begin{array}{cr}
A \vdash C_i' :: \zeta(\rho)(B_i' \oplus (B_2 \setminus L_i))^{W_i'',V_i''} & (14)\\
L_i \subseteq (dl(S) \setminus dl(C_i')) \cup (\overline{B_1} \setminus \overline{B_i'}) & (15)\\
W_i'' \subseteq W_i' \cup (W_2 \cap \overline{B_i' \oplus (B_2 \setminus L_i)}) & (16)\\
V_i'' \subseteq V_i' \cup (V_2 \cap (\overline{B_1} \setminus L_i)) & (17)
\end{array}
$$

for $i \in \{1,2\}$. By rule Disjunction applied to the two cases of (14) and since B_1', B_2' and B_2'' are compatible by (6), (5) and by the definition of B_1' and B_2', we have:

$$
A \vdash C_1' \text{ or } C_2' :: \zeta(\rho)(B_1' \oplus B_2' \oplus (B_2 \setminus L_1) \oplus (B_2 \setminus L_2))^{W',V'} \qquad (18)
$$

where

$$
\begin{aligned}
W' &= W_1'' \cup W_2''\\
V' &= V_1'' \setminus (\overline{B_2' \oplus (B_2 \setminus L_2)} \setminus V_2'') \cup V_2'' \setminus (\overline{B_1' \oplus (B_2 \setminus L_1)} \setminus V_1'')
\end{aligned}
$$

Let L be $L_1 \cap L_2$. Then

$$
\begin{array}{lr}
L = L_1 \cap L_2 & \\
\quad \subseteq ((dl(S) \setminus dl(C_1)) \cup (\overline{B_1} \setminus \overline{B_1'})) \cap ((dl(S) \setminus dl(C_2)) \cup (\overline{B_1} \setminus \overline{B_2'})) & \text{by (15)}\\
\quad \subseteq (dl(S) \setminus dl(C_1)) \cap (dl(S) \setminus dl(C_2)) \cup (\overline{B_1} \setminus \overline{B_1'}) \cap (\overline{B_1} \setminus \overline{B_2'}) & \text{by distributivity}\\
\quad = (dl(S) \setminus (dl(C_1) \cup dl(C_2))) \ \cup \ (\overline{B_1} \setminus (\overline{B_1'} \cup \overline{B_2'})) & \\
\quad = (dl(S) \setminus dl(C)) \cup (\overline{B_1} \setminus \overline{B}) & (19)
\end{array}
$$

To conclude, we prove that (18) satisfies the constraints of the lemma. Indeed, we have:

1. $B \oplus (B_2 \setminus L) = B_1' \oplus B_2' \oplus (B_2 \setminus L_1) \oplus (B_2 \setminus L_2)$. Since $B = B_1' \oplus B_2'$ and $B_2 \setminus L = B_2 \setminus L_1 \oplus B_2 \setminus L_2$.

2. $W' \subseteq W \cup (W_2 \cap \overline{B \oplus (B_2 \setminus L)})$. Since $W' = W_1'' \cup W_2''$, it suffices to show that $W_i'' \subseteq W \cup (W_2 \cap \overline{B \oplus (B_2 \setminus L)})$, for $i \in \{1, 2\}$. This follows by (16), (10) and because $B_i' \oplus (B_2 \setminus L_i) \subseteq B \oplus (B_2 \setminus L)$ (by previous item).

3. $V' \subseteq V \cup (V_2 \cap (\overline{B_1} \setminus L))$. It suffices to show that both
$$V_1'' \setminus (\overline{B_2' \oplus (B_2 \setminus L_2)} \setminus V_2'') \subseteq V \cup (V_2 \cap (\overline{B_1} \setminus L))$$
and
$$V_2'' \setminus (\overline{B_1' \oplus (B_2 \setminus L_1)} \setminus V_1'') \subseteq V \cup (V_2 \cap (\overline{B_1} \setminus L))$$
Each of these two containments follows by (17), which establishes a stronger relation between a superset of the left hand side and two subsets of the two right hand sides.

$\square$

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

We show that class reduction $\stackrel{x}{\longmapsto}$ and process reduction $\longmapsto$ preserve typing, simultaneously.

That is, we prove that

1. if $A + x : [\rho], x.(B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V}$ and $C \stackrel{x}{\longmapsto} C'$ then $A + x : [\rho], x.(B \upharpoonright \mathcal{F}) \vdash C' :: \zeta(\rho)B^{W,V}$;
2. if $A \vdash P$ and $P \longmapsto P'$ then $A \vdash P'$.

<u>Proof</u> We reason by induction on the depth of the proofs of $C \stackrel{x}{\longmapsto} C'$ and $P \longmapsto P'$. We write A_x for $A + x : [\rho], x.(B \upharpoonright \mathcal{F})$.

Basic cases for class reduction.

Case Self. Let $\mathsf{self}(z)\, C \stackrel{x}{\longmapsto} C\{x/z\}$ and let $A_x \vdash \mathsf{self}(z)\, C :: \zeta(\rho)B^{W,V}$. A derivation of this judgment must end with an instance of Self-Binding, followed by a sequence of Sub rules. Hence,
$$A_x + z : [\rho], z.(B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W',V'}$$
with $W' \subseteq W$ and $V' \subseteq V$. Then, by Lemma 5, we have:
$$A_x \vdash C\{x/z\} :: \zeta(\rho)B^{W',V'}$$
We conclude $A_x \vdash C\{x/z\} :: \zeta(\rho)B^{W,V}$ by rule Sub.

Case Or-Pat. Let $J_1 \;\mathsf{or}\; J_2 \triangleright P \stackrel{x}{\longmapsto} J_1 \triangleright P \;\mathsf{or}\; J_2 \triangleright P$ and let $A_x \vdash J_1 \;\mathsf{or}\; J_2 \triangleright P :: \zeta(\rho)B^{W,V}$ (1). The derivation of this judgment must end with the following derivation followed by a sequence of Sub rules:

$$\text{Alternative}\ \frac{A' \vdash J_1 :: B_1 \qquad A' \vdash J_2 :: B_2}{A' \vdash J_1 \;\mathsf{or}\; J_2 :: B}$$
$$\text{Reaction}\ \frac{A_x + A' \vdash P \qquad dom(A') = fn(J_1 \;\mathsf{or}\; J_2)\ (2)}{A_x \vdash J_1 \;\mathsf{or}\; J_2 \triangleright P :: \zeta(\rho)B^{cl(J_1) \cup cl(J_2), \emptyset}}$$

where B is $B_1 \oplus B_2$ and $cl(J_1) \cup cl(J_2) \subseteq W$. Since $fn(J_1 \text{ or } J_2) = fn(J_1) = fn(J_2)$, we have:

$$\textsc{Disjunction}\ \dfrac{\textsc{Reaction}\ \dfrac{\begin{array}{c}A' \vdash J_i :: B_i \\ A_x + A' \vdash P \qquad dom(A') = fn(J_i)\end{array}}{A_x \vdash J_i \triangleright P :: \zeta(\rho)B_i^{cl(J_i),\emptyset}}\ i = 1,2}{A_x \vdash J_1 \triangleright P \text{ or } J_2 \triangleright P :: \zeta(\rho)B^{cl(J_1)\cup cl(J_2),\emptyset}}$$

The we conclude $A_x \vdash J_1 \triangleright P \text{ or } J_2 \triangleright P :: \zeta(\rho)B^{W,V}$ by rule Sub.

Case Abstract-Cut. Let $C \text{ or } L \xrightarrow{x} C \text{ or } L'$ and $A_x \vdash C \text{ or } L :: \zeta(\rho)B^{W,V}$ and $L' = L \setminus dl(C)$, with $L \neq L'$. Therefore $L' \subseteq L$ (1). The derivation of the judgment $A_x \vdash C \text{ or } L :: \zeta(\rho)B^{W,V}$ must end with the following derivation followed by a sequence of Sub rules:

$$\textsc{Disjunction}\ \dfrac{\textsc{Abstract}\ \dfrac{dom(B_2) = L}{A_x \vdash L :: \zeta(\rho)B_2^{\emptyset,L}} \quad A_x \vdash C :: \zeta(\rho)B_1^{W_1,V_1} \qquad L'' = L \setminus (dom(B_1) \setminus V_1)(2)}{A_x \vdash C \text{ or } L :: \zeta(\rho)(B_1 \oplus B_2)^{W_1,V_1 \cup L''}}$$

where $B = B_1 \oplus B_2$, $W_1 \subseteq W$ (3) and $V_1 \cup L'' \subseteq V$ (4).

We first observe that $B_1 \oplus B_2 = B_1 \oplus (B_2 \upharpoonright L')$. Therefore we can derive:

$$\textsc{Disjunction}\ \dfrac{\textsc{Abstract}\ \dfrac{dom(B_2 \upharpoonright L') = L'}{A_x \vdash L' :: \zeta(\rho)(B_2 \upharpoonright L')^{\emptyset,L'}} \quad A_x \vdash C :: \zeta(\rho)B_1^{W_1,V_1} \qquad L''' = L' \setminus (dom(B_1) \setminus V_1)(5)}{A_x \vdash C \text{ or } L' :: \zeta(\rho)(B_1 \oplus B_2)^{W_1,V_1 \cup L'''}}$$

By (2), (5) and (1), we derive $V_1 \cup L''' \subseteq V_1 \cup L''$. Hence, by (3), (4) and rule Sub, we obtain $A_x \vdash C \text{ or } L' :: \zeta(\rho)(B_1 \oplus B_2)^{W,V}$.

Case Class-Abstract. Let $C \text{ or } \emptyset \xrightarrow{x} C$ and $A_x \vdash C \text{ or } \emptyset :: \zeta(\rho)B^{W,V}$. The derivation of this judgment must end with rule Disjunction followed by a sequence of Sub rules:

$$\textsc{Disjunction}\ \dfrac{A_x \vdash C :: \zeta(\rho)B^{W',V'} \qquad A_x \vdash \emptyset :: \zeta(\rho)\emptyset^{\emptyset,\emptyset}}{A_x \vdash C \text{ or } \emptyset :: \zeta(\rho)B^{W',V'}}$$

where $W' \subseteq W$ and $V' \subseteq V$. Then, by rule Sub applied to $A_x \vdash C :: \zeta(\rho)B^{W',V'}$, we obtain $A_x \vdash C :: \zeta(\rho)B^{W,V}$.

Basic cases for processes.

Case Class-Var. Let us assume $A \vdash \text{class } c = \text{self}(z)\, C \text{ in } P$ (1) and $\text{class } c = C \text{ in } P \longmapsto P\{{}^{C}/_{x}\}$. The final part of the derivation of (1) must have the form

$$\textsc{Class}\ \dfrac{A \vdash C :: \zeta(\rho)B^{W,V}\ (3) \qquad A + c : \forall\text{Gen}\ (\rho, B, A).\zeta(\rho)B^{W,V} \vdash P\ (2)}{A \vdash \text{class } c = C \text{ in } P}$$

By Lemma 6 applied to (3) and (2), we derive $A \vdash P\{{}^{C}/_{c}\}$.

Inductive cases for classes.

Case Class-Context. Let $A_x \vdash E[C] :: \zeta(\rho)B^{W,V}$ and $E[C] \xrightarrow{x} E[C']$. By inductive hypothesis, if $A_x \vdash C :: \zeta(\rho){B'}^{W',V'}$ then $A_x \vdash C' :: \zeta(\rho){B'}^{W',V'}$, since $C \xrightarrow{x} C'$. The judgment $A_x \vdash E[C'] :: \zeta(\rho)B^V$ follows by induction on the structure of $E[\cdot]$. The details are omitted.

Case Match. Let us assume that $A \vdash \mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end} : \zeta(\rho)B^{W,V}$ (1) and $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end} \longrightarrow C'$ (2). We must prove that $A \vdash C' : \zeta(\rho)B^{W,V}$ (3)

A derivation of (1) must end with an instance of rule Refinement followed by a sequence of Sub. Hence, B is of the form $B_1 \oplus B_2$ (4) and

$$\begin{aligned} A \vdash C :: \zeta(\rho)B_1^{W_1,V_1} &\quad (5)\\ A \vdash S :: B_1^{W_1} \Rightarrow B_2^{W_2,V_2} &\quad (6)\\ dl(S) \cap dom(B_1) = \emptyset &\quad (7)\\ W_1 \cup W_2 \subseteq W &\quad (8)\\ V_1 \cup V_2 \subseteq V &\quad (9) \end{aligned}$$

The derivation of (2) must contain a rule Match with the premises:

$$\begin{aligned} C\ \mathsf{with}\ S \longrightarrow C' &\quad (10)\\ dl(S) \subseteq dl(C') &\quad (11) \end{aligned}$$

From (4) it follows that $B_2 \upharpoonright dom(B_1) \subseteq dom(B_2)$ (12). Lemma 9 applied to (10), (5), (6), (7), and (12) implies that

$$\begin{aligned} A \vdash C' :: \zeta(\rho)(B_1 \oplus (B_2 \setminus L))^{W',V'} &\quad (13)\\ L \subseteq (dl(S) \setminus dl(C')) \cup (dom(B_1) \setminus dom(B_1)) &\quad (14)\\ W' \subseteq W_1 \cup (W_2 \cap dom(B_1 \oplus (B_2 \setminus L))) &\quad (15)\\ V' \subseteq V_1 \cup (V_2 \cap (dom(B_1) \setminus L)) &\quad (16) \end{aligned}$$

The property (11) combined with (14) imply that L is empty. Therefore $W' \subseteq W_1 \cup W_2$ and $V' \subseteq V_1 \cup V_2$. Hence (3) follows by (13), (8), (9) and rule Sub.

Inductive cases for processes.

Case Class-Red. Let $A \vdash \mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ Q$ (1) and $\mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ Q \longmapsto \mathsf{obj}\ x = C'\ \mathsf{init}\ P\ \mathsf{in}\ P'$, under the assumption that $C \xrightarrow{x} C'$ (2).

A derivation of (1) has the shape

$$\textsc{Self-Binding}\ \frac{A + x : [\rho], x : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,\emptyset}\ (3)}{A \vdash \mathsf{self}(x)\ C :: \zeta(\rho)B^{W,\emptyset}}$$

$$\textsc{Object}\ \frac{\begin{array}{c}\rho = B \upharpoonright \mathcal{M} \qquad X = \mathsf{Gen}\ (\rho, B, A) \setminus ctv(B \upharpoonright W)\\ A + x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F}) \vdash P \qquad A + x : \forall X.[\rho] \vdash Q\end{array}}{A \vdash \mathsf{obj}\ x = C\ \mathsf{init}\ P\ \mathsf{in}\ Q}$$

By induction hypothesis applied to (2) and (3), we obtain the judgment $A + x : [\rho], x : (B \upharpoonright \mathcal{F}) \vdash C' :: \zeta(\rho)B^{W,\emptyset}$, which we can substitute in the previous derivation, thus concluding $A \vdash \mathsf{obj}\ x = C'\ \mathsf{init}\ P\ \mathsf{in}\ Q$. □

B.4 Safety (Theorem 2)

<u>Proof</u> Let us assume $\vdash (\mathcal{D} \Vdash \mathcal{P})$. By Chemical-Solution and Definition, $\vdash (\mathcal{D} \Vdash \mathcal{P})$ holds provided

$$(A^{\psi} \vdash D :: A_x)^{\psi x \,\#\, D \in \mathcal{D}} \ (1) \qquad (A^{\psi} \vdash P)^{\psi \,\#\, P \in \mathcal{P}} \ (2) \qquad A = \cup_{\psi x \,\#\, D \in \mathcal{D}} A_x \ (3)$$

We check that no case listed in Definition 1 (Section 6.1) can occur.

No free variables. By definition of type judgments, since (1) hold, every free object name in D, with $\psi x \,\#\, D \in \mathcal{D}$, should appear as a leaf of the proof tree of $A^{\psi} \vdash D :: A_x$. This leaf must be of the form $A^{\psi} + A' \vdash x : \tau$. This implies that x belongs to the domain of A^{ψ} because x is free in D. Similarly, every class variable in D should belong to the domain of A^{ψ}, which actually only contains object names. The proof is similar for free names in P using (2).

No runtime failure. Let $\psi \,\#\, x.\ell(\widetilde{u}) \in \mathcal{P}$ and $\psi' x \,\#\, D \in \mathcal{D}$ (4).

1. (*no privacy failure*) Let ℓ be a private label f. We prove that $\psi' x$ is a prefix of ψ. A derivation of $A^{\psi} \vdash x.\ell({u_i}^{i \in I})$ must be:

$$\dfrac{\dfrac{\cdots}{A^{\psi} \vdash x.f : ({\tau_i}^{i \in I})}((5)) \qquad (A^{\psi} \vdash u_i : \tau_i)^{i \in I}}{A^{\psi} \vdash x.\ell({u_i}^{i \in I})} \ \textsc{Send}$$

   where (5) is an instance of Private-Message. The premise of (5) requires that $x : \forall X.(\ell : ({\tau_i}^{i \in I}); B')$ be in A^{ψ}. Therefore, by definition of A^{ψ}, variable x must appear in ψ. Furthermore, by well-formedness of chemical solutions, a name can have a unique prefix. Since ψ' is already a prefix of x, then ψ must be of the form $\psi' x \psi''$.

2. (*no undeclared label*) We show that $\ell \in dl(D)$. Given (4), the judgment $A^{\psi'} \vdash D :: A_x$, where $A_x = x : \forall X.\rho, x : \forall X.(B \upharpoonright \mathcal{F})$, follows by rule Definition applied to (1) with the premises below:

$$A^{\psi'} \vdash \mathsf{self}(x)\, D :: \zeta(\rho) B^{W,\emptyset} \ (6) \qquad \rho = B \upharpoonright \mathcal{M} \ (7)$$

$$X = \mathsf{Gen}\ (\rho, B, A^{\psi'}) \setminus ctv(B \upharpoonright W)$$

   Since $A^{\psi} \vdash x.\ell(\widetilde{u})$ by (2), either ρ is of the form $[\ell : \widetilde{\tau}; \rho']$ or B is of the form $(\ell : \widetilde{\tau}; B')$ depending on whether ℓ is public or private. In each case, using (7), ℓ is in $dom(B)$. The conclusion follows by Lemma 7.

3. (*no arity mismatch*) Let D be of the form $[M \triangleright P]$ where M is itself of the form $\ell(\widetilde{y})\ \&\ J$. We show that $\widetilde{y}$ and $\widetilde{u}$ have the same arities.

   For that purpose, it suffices to show that the type of $\widetilde{u}$ and the type of $\widetilde{y}$

in A are instances of a same tuple type. A leaf of (6) must be

$$\textsc{Reaction}\ \dfrac{\dfrac{\left(\dfrac{\textsc{Message-Pattern}\quad (y_i : \tau'_i \in A')^{i\in I}}{A' \vdash \ell(\widetilde{y}) :: \ell : (\tau'_i{}^{i\in I})}\right)^{\ell(\widetilde{y})\in M}}{A' \vdash M :: B} \quad \begin{array}{c} dom(A') = fn(M) \\ \vdots \quad A^{\psi'} + x : [\rho], x : (B \upharpoonright \mathcal{F}) + A' \vdash P \end{array}}{A^{\psi'} + x : [\rho], x : (B \upharpoonright \mathcal{F}) \vdash M \triangleright P :: \zeta(\rho)B^{cl(M),\emptyset}}$$

Therefore, the type of $\widetilde{y}$ in A' is $B(\ell)$. By rules Chemical-Solution and Definition, A contains a generalization of $x : [\rho], x : (B \upharpoonright \mathcal{F})$. Thus, the type of $x.\ell$ in A^{ψ} is a generalization of $B(\ell)$. The proof tree illustrated in item 1 is required to prove $A^{\psi} \vdash x.\ell(u_i{}^{i\in I})$. Then, as a consequence of rule (5), the type of $\widetilde{u}$ is an instance of the type of $x.\ell$ in A^{ψ}, *i.e.* of the generalization of the type of $\widetilde{y}$ in A'.

No class rewriting failure. Let $\psi \# P \in \mathcal{P}$, $P = \mathsf{obj}\ x = C\ \mathsf{init}\ Q\ \mathsf{in}\ Q'$, rule Class-Red does not apply to P, i.e. there is no C' such that $C \xmapsto{x} C'$, and P is not a refinement error. We show that P is not a failure; namely, for every evaluation context E,

1. $C \neq E[c]$, and c is free. By (1), $dom(A)$ only contains object names. Therefore, by (2), P cannot contain free class names.

2. Let $E[L] = C'\ \mathsf{or}\ L$ (the case $E[L] = L\ \mathsf{or}\ C'$ is similar). We demonstrate that, if $A' \vdash C'\ \mathsf{or}\ L :: \zeta(\rho)B^{W,V}$ then $L \subseteq V$. By rule Abstract, $A' \vdash L :: \zeta(\rho)B_1^{\emptyset,L}$ (8). Let $A' \vdash C' :: \zeta(\rho)B_2^{W_2,V_2}$ (9) and $V'_1 = L \setminus (dom(B_2) \setminus V_2)$ (10) and $V'_2 = V_2 \setminus (dom(B_1) \setminus L)$ (11). Since there does not exists C'' such that $C \xmapsto{x} C''$, the rule Abstract-Cut cannot be applied. This means that $L = L \setminus dl(C') = L \setminus dom(B_2)$, which implies $V'_1 = L$. By (8), (9), (10), (11) and rule Disjunction we obtain $A' \vdash C\ \mathsf{or}\ L :: \zeta(\rho)(B_1 \oplus B_2)^{W_1 \cup W_2, L \cup V'_2}$ (12).

   On the other hand, by (2), a derivation of $A^{\psi} \vdash P$ must contain ($P = \mathsf{obj}\ x = C\ \mathsf{init}\ Q\ \mathsf{in}\ Q'$)

$$\textsc{Object}\ \dfrac{\textsc{Self-Binding}\ \dfrac{A^{\psi} + x : [\rho],\ x.(B \upharpoonright \mathcal{F}) \vdash (C'\ \mathsf{or}\ L) :: \zeta(\rho)B^{W,\emptyset}(13)}{A^{\psi} \vdash \mathsf{self}(x)\,(C'\ \mathsf{or}\ L) :: \zeta(\rho)B^{W,\emptyset}} \quad \begin{array}{c} \rho = B \upharpoonright \mathcal{M} \qquad X = \mathsf{Gen}\ (\rho, B, A) ctv(B \upharpoonright W) \\ A^{\psi} + x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F}) \vdash Q \qquad A^{\psi} + x : \forall X.[\rho] \vdash Q' \end{array}}{A \vdash \mathsf{obj}\ x = (C'\ \mathsf{or}\ L)\ \mathsf{init}\ Q\ \mathsf{in}\ Q'}$$

   To conclude, observe that (12) and (13) do not unify because virtual labels in (12) are not empty.

$\square$