Inheritance in the Join Calculus

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

December 27, 2000

Abstract

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

---

[*] 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

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

The design and the implementation of concurrent object-oriented languages, *e.g.* [2, 23, 1, 4], has recently prompted the investigation of the theoretical foundations of concurrent objects. Several works provide encodings of objects in process calculi [22, 20, 12, 6] or, alternatively, supplement objects with concurrent primitives [16, 3, 11]. Those works promote a unified framework for reasoning about objects and processes, but they do not address the composition of object definitions or its typechecking.

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

The join calculus is a simple name-passing calculus, related to the pi calculus but with a functional flavor. It is the core of a distributed programming language, currently implemented as an extension of ML [9, 13]. In the join calculus, communication channels are statically scoped: when channels are created, their definition provides a set of *reaction rules* that specify, once for all, how messages sent on these names will be synchronized and processed. Although the join calculus does not have a primitive notion of object, definitions encapsulate the details of synchronization, much as concurrent objects.

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

For every object of our language, message synchronization is defined and compiled as a whole. This allows an efficient compilation of message delivery into automata [14] and simplifies reasoning on objects. However, the static definition of behavior can be overly restrictive for the programmer. This suggests some compile-time mechanism for assembling partial definitions. To this end, we promote partial definitions into classes. Classes can be combined and transformed to form new classes. They can also be closed to create objects.

The class language is layered on top of the core objective calculus, with a semantics that reduces classes into plain object definitions. We thus retain strong static properties for all objects at run-time. Some operators are imported from sequential languages and adapted to a concurrent setting. For instance, multiple inheritance is expressed as a disjunction of join definitions, but some disjunctions have no counterpart in a sequential language. In addition, we propose a new operator, called *selective refinement*. Selective refinement applies to a parent class, and rewrites the parent reaction rules according to their synchronization patterns. Selective refinement treats synchronization concretely, but it handles the parent processes abstractly. Our approach is compatible with the JoCaml implementation of the join calculus [13], which already singles out synchronization patterns using concrete compile-time representation, and, on the contrary, compiles behaviors into functional closures.

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

We finally equip the language with a polymorphic type system in the style of [10]. (We introduce the type system at the end of the paper only, to separate design issues and typing issues.)

In addition to basic safety properties, the type system enforces privacy. Indeed, the untyped objective join calculus lacks expressiveness as regards encapsulation[1]. In order to restrict access to the internal state of objects, labels are partitioned into two sets, *public* and *private*. Then, the typing system guarantees that private labels are accessed only from the body of a class used to create the object. The correctness of the typing system is expressed for an operational semantics supplemented with privacy information.

The paper is organized as follows. In section 2, we present the objective join calculus and develop a few examples. In section 3, we supplement the language with classes and give a reduction semantics for the class language, which can be seen as a compilation down to the core language. In section 4, we provide more involved examples of inheritance and concurrency. In section 5, we provide a static semantics for our calculus and we state its correctness. In section 6, we discuss related works and possible extensions. The main proofs are gathered in an appendix.

2 The objective join calculus

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

2.1 Getting started

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

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

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

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

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

---

[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.

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 contents a, and passing an empty message to s. (Object r may be the previously-defined *continuation*; object s is another continuation taking no argument on *reply*.) As regards the syntax, concurrent execution and message synchronization are expressed in a symmetric manner using the same infix operator &. Also, the calculus is polyadic, *i.e.*, messages carry tuples of values.

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

```
obj abuffer = self(z)
   put(a,r) ▷ r.reply() & z.Some(a)
or get(r) & Some(a) ▷ r.reply (a)
```

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

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

```
obj buffer = self(z)
   put(a,r) & Empty() ▷ r.reply() & z.Some(a)
or get(r)  & Some(a) ▷ r.reply(a) & z.Empty()
init  buffer.Empty()
```

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 cannot send messages on these labels directly. In section 5, we describe a refined semantics and a type system that distinguishes private labels such as *Empty* and *Some* from public labels, and we restrict access to private labels. In the examples, private labels conventionally bear an initial capital letter.

Once private labels are hidden, each variant of our buffer provides the same interface to the outside world (two methods labeled *get* and *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 for labels $\ell \in \mathcal{L}$. Tuples of names are written ${x_i}^{i \in I}$ or simply $\widetilde{x}$. The grammar of the *objective join calculus* (without classes) is given in Figure 1; it has syntactic categories for processes P, definitions D, and patterns M. We

Figure 1: Syntax for the core object calculus

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

abbreviate $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ by omitting $\mathsf{self}(z)$ when z does not occur free in D and omitting $\mathsf{init}\ P$ when P is $\mathsf{0}$.

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. We require that every pattern M guarding a reaction rule be linear; that is, each label and each name appears at most once in M. This will be enforced by typing. In addition, the object definition $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$ binds two names x and z to D. The scope of x is the processes P_1 and P_2; the scope of z is every guarded process in D. Free names in processes and definitions, written $\mathit{fn}(\cdot)$, are defined according to these binders. 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 defined in terms of a *reflexive chemical abstract machine* [8]. Each rewrite rule applies to configurations of objects and processes, called *chemical solutions.* A solution $\mathcal{D} \Vdash \mathcal{P}$ consists of a set of named object definitions $\mathcal{D}$ and of a multiset of processes $\mathcal{P}$ running in parallel. We write $x.D$ for a named definition in $\mathcal{D}$, and always assume that there is at most one definition for x in $\mathcal{D}$. We use two families of rewrite rules between solutions: *structural rules* $\equiv$ represent the syntactical rearrangement of terms; *reduction rules* $\longrightarrow$ represent the basic computation steps for objects.

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

The rules Par and Nil make parallel composition of processes associative and commutative, with unit $\mathsf{0}$. The rule Obj describes the introduction of an object; its side condition enforces a lexical scoping discipline for object names.

Figure 2: Chemical semantics

Par

$$\Vdash P \mathrel{\&} Q \;\equiv\; \Vdash P\,,\; Q$$

Nil

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

Obj

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

Join

$$\Vdash x.(M \mathrel{\&} M') \;\equiv\; \Vdash x.M\,,\; x.M'$$

Red

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

Delta

$$\frac{(P\,, P') \in \delta}{\Vdash P \;\longrightarrow \Vdash P'}$$

(Preliminary α-conversion may be required to pick a fresh name x.) The rule Join gathers messages sent to the same object. The reduction rule Red states how messages can be jointly consumed and replaced by a copy of a guarded process, in which the contents of these messages are substituted for the formal parameters appearing in the pattern. The reduction rule Delta allows to parameterize the chemical semantics by a primitive reduction of processes modelled by a relation δ on processes. By default, δ can be taken as the empty relation.

As usual with chemical semantics, each rule mentions only the components that participate to the rewriting, but the rewriting applies to every chemical solution that contains them.

That is, for $\Rrightarrow$ in $\{\equiv, \longrightarrow\}$, we also have the context rule

Chemistry

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

Moreover, we have a context rule that deals with name generation

Chemistry-Obj

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

where the sets $\mathit{fn}(\mathcal{D})$ and $\mathit{fn}(\mathcal{P})$ are defined in Figure 4. The side condition $x \notin \mathit{fn}(\mathcal{D}) \cup \mathit{fn}(\mathcal{P})$ prevents name capture.

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

3 Inheritance and concurrency

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

3.1 Deriving a concurrent class

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

```
class buffer = self(z)
    get(r)  & Some(a) ▷ r.reply(a) & z.Empty()
or  put(a,r) & Empty() ▷ r.reply() & z.Some(a)
```

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

```
obj b = buffer init b.Empty()
```

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

```
class logged_buffer = self(z)
    buffer
or  log() & Some(a) ▷ out.print_int(a) & z.Some(a)
or  log() & Empty() ▷ out.print_string("Empty") & z.Empty()
```

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 that 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.

Other possible debugging information is the synchronous log of all messages that are consumed on *put*. This is done by selecting the patterns in which *put* occurs and adding a printing message to the corresponding guarded processes:

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

The match construct can be understood by analogy with pattern matching *à la* ML, applied to the reaction rules of the parent class. 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 ⇒) and with the original guarded process in parallel with the new printing message (the original guarded process and the & are left implicit in the match syntax). Every other parent rule is kept unchanged. Hence, the above definition behaves as the following definition:

```
class logged_buffer_bis =
    get(r)  & Some(a) ▷  r.reply(a) & z.Empty()
or  put(a,r) & Empty() ▷  r.reply() & z.Some(a) & out.print_int(a)
```

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

```
class $logged\_buffer\_ter = \mathsf{self}(z)$
  match $buffer$ with
    $put(a,r) \Rightarrow Parent\_put(a,r) \triangleright O$
  end
or $put(a,r) \triangleright out.print\_int(a)\ \&\ z.Parent\_put(a,r)$
```

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

The examples above illustrate that the very idea of class refinement is less abstract in a concurrent setting than in a sequential one. In the first $logged\_buffer$ example, logging the buffer state requires knowledge of how this state is encoded; otherwise, some states might be forgotten or 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 or. We use two additional sets of identifiers for class names $c \in \mathcal{C}$ and for sets of labels $L \in 2^{\mathcal{L}}$. Such sets L are used to represent abstract classes that declare the labels in L but do not define them.

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

Selection patterns K are either join patterns or the empty pattern $\mathsf{0}$. Their normal forms are of the form above, except that I can be empty. We sometimes use the notation $K_1 \mathbin{\&} K_2$ for decomposing patterns M; we then use the additional law $K \mathbin{\&} \mathsf{0} = K$.

We asume a well-formedness condition on refinement clauses, that forbids useless clauses. That is, in a refinement sequence $K \Rightarrow K' \triangleright P \| S$ the filter pattern K cannnot be a subset of (or equal to) any other filter pattern K_i of a refinement rule $K_i \Rightarrow K_i' \triangleright P_i$ of S.

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

1. All conjuncts M_i in the normal form of K are linear (as defined in section 2.2) and bind the same names. By extension, we say that K binds the names $\mathit{fn}(M_i)$ bound in every 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 $\mathsf{0}$, and the pattern K_2 binds at least the names of K_1 ($\mathit{fn}(K_1) \subseteq \mathit{fn}(K_2)$).

Figure 3: Syntax for classes

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

As described in section 2.2, binders for object names include object definitions (binding the defined object to name x and self name z) and patterns (binding the received names). In a reaction rule $J \triangleright P$, the join pattern J binds $\mathit{fn}(J)$ 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, the self renaming $\mathsf{self}(x)\, C$ binds the object name x with scope C. Class definitions $\mathsf{class}\ c = C\ \mathsf{in}\ P$ are the 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.

Labels don't have scopes. Labels declared in patterns and classes, written $\mathit{dl}(K)$ and $\mathit{dl}(C)$, are specified in Figure 4.

3.3 Class rewriting semantics

Reduction rules are described below, first for classes, then for processes.

Reductions for classes. Class reduction $\overset{x}{\longmapsto}$ carries an object name that stands for the (topmost) name of self. This reduction is defined in Figure 5.

The first two rules simplify abstract classes: rule Class-Abstract discards

Figure 4: Definitions of $dl(\cdot)$ and $fn(\cdot)$

In join patterns:

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

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

In selection patterns:

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

In classes:

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

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

In processes:

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

In solutions:

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

Figure 5: Reduction for classes

Rules for classes

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

$$\textsc{Abstract-Cut}\qquad \frac{L \setminus dl(C) \subseteq L' \subseteq L \qquad L' \neq L}{C \mathsf{\ or\ } L \xmapsto{x} C \mathsf{\ or\ } L'}$$

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

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

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

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

Rules for filters

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

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

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

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

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

Evaluation contexts for classes

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

empty abstract classes; rule Abstract-Cut discards labels that are declared elsewhere. Rule Self removes inner bindings for the name of self. Rule Or-Pat lifts or constructs from patterns to classes.

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

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

When we have C with $S \longmapsto C'$, some labels in the reactions rules of C may not appear anymore in those of C'. More specifically, rule Filter-Apply removes all the labels in $dl(K_1) \setminus dl(K_2)$ from the matched reaction rule K_1 & $K \triangleright P$. However, lost labels are kept as abstract labels. Our type system (defined in section 5) computes an approximation of abstract names. Therefore, we statically ensure that lost labels are redefined (by providing suitable reaction rules) before an instance of the class can be taken.

Conversely, some labels in the reaction rules of C' may not appear in those of C. More specifically, rule Filter-Apply introduces fresh labels $dl(K_2) \setminus dl(K_1)$ into the refined reaction rule K_2 & $K \triangleright P$ & Q. The set $dl(S)$ in Figure 5 collects the potential fresh labels of S and is statically computable. However, some clauses of S may not be used, hence some label of $dl(S)$ may not actually appear in the patterns of C'. This situation often corresponds to a programming error; its detection usually requires knowledge of the synchronization patterns of C. To retain expressiveness of selective refinement and keep types simple, we detect this situation dynamically. This is expressed by the premise $dl(S) \subseteq dl(C')$ of rule Match.

Reductions for processes. Reduction $\longmapsto$ for processes is defined in Figure 6. An irreducible process either belongs to the core objective join-calculus, or it contains an irreducible class construct, as described above.

Figure 6: Reduction for processes

Class-Red

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

Class-Var

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

Note that, we will not allow reduction of classes under guards. This could be modeled by adding a the following reduction rule for processes:

Process-Context

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

where $E[\,\cdot\,]$ is any process context.

With the rules in Figures 5 and 6, our rewriting semantics is deterministic. Moreover, an arbitrary class definition can be reduced to a definition in the core object calculus, provided that it does not contain free class names, undefined declared labels, or unresolved refinements. The first two cases are ruled out by the type system in section 5. The last case occurs only when the rule Match does not apply because of the condition $dl(S) \subseteq dl(C')$.

3.4 Implementation issues

Both disjunction and selective refinement hinder the abstract compilation of patterns, because the compiler must access the patterns of the parent class to build the rules of the derived class. However, the guarded processes attached to individual reaction rules can be compiled into closures abstracted with respect to their free names (including formal message parameters), since no class construct can affect these processes.

This approach is compatible with the JoCaml implementation, which represents join patterns as vectors of bits; the control of the synchronization of messages and the activation of guarded processes is then realized by interpreting these bit vectors at runtime [14].

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

4 Inheritance anomaly

As remarked by many authors, the classical point of view on class abstraction—method names and signatures are known, method bodies are abstract—does not mix well with concurrency. More specifically, 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 [15] 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 controls synchronization in the parent class (which is 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 started 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 [15], as we do in this section, appears to be a valuable test of our design.

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 create(n), that 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 [15] can then be written as follows:

```
class $buff$ = self ($z$)
   $put(v,r)$ & ($Empty(A, i, n)$ or $Some(A, i, n)$) ▷
      $r.reply()$ & $z.Check(A[(i+n) \bmod A.size] \leftarrow v, i, n+1)$
or $get(r)$ & ($Full(A, i, n)$ or $Some(A, i, n)$) ▷
      $r.reply(A[i])$ & $z.Check(A, (i+1) \bmod A.size, n-1)$
or $Check(A,i,n)$ ▷
      if $n = A.size$ then $z.Full(A, i, A.size)$
      else if $n = 0$ then $z.Empty(A, 0, 0)$
      else $z.Some(A, i, n)$
or $Init(size)$ ▷ $z.Empty$(create($size$),0,0)
```

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

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

Since *get2* succeeds when the buffer contains two elements or more, the buffer state needs to be refined. Furthermore, since for instance a successful *get2* may disable *get* or enable *put*, the addition of *get2* has an impact on the "acceptable states" of 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 the state when the buffer holds exactly one item; *Many* represents a state with two items or more in the buffer.

```
class $buff2$ = self($z$)
   $get2(r)$ & ($Full(A,i,n)$ or $Many(A, i, n)$) ▷
        $r.reply(A[i], A[(i+1) \bmod A.size])$
    &   $z.Check(A, (i+2) \bmod A.size, n-2)$
or match $buff$ with
   $Some(A, i, n)$ ⇒ ($One(A, i, n)$ or $Many(A, i, n)$) ▷ $O$ 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:

```
class locker = self (z)
    suspend(r) & Free() ▷ r.reply() & z.Locked()
or resume(r) & Locked() ▷ r.reply() & z.Free()
```

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:

```
class locked_buff = self (z)
    locker
or match buff with
      Init(size) ⇒ Init(size) ▷ z.Free()
    | O ⇒ Free() ▷ z.Free()
    end
```

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.

4.1 Further anomalies

Passing the test suite above does not mean that we "solve" the inheritance anomaly problem. Indeed, most limitations of expressiveness can be interpreted as inheritance anomalies. We conclude this section with a more difficult pattern, for which we provide 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 simple reference cell

```
class refcell = self(z)
        read(r) | State(v) ⇒ r.reply(v) & z.State(v)
     or write(v,r) | State(w) ⇒ r.reply() & z.State(v)
```

We want to supplement *refcell* with an *incr* method that increments the buffer content. For some reason, we also require that the incrementer's *read* and *write* operations be encapsulated in another object *server*, such as

```
obj server =
   do_incr(x,r) ▷ obj s = reply(v) ▷ x.write(v+1,r) in x.read(s)
```

In *refcell*, calls to *read* and *write* are handled in parallel; in the refined class, however, any call to *write* should be blocked during the execution of *do_incr*, unless it is issued by *server*. To enforce this partial exclusion, we introduce a *Mutable* flag, we take two copies of the parent patterns, and we specialize their definitions of *write* to handle both external calls (when the cell is mutable) and privileged calls (only from the server). In the latter refinement clause, the conflicting method *write* is renamed to *Write_int*, and a "proxy object" that forwards *write* calls to *Write_int* is passed to the server. We arrive at the following refinement:

```
class counter = self(z)
        match buffer with
          write(v,r) ⇒ write(v,r) & Mutable() ▷ z.Mutable()
     or match buffer with
          write(v,r) ⇒ Write_int(v,r) ▷ 0
     or incr(r) & Mutable() ▷
          obj s = reply() ▷ r.reply() & z.Mutable() in
          obj proxy = read(r) ▷ z.read(r)
                   or write(v,r) ▷ z.Write_int(v,r) in
          server.do_incr(proxy,s)
     or Init() ▷ z.State(0) & z.Mutable()
```

Figure 7: Syntax for types

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

5 Types and privacy

The static semantics of our calculus is inspired by those of the core join calculus for concurrency and synchronization [10] and of Objective Caml for the class-layer [18].

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

For that purpose, we partition labels $\ell \in \mathcal{L}$ into *private labels* $f \in \mathcal{F}$ and *public labels* $m \in \mathcal{M}$. A message on a private label can only be sent internally, that is, from within a reaction rule of the object to which the message is sent. (A precise definition can be found in section 5.4.) Conversely, a message on a public label can be sent from any context that has access to the object name.

5.1 Syntax

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

Object types $[\rho]$ are modeled as records of the types of public labels. Types B are used to describe private labels of objects and class types. 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 $\ell_1, \ldots\ \ell_n$.

Class types have the form $\forall X.\zeta(\rho)B^{W,V}$. The internal type B lists all public and private labels declared in the class, with their types; we write $dom(B)$ for the set of labels listed in B. 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. The set $W \subseteq dom(B)$ contains a subset of labels that are used for synchonization in B. Other labels of B can only appear in simple join patterns that do not involve any join.

Figure 8: Typing Judgments

$A \vdash P$	the process P is well-typed in environment A;
$A \vdash x : \tau$	the object x has type τ in environment A;
$A \vdash x.\ell : \widetilde{\tau}$	the label ℓ conveys messages of type $\widetilde{\tau}$ for object x in environment A;
$A \vdash K :: B^W$	the selection pattern K binds variables well-typed in A, joins labels in B, and synchronizing labels in W ($W \subseteq dom(B)$).
$A \vdash C :: \zeta(\rho)B^{W,V}$	the class C is well-typed in environment A, declares the labels of B, has virtual labels V, and synchonizing labels in W ($V, W \subseteq dom(B)$).
$A \vdash S :: (\vert^{i \in I}\ B_i \Rightarrow B_i'^{W_i})^V$	the refinement clauses S are well-typed in environment A, refine patterns with labels in B_i into patterns with labels in B'_i, has virtual labels V, and synchronizing labels in W_i ($V \subseteq \bigcup_{i \in I} B_i$, $W_i \subseteq dom(B'_i)$, for every i).

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 mostly from recursive calls, and also from passing self as a parameter in messages. Some labels may appear in $[\rho]$ although they are not declared in B; classes with such labels cannot be instantiated.

5.2 Typing rules

The typing judgments are described in Figure 8.

A type environment A binds object names x to type schemes σ or to internal type schemes $\forall X.B$ and class names c to class type schemes; we write $dom(A)$ for the set of names bound in A. Each object x in A usually has two bindings $x : \forall X.[\rho]$ and $x : \forall X.B$, where labels in the domain of B are private. The binder $x : \forall X.B$ defines the type of private labels of x.

The typing rules rely on the following auxiliary definitions:

- $ftv(\_)$ is the set of free type variables occurring in a type, a tuple type, a type scheme, or a typing context.
- $B \upharpoonright \mathcal{M}$ and $B \upharpoonright \mathcal{F}$ are the restrictions of B on public labels and private labels, respectively;
- $B_1 \oplus B_2$ is the union of B_1 and B_2, with the statement that B_1 and B_2 coincide on their common labels. We write $B_1 \subseteq B_2$ when there is B'_1 such that $B_1 \oplus B'_1 = B_2$.
- $A + A'$ is $(A \setminus dom(A')) \cup A'$, where $A \setminus X$ removes from A all the bindings of names in X.

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

Figure 9: Typing rules for names

$$\textsc{Object-Var}\quad \frac{x : \forall X.\tau \in A}{A \vdash x : \tau\{\gamma_\alpha/\alpha^{\alpha\in X}\}} \qquad \textsc{Message}\quad \frac{A \vdash x : [m : \widetilde{\tau}; \rho]}{A \vdash x.m : \widetilde{\tau}} \qquad \textsc{Private-Message}\quad \frac{x : \forall X.(f : \widetilde{\tau}; B) \in A}{A \vdash x.f : \widetilde{\tau}\{\gamma_\alpha/\alpha^{\alpha\in X}\}}$$

Typing rules for processes. 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 defined in B to public ones. The constraint $\rho = B \restriction \mathcal{M}$ checks, for each public label m of B, that the type given to m in B and in ρ are the same; this latter type is the one that has been used for recursive calls.

The process Q is typed in an environment extended with the object x bound to a generalized $[\rho]$. Generalizable variables $\mathsf{Gen}\ (B, A, W)$ are free type variables of B that do not appear in A and in different labels of $B \restriction W$. The restriction to labels in W is placed because these labels are those that may synchronize. Conversely, labels in $dom(B) \setminus W$ can only appear in singleton join patterns. Any type variable occurring in two different synchronizing labels, may carry values used for synchronization, and should not be generalized. Formally,

$$\begin{aligned} stv(B, W) &\triangleq \{ftv(B(\ell)) \cap ftv(B(\ell')) \mid \ell \neq \ell',\ \ \ell, \ell' \in dom(B \restriction W)\} \\ \mathsf{Gen}\ (B, A, W) &\triangleq ftv(B) \setminus (ftv(A) \cup stv(B, W)) \end{aligned}$$

We write $\mathsf{Gen}\ (B, A)$ for $\mathsf{Gen}\ (B, A, dom(B))$, that is $ftv(B) \setminus ftv(A)$.

The process P is typed as Q, except that P can also use the private labels of x.

In the class typing rules, all type variables can be generalized, regardless of whether they are used for synchronization or not. This is safe because classes are templates for objects and the synchronization patterns of classes are not definitive, since they may be refined during the inheritance.

Typing rules for patterns. Typing rules for join patterns check that the patterns are well-formed, gather a typing environment for their defined labels, and collect the labels which synchronize.

Messages sent on synchronized labels must agree on the instantiation of their type variables. Since generalized variables can be independently instantiated when a message is sent, any shared variable cannot be generalized (see rule Object). For instance, in the rendez-vous of section 2.1 defined by $get(r)$ & $set(a, s) \triangleright r.reply(a)$ & $s.reply()$, the type of the first arguments of get and set are not generalized.

Figure 10: Typing rules for processes

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

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

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

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

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

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

Typing rules for 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 $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$, out of typings of C and S. It uses the auxiliary judgment for selection clauses that keeps track of labels that are defined in the left hand side and right hand side of patterns. The premise $\oplus_{i \in I} B_i \subseteq B_1$ checks that labels in the selection patterns are all defined in B_1, hence declared in C. On the contrary, the premise $\mathit{dl}(S) \cap \mathit{dom}(B_1) = \emptyset$ implies that names in $\mathit{dl}(S)$ are not already declared in C. In particular, this ensures that the pattern of every refined reaction rule is linear (condition (2) in section 3.3).

Last, the computation of synchronizing labels for selections uses the auxiliary operation $\mathsf{match}\ B^W\ \mathsf{with}\ |^{i \in I}(B_i \Rightarrow {B'_i}^{W'_i})\ \mathsf{end}$ that is defined as

$$W \cup \bigcup_{i \in J} W'_i \cup \bigcup_{i \in I \setminus J} \mathit{dom}(B'_i) \cup \underbrace{\left(\mathit{dom}(B) \setminus \bigcup_{i \in J} \mathit{dom}(B_i)\right)}_{\mathsf{if}\ \exists i \in I, B_i = \emptyset \wedge B'_i \neq \emptyset}$$

where

$$J \triangleq \{i \in I \mid \exists \ell \notin W, \mathit{dom}(B_i) = \{\ell\}\}$$

According to this formula, a label synchronizes if one of the followings hold

1. it already synchronized in the parent class:

Figure 11: Typing rules for patterns

Empty-Pattern

$$A \vdash \mathsf{0} :: \emptyset^{\emptyset}$$

Message-Pattern

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

Synchronization

$$\frac{A \vdash J_1 :: {B_1}^{W_1} \qquad A \vdash J_2 :: {B_2}^{W_2} \qquad [dom(B_1) \cap dom(B_2) = \emptyset]}{A \vdash J_1 \ \&\ J_2 :: (B_1 \oplus B_2)^{dom(B_1 \oplus B_2)}}$$

Alternative

$$\frac{A \vdash J_1 :: {B_1}^{W_1} \qquad A \vdash J_2 :: {B_2}^{W_2} \qquad fn(J_1) = fn(J_2)}{A \vdash J_1 \ \mathsf{or}\ J_2 :: (B_1 \oplus B_2)^{W_1 \cup W_2}}$$

2. it synchronizes in a refinement clause $\ell(\widetilde{u}) \Rightarrow K_i' \triangleright P_i$ and ℓ did not synchronize in the parent class (i.e. $i \in J$);
3. it occurs into a modification pattern K_i' of a refinement clause $K_i \Rightarrow K_i' \triangleright P_i$, such that K_i is empty, or a synchronization, or a synchronized label of the parent class;
4. it is an unsynchronized label of the parent class that may synchronize with the labels of the refinement clauses (when there is a subclause $\emptyset \Rightarrow K \triangleright P$).

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

5.3 Example of typing

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

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

of the form $\mathsf{self}(z)\,(J_1 \triangleright P_1\ \mathsf{or}\ J_2 \triangleright P_2)$. First, consider the typing of pattern J_1. By rules Message-Pattern and Synchronization we have:

$$A_1 \vdash get(r)\ \&\ Some(a) :: {B_1}^{W_1} \tag{1}$$

where set $W_1 = \{get, Some\}$, whereas the type environment A_1 and internal type B_1 are of the forms $A_1 = (r : \theta_1, a : \theta_2)$ and $B_1 = (get : (\theta_1); Some : (\theta_2))$, reflecting the labels and their arity.

However, pattern J_1 is not typed in isolation but as the pattern of a reaction rule whose guarded process P_1 includes $r.reply(a)$. Rule Reaction requires that P_1 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: $\theta_1 = [reply : (\theta); \varrho]$

Figure 12: Typing rules for classes

Class-Var

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

Reaction

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

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

$$\frac{dom(B) = L \qquad W \subseteq L}{A \vdash L :: \zeta(\rho)B^{W,L}}$$

Disjunction

$$\frac{\begin{array}{c} A \vdash C_1 :: \zeta(\rho)B_1^{W_1,V_1} \qquad A \vdash C_2 :: \zeta(\rho)B_2^{W_2,V_2} \\ V_1 \setminus V \subseteq dom(B_2) \setminus V_2 \qquad V_2 \setminus V \subseteq dom(B_1) \setminus V_1 \qquad W_1 \cup W_2 \subseteq W \end{array}}{A \vdash C_1 \;\mathsf{or}\; C_2 :: \zeta(\rho)(B_1 \oplus B_2)^{W,V}}$$

Refinement

$$\frac{\begin{array}{c} A \vdash C :: \zeta(\rho)B_1^{W_1,V_1} \qquad A \vdash S :: (\left|^{i\in I} B_i \Rightarrow B_i'^{W_i'}\right.)^{V_2} \\ \oplus_{i\in I} B_i \subseteq B_1 \qquad B_2' = \oplus_{i\in I} B_i' \qquad dl(S) \cap dom(B_1) = \emptyset \\ V_1 \cup V_2 \subseteq V \qquad \mathsf{match}\; B_1^{W_1} \;\mathsf{with}\; (\left|^{i\in I} B_i \Rightarrow B_i'^{W_i'}\right.) \;\mathsf{end} \subseteq W \end{array}}{A \vdash \mathsf{match}\; C \;\mathsf{with}\; S \;\mathsf{end} :: \zeta(\rho)(B_1 \oplus B_2')^{W,V}}$$

(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 $J_1 \triangleright P_1$:

$$A \vdash J_1 \triangleright P_1 :: \zeta(\rho){B_1}^{W_1,\emptyset} \tag{2}$$

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

$$A \vdash J_2 \triangleright P_2 :: \zeta(\rho){B_2}^{W_2,\emptyset} \tag{3}$$

where $W_2 = \{put, Empty\}$ and $B_2 = (put : (\theta', [reply : (); \varrho']); Empty : ())$.

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

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

The synchronizing labels W are $W_1 \cup W_2$; the internal type B is $B_1 \oplus B_2$. Here, this amounts to $B_1 \cup B_2$ since patterns J_1 and J_2 have no label in common. Hence $B = (get : ([reply : (\theta); \varrho]); Some : (\theta); put : (\theta', [reply : (); \varrho']); Empty : ())$.

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

Figure 13: Typing rules for refinement clauses

Modifier

$$\frac{\left(\begin{array}{ccc} & A'_i \vdash K'_i :: (B'_i)^{W'_i} & A + A'_i \vdash P_i \\ \mathit{dom}(A'_i) = \mathit{fn}(K'_i) & A'_i \vdash K_i :: B_i^W & \mathit{dom}(B_i) \setminus \mathit{dom}(B'_i) \subseteq V \end{array}\right)^{i \in I}}{A \vdash \left|^{i \in I} (K_i \Rightarrow K'_i \triangleright P_i) :: \right|^{i \in I} (B_i \Rightarrow {B'_i}^{W'_i})^V}$$

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:

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

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

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

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

where B is as before (after equating θ and θ'). Observe that, according to Class, all the type variables (ϱ, ϱ' and θ) are generalized. Nevertheless, when *buffer* is instantiated, the object type will not be polymorphic with respect to stored values θ. The row type ρ is yet unconstrained, but rule Obj (for typing object creation) would later enforce that ρ contains exactly the types of the public labels of the class.

5.4 A semantics with privacy

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

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

In the refined chemical solution, every running process is thus equipped with a prefix $\psi_{\#}$, in which ψ is a string of names describing the static environment of the process. Similarly, the definition for every object x is equipped with a prefix $\psi x_{\#}$, in which ψ is of the form $y_1 \ldots y_n$; the prefix indicates that x is in the scope of y_1, y_2, …, and y_n. A well-formed refined chemical solution is such that all prefixes agree on object nesting, *i.e.*, if ψx and φx appear in prefixes, then $\psi = \varphi$. As before, we also assume that there is a single definition for every object in solution. These properties are preserved by chemical rewriting.

We use the refined rules of Figure 14. There are some side conditions: for Obj, x is a fresh name in the solution; for Red, $\psi x_{\#}[M \triangleright P]$ abbreviates

$\psi x \,\#\, D$ for some definition D containing the rule $M \triangleright P$ and σ is a substitution on the names bound in M.

Except for the bookkeeping on environments, rules Nil, Par, Join, Obj, and Red are the same as in section 2. Note that Red consumes only messages with a prefix that matches the object definition, and triggers a process in the same environment as the object definition. Since messages may be sent from other objects, an intermediate *routing step* is called for. Such steps are modeled by rules Public-Comm and Private-Comm that carry messages from their emitter to their receiver. The new semantics is a refinement of the previous one. (Formally, we could show that reductions in the two semantics commute with the addition of privacy annotations.)

Figure 14: Chemical semantics with privacy

Par

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

Nil

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

Join

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

Obj

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

Public-Comm

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

Private-Comm

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

Red

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

Our privacy policy states that a message sent from object y to object x on a private label is valid inasmuch as y is a sub-object of x (*cf.* rule Private-Comm). We take the presence of non-routable messages as our definition of a privacy error. We also give a formal definitions for other error patterns, which do not depend on privacy information.

Definition 1 (Runtime Failure) *A chemical solution* $\mathcal{D} \Vdash \mathcal{P}$ fails *when* $\mathcal{P}$ *contains a message* $\psi \,\#\, x.\ell(\widetilde{u})$ *such that one of the following holds:*

Undefined object: x *is not defined in* $\mathcal{D}$.

Privacy: $\psi' x \,\#\, D \in \mathcal{D}$, $\ell \in \mathcal{F}$, *and* ψ' *is not a prefix of* ψ.

Undefined label: $\psi' x \,\#\, D \in \mathcal{D}$ *and* $\ell \notin dl(D)$.

Arity Mismatch: $\psi' x \,\#\, D \in \mathcal{D}$, $\ell(\widetilde{y})$ *appears in a pattern of* D, *and* $\widetilde{y}$ *and* $\widetilde{u}$ *have different arities.*

5.5 Type checking solutions

To establish subject reduction, we extend typing from programs to chemical solutions. The typing judgment $A \vdash (\mathcal{D} \Vdash \mathcal{P})$ states that the chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is well-typed in environment A. We also use auxiliary judgments $A \vdash \mathcal{D} :: B$ and $A \vdash \mathcal{P}$. We let A^{ψ} be A where the internal bindings $x : \forall X.\, B$ have been removed for all names x that do not appear in ψ.

$$\textsc{Soup}\quad \frac{A + A' \vdash \mathcal{D} :: A' \qquad A + A' \vdash \mathcal{P}}{A \vdash (\mathcal{D} \Vdash \mathcal{P})} \qquad\qquad \textsc{Processes}\quad \frac{(A^{\psi} \vdash P)^{\psi \,\#\, P \in \mathcal{P}}}{A \vdash \mathcal{P}}$$

$$\textsc{Definitions}\quad \frac{\begin{pmatrix} A^{\psi} + x : [\rho_x], x.(B_x \upharpoonright \mathcal{F}) \vdash D :: \zeta(\rho_x) B_x^{W,\emptyset} \\ \rho_x = B_x \upharpoonright \mathcal{M} \qquad X = \mathsf{Gen}\ (B_x, A^{\psi}, W) \end{pmatrix}^{\psi x \,\#\, D \in \mathcal{D}}}{A \vdash \mathcal{D} :: (x : \forall X.[\rho_x], x : \forall X.(B_x \upharpoonright \mathcal{F}))^{\psi x \,\#\, D \in \mathcal{D}}}$$

In the judgment $A \vdash \mathcal{D} :: A'$ of rule Definitions, A' may be a polymorphic typing context. Therefore, rule Soup allows polymorphic type-checking of solutions, as opposed to rule Object.

Remarkable, the above type system may type solutions that are not reachable from objective join calculus processes, such as $x_1.[\ell(u) \triangleright x_2.\ell(u)], x_2.[\ell(u) \triangleright x_1.(u)] \Vdash 0$. Type inference might not be decidable for these solutions, because of recursive dependencies in polymorphic typing assumptions. Nevertheless, type inference is always possible for solutions that can be folded back into processes. As shown by the semantics with privacy, in these latter solutions, objects have a static type which is a total ordering.

Remark that a well-typed solution does not fail and does not violates the privacy. For instance, if $A \vdash (\mathcal{D} \Vdash \mathcal{P})$ and $\mathcal{P}$ contains $\psi \,\#\, x.\ell(\widetilde{u})$ then, by Processes, $A^{\psi} \vdash x.\ell(\widetilde{u})$, which means that $x \in dom(A)$. Therefore, by Definitions, $\varphi x \,\#\, D \in \mathcal{D}$, for some φ and some D.

5.6 Subject reduction with privacy

We define two notions of type preservation: A relation $\subset$ preserves process typings when $A \vdash P$ and $P \subset P'$ implies $A \vdash P'$. Similarly, a relation $\subset$ preserves chemical typings when $A \vdash \mathcal{D} \Vdash \mathcal{P}$ and $(\mathcal{D} \Vdash \mathcal{P}) \subset (\mathcal{D}' \Vdash \mathcal{P}')$ implies $A \vdash \mathcal{D}' \Vdash \mathcal{P}'$.

Theorem 1 states that our operational semantics preserves typings. It is parameterized by a relation δ that accounts for any (well-typed) delta-rules. Its proof, as well as the proofs of the other statements, is deferred to the appendix.

Theorem 1 (Chemistry) *If δ preserves process typings, then chemical steps $\equiv$ and $\longrightarrow$ preserve chemical typings.*

As can be expected, our type system prevents runtime failures:

Theorem 2 (Progress) *A chemical solution that is well-typed in the empty environment never fails.*

In addition, our type system also detects most errors in the reduction semantics for processes. The following results state that reduction for processes preserves typings, and that reduction for well-typed processes can fail only in two simple cases:

Theorem 3 *If* $A \vdash P$ *and* $P \longmapsto P'$*, then* $A \vdash P'$*.*

Theorem 4 *If* $A \vdash P$ *and* $P \not\longmapsto$*, then one of the following cases hold:*

(Completion) P *is a process of the core object calculus (figure 1).*

(Undefined class) P *contains a free class name.*

(Inadequate filter) P *contains* $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$ *such that* $dl(S) \not\subseteq C$*.*

An important consequence of Theorems 1 and 3 is that chemical and reduction transitions may be safely merged, thus leaving the space to separate compilation.

Corollary 1 *Chemical relations* $\equiv$ *and* $\longrightarrow$ *preserve chemical typings modulo process-reduction (*i.e. *taking* $\longmapsto$ *for* δ*).*

6 Related and future works

There are many works on supplementing object calculi with concurrent primitives [16, 3, 11], or on supplementing process calculi with objects (usually by the mean of an encoding in the original calculus) [22, 20, 12, 6]. Our work follows the latter tradition. However, to our knowledge, it is the only one to address safe object composition in a process calculus.

In [17], Odersky proposes an object-oriented extension of a language based on the join calculus. His proposal amounts to adding some record structure to join definitions, as we do in section 2. However, Odersky does not consider the problem of inheritance and refinement of synchronization. A small technical difference is that Odersky's calculus is designed with pattern matching on values: values in Odersky's calculus are not only object names but also constructed values (such as strings, integers, lists, etc.); then, the shape of values can be taken into account during synchronization. For instance, a rule of the form $\ell(h :: t) \triangleright P$ is allowed, and will only fire when a message on ℓ carries a list that contains at least one cell. The extension of our calculus to structured values is easy (see [10]). However, we believe that synchronization should only concern names, and not depend on the shape of values itself. In particular, this allows a simpler semantics and an efficient compilation of synchronization into automata [14].

As regards polymorphism, the generalization for objects (rules OBJECT) corresponds to the generalization implemented in JoCaml [13] for definitions. It is more expressive than the generalization rule used in [10]. It seems equivalent to the generalization rule independently proposed in [5], even though the two formalizations differ. In all of these proposal, type variables that appear in two names that may be used for synchronization are potentially *dangerous* and cannot be generalized, so as to ensure type soundness. In [10], dangerous type variables where computed altogether for all clauses of a definition, which forced to be pessimistic. In both [13] and [5], dangerous type variables are computed rule by rule, leading to more precise information, *i.e.* smaller sets of dangerous variables. In fact, in the objective join calculus, the rule by rule treatment comes comes quite naturally and in a simpler form because the recursion in object definition always go through the variable self and is left open until the class is turned into an object. This amounts to typing $\mathsf{let}\ x_1 = a_1\ \mathsf{and}\ x_2 = a_2\ \mathsf{in}\ a$ as $\mathsf{let}\ x = \mathsf{fix}\ (\lambda(x_1, x_2).(a_1, a_2))\ \mathsf{in}\ a'$ in ML. The latter form separates the constraints in typing the body of the recursion from typing the recursion in two different steps.

However, in the objective join calculus also has a notion of classes that can be instantiated several times creating independent object definitions, and that can be inherited to create further classes. Classes can always be polymorphic, since they do not create any sharing themselves. This is to be compared with the function $\lambda x.\mathsf{ref}\ x$ in ML. On the opposite, when turning a class into an object, the monomorphic restriction should be enforced for all type variables that appear in the type of names that may synchronize. In the objective join-calculus, we decided to keep in the type of each class, the list of names may be used for synchronization. Then, only type variables appearing in those names might be dangerous. This information is an approximation and can safely be increased. By default all names may considered synchronizing. Indeed, this would be equivalent to not keeping track of synchronization is class types; then, the solution would be equivalent to [10].

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

In sequential languages, deep method renaming, *i.e.* including rewrite of recursive calls, (or, more generally, method hiding) can be solved by introducing dictionaries [19] or views [21]. Views would also give other benefits in a concurrent setting. In particular, combining views with multiple inheritance would allow to build a class with two variants of its parent class, having slightly different synchronization policies. This can be used to distinguish between internal calls and external calls, for example. The integration of views with our calculus deserves future investigation.

Our type system has been carefully designed so as to allow type inference. We conjecture that well-typed programs have principal types. However, this remains to be checked formally.

Technically, our type system could easily be refined to describe synchronization patterns of classes. This should allow to detect blocking refinements at compile time. Additionally, allowing the programmer to specify synchronization would facilitate modular programming and reduce synchronization design errors. However, such types would be so precise that a form of subtyping seems unavoidable in order to forget excessive information. Probably, one should then distinguish between exact and partial, positive and negative synchronization information, which is likely to raise typing difficulties. This is another research direction for the future.

7 Conclusions

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

Acknowledgments. This work benefited from discussions with Luca Cardelli, Sylvain Conchon, Fabrice Le Fessant, and François Pottier. In particular, Sylvain Conchon and Fabrice Le Fessant contributed to discussions on the inheritance operators and the selective refinement. We also discussed the relation between generalization and synchonization with Sylvain Conchon and Franois 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] P. D. Blasio and K. Fisher. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, LNCS 1119, pages 406–421, 1996.

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

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

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

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

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

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

[10] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory*, LNCS 1243, pages 196–212, 1997.

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

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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`.

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

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

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

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

[23] 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 Proofs

A.1 Basic properties for the typing

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:*

$$A \vdash \Delta \Leftrightarrow A + x : \sigma \vdash \Delta$$

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 (Weakening) *If any of the judgment $A \vdash C :: \zeta(\rho)B^{W,V}$ holds and $V \subseteq V' \subseteq \mathrm{dom}(B)$ and $W \subseteq W' \subseteq \mathrm{dom}(B)$, then the same judgment holds when replacing W by W' or V by V'.*

If the judgement $A \vdash S :: \zeta(\rho)(B \Rightarrow {B'}^{W_i})^V$ holds, and $W_i \subseteq W'_i\,\mathrm{dom}(B'_i)$, and $V \subseteq V' \subset \oplus^{i \in I}\,\mathrm{dom}(B'_i)$, then the same judgment holds when replacing W_i by W'_i or V by V'.

Lemma 6 (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 $(A \vdash y.m_i : \widetilde{\tau}_i)^{i \in I}$ and $(A \vdash y.\ell : \widetilde{\tau}_\ell)^{\ell \in S}$ then $A \vdash \Delta\{^{y}/_{x}\}$.*

Corollary 7 (Renaming of object variables) *If $A \vdash \Delta$, $y \notin \mathrm{dom}(A)$ and y is not free in Δ, then $A\{^{y}/_{x}\} \vdash \Delta\{^{y}/_{x}\}$.*

Lemma 8 (Substitution of a class name in a term) *Let $A{+}c : \forall X.\zeta(\rho)B^V \vdash \Delta$ and $A{+}B \vdash C :: Y.\zeta(\rho)B^V$ and $X \subseteq \mathit{ftv}(B) \setminus (\mathit{ftv}(A) \cup Y)$. Then $A \vdash \Delta\{^{C}/_{c}\}$.*

A.2 Proofs of subject reduction for the chemical semantics

We prove this property by induction on the number of applications of rule Chemistry and Chemistry-Obj in the derivation of the one-step reduction.

Basic cases. We first consider the basic case for every reaction rule.

Case Nil**:** The equivalence is $\Vdash \psi \# \mathsf{0} \;\equiv\; \Vdash$. Indeed the judgments $A \vdash \Vdash \psi \# \mathsf{0}$ and $A \vdash \Vdash$ are always true, independently, and for any environment A.

Case Par**:** The equivalence is $\Vdash \psi \# (P \,\&\, Q) \equiv \Vdash \psi \# P, \psi \# Q$. The two judgments $A \vdash \psi \# (P \,\&\, Q)$ and $A \vdash \Vdash \psi \# P, \psi \# Q$ are both equivalent to the conjunction of the judgments $A^{\psi} \vdash P$ and $A^{\psi} \vdash Q$ (by rules Par and Soup).

Case Join**:** This is similar to the above case, and rely on rules Soup and Par and Join.

Case Obj**:** The rule equates the two solutions $\Vdash \psi \# \mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ and $\psi x' \# D\{x'/z\} \Vdash \psi x' \# P\{x'/x\}, \psi \# Q\{x'/x\}$. For sake of simplicity, we assume that processes have been renamed so that x can be chosen for x' (x does not occur in D).

On the one hand, a derivation of $A \vdash \;\Vdash \psi \# \mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q$ (1) must have the form:

$$\textsc{Object}\ \dfrac{\begin{array}{c} A^{\psi} \vdash \mathsf{self}(z)\, D :: \zeta(\rho)B^{W,\emptyset}\ (6) \\ X = \mathsf{Gen}\ (B, A^{\psi}, W)\ (5) \qquad A^{\psi} + x : \forall X.[\rho], x : \forall X.(B \restriction \mathcal{F}) \vdash P\ (4) \\ \rho = B \restriction \mathcal{M}\ (3) \qquad A^{\psi} + x : \forall X.[\rho] \vdash Q\ (2) \end{array}}{\textsc{Soup}\ \dfrac{A^{\psi} \vdash \mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q}{(1)}}$$

where (6) is derived by:

$$\textsc{Self-Binding}\ \dfrac{A^{\psi} + z : [\rho], z.(B \restriction \mathcal{F}) \vdash D :: \zeta(\rho)B^{W,\emptyset}\ (7)}{A^{\psi} \vdash \mathsf{self}(z)\, D :: \zeta(\rho)B^{W,\emptyset}}$$

On the other hand, a derivation of $A' \vdash \psi x \# D\{x/z\} \Vdash \psi x \# P, \psi \# Q$ (8) must be of the form:

$$\dfrac{\dfrac{\begin{array}{c} {A'}^{\psi} + x : [\rho], x.(B' \restriction \mathcal{F}) \vdash D\{x/z\} :: \zeta(\rho){B'}^{W',\emptyset}\ (13) \\ \rho = B' \restriction \mathcal{M}\ (12) \qquad X' = \mathsf{Gen}\ (B', {A'}^{\psi}, W')\ (11) \end{array}}{A' \vdash \psi x \# D\{x/z\} :: A''}\ \textsc{Definitions} \qquad \begin{array}{c} \\ {A'}^{\psi x} \vdash P\ (10) \qquad {A'}^{\psi} \vdash Q\ (9) \end{array}}{(8)}\ \textsc{Soup}$$

where $A'' = x : \forall X'.[\rho] \oplus x : \forall X'.(B' \restriction \mathcal{F})$. Note that since x is not free $\mathsf{obj}\ x = \mathsf{self}(z)\, D\ \mathsf{init}\ P\ \mathsf{in}\ Q$, we may assume that A does not bind x. Let us assume that A' is equal to $A + x : \forall X.[\rho], x : \forall X.(B' \restriction \mathcal{F})$.

Then, ${A'}^{\psi x}$ is equal to $A^{\psi x} + x : \forall X.[\rho], x : \forall X.(B' \restriction \mathcal{F})$ (14) and ${A'}^{\psi}$ is equal to $A^{\psi} + x : \forall X.[\rho]$ (15). Observe that the following premises may be made equivalent to one another:

(7) ≡ (13): By corollary 7, hence we may identify B' with B and W' with W.

(5) ≡ (11): By definition, we have

$$X = \mathit{ftv}(B) \setminus (\mathit{ftv}(A^{\psi}) \cup \mathit{stv}(B, W))$$

$$X' = \mathit{ftv}(B) \setminus (\mathit{ftv}({A'}^{\psi}) \cup \mathit{stv}(B, W))$$

Since ${A'}^{\psi}$ is equal to $A^{\psi} \oplus x : \forall X.\rho$, we have $X' \subseteq X$. Furthermore,

$$\begin{aligned} X \setminus X' &= (\mathit{ftv}(B) \setminus \mathit{stv}(B, W)) \cap \mathit{ftv}(\forall X.\rho) \\ &\subseteq \mathit{ftv}(\forall X.\rho) \\ &\subseteq \mathit{ftv}(B) \setminus X \\ &\subseteq \mathit{ftv}(B) \setminus (\mathit{ftv}(B) \setminus (\mathit{ftv}(A^{\psi}) \cup \mathit{stv}(B, W))) \\ &= \underbrace{\mathit{ftv}(B) \cap (\mathit{ftv}(A^{\psi}) \cup \mathit{stv}(B, W))}_{\text{does not intersect } X} \end{aligned}$$

Thus, $X \subseteq X'$.

(4) ≡ (10): By (14)

(2) ≡ (9): By (15)

(3) ≡ (12): they are identical

As a result the judgments (1) and (8) are also equivalent.

To conclude, if (1) holds, then (8) holds by taking $A + x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F})$ for A', and conversely, if (8) holds, then (1) holds by taking $A' \setminus x$ for A.

Case Public-Comm**:** The reduction is $\psi x \,\#\, D \Vdash \psi' \,\#\, x.m(\widetilde{u}) \longrightarrow \psi x \,\#\, D \Vdash \psi x \,\#\, x.m(\widetilde{u})$. Let us assume that $A \vdash \psi x \,\#\, D \Vdash \psi' \,\#\, x.m(\widetilde{u})$. That is, $A + A' \vdash \psi \,\#\, D :: A'$ (1) and $A + {A'}^{\psi'} \vdash x.m(\widetilde{u})$ (2). A derivation of (2) can only be:

$$\text{\textsc{Send}}\ \dfrac{\text{\textsc{Message}}\ \dfrac{A'' \vdash x : [m : ({\tau_i}^{i\in I}); \rho]}{A'' \vdash x.m : ({\tau_i}^{i\in I})} \qquad (A'' \vdash x_i : \tau_i)^{i\in I}}{A'' \vdash x.m({x_i}^{i\in I})}$$

where A'' is $(A + A')^{\psi'}$. Since this derivation does not uses any internal type assumption of A'', it is not affected by replacing A' with $A^{\psi x}$. Thus, $(A + A')^{\psi x} \vdash x.m({x_i}^{i\in I})$ holds combining with (1), we obtain $A \vdash \psi x \,\#\, D \Vdash \psi x \,\#\, x.m(\widetilde{u})$ by rule Soup.

Case Private-Comm**:** The reduction is $\psi x \,\#\, D \Vdash \psi x \psi' \,\#\, x.f(\widetilde{u}) \longrightarrow \psi x \,\#\, D \Vdash \psi x \,\#\, x.f(\widetilde{u})$. Let us assume that $A \vdash \psi x \,\#\, D \Vdash \psi x \psi' \,\#\, x.f(\widetilde{u})$. That is, $A + A' \vdash \psi \,\#\, D :: A'$ (1) and $A + {A'}^{\psi x \psi'} \vdash x.f(\widetilde{u})$ (2).

A derivation of (2) can only be:

$$\text{\textsc{Send}}\ \dfrac{\overset{\text{[\textsc{Private-Message}]}}{\dfrac{x : \forall X.(f : ({\tau_i}^{i\in I}); B) \in A''}{A'' \vdash x.f : (\tau_i\{\gamma_\alpha/\alpha^{\alpha\in X}\}^{i\in I})}} \qquad (A'' \vdash u_i : \tau_i\{\gamma_\alpha/\alpha^{\alpha\in X}\})^{i\in I}}{A'' \vdash x.f({u_i}^{i\in I})}$$

where A'' is $(A + A')^{\psi x \psi'}$. The only internal type assumption in the premise is on x, which remains in $A + A'^{\psi x}$. Thus, we can still replace A'' by $(A + A')^{\psi x}$ in the above derivation and, as above, conclude with rule Soup that $A \vdash \psi x_{\#} D \Vdash \psi x_{\#} x.f(\widetilde{u})$.

Case Red: The reduction is $\psi x_{\#} D \Vdash \psi x_{\#} x.(M\sigma) \longrightarrow \psi x_{\#} D \Vdash \psi x_{\#} (P\sigma)$, where D is of the form $M \rhd P$ or D' and M is of the form $\&_{i \in I}\ \ell_i(\widetilde{x}_i)$. Let $A \vdash \psi x_{\#} D \Vdash \psi x_{\#} x.(M\sigma)$. For simplicity, by lemma 2, we may always assume that x does not appear in A. This derivation must end with rule Definitions and Soup with the premises.

$$
\begin{align}
A' \triangleq x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F}) \tag{3}\\
\rho = B \upharpoonright \mathcal{M} \tag{4}\\
X = \mathsf{Gen}\ (B, A^{\psi}, W) \tag{5}\\
\underbrace{A^{\psi} + x : [\rho], x : (B \upharpoonright \mathcal{F})}_{\triangleq A^{\psi}_{+x}} \vdash D :: \zeta(\rho) B^{W,\emptyset} \tag{6}\\
A^{\psi} + A' \vdash x.(M\sigma) \tag{7}
\end{align}
$$

Observe that we have replaced $(A + A')^{\psi x}$ by $A^{\psi} + A'$ in (7), which is equal since $x \notin dom(A)$ and $dom(A') = \{x\}$. We have also simplified $ftv((A + A')^{\psi})$ by $ftv(A^{\psi})$ in (5) since it does not change the resulting set X. To demonstrate that $A \vdash \psi x_{\#} D \Vdash \psi x_{\#} (P\sigma)$, it suffices to show that $A^{\psi} + A' \vdash P\sigma$, since the left hand side is unchanged. The derivation of (7) ends with a rule Join. Hence, for each message $\ell_i(\sigma(\widetilde{x}_i))$ of M, we have $A^{\psi} + A' \vdash x.\ell_i(\sigma(\widetilde{x}_i))$, *i.e.*, by Send, $A^{\psi} + A' \vdash x.\ell_i : \widetilde{\tau}_i$ (8) and $A^{\psi} + A' \vdash \sigma(\widetilde{x}_i) : \widetilde{\tau}_i$ (9). The judgment (8) is derived by either rule Private-Message or Message, depending on whether ℓ is private or public. It results that 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 (6) must contain a sub-derivation

$$
\dfrac{\textsc{Reaction}\qquad \begin{array}{c} A'' \vdash M :: B_0^{W'_0}\ (11) \\ A^{\psi}_{+x} + A'' \vdash P\ (10) \qquad dom(A'') = fn(M) \qquad W'_0 \subseteq W_0 \end{array}}{A^{\psi}_{+x} \vdash M \rhd P :: \zeta(\rho) B_0^{W_0,\emptyset}\ (12)}
$$

(the judgment (12) is then combined with other judgments for D' by a sequence of Disjunction rules to end up with (6)). The internal type B_0 and the set W_0 are therefore subset of B and W, respectively. As regards the premise (11), it is derived by a combination of rules Synchronization, Message-Pattern, and Empty-Pattern. Hence, we have:

$$A'' \vdash \ell_i(\widetilde{x}_i) :: (\ell_i : B(\ell_i))^{\emptyset}\ (13)$$

for every $i \in I$, and where $W'_0 = \{\ell_i \mid i \in I\}$. In any case, for every $i, j \in I$, $i \neq j$, we have $ftv(B(\ell_i)) \cap ftv(B(\ell_j)) \cap X = \emptyset$. 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}. Applying lemma 3 to (10), we have $\eta(A^{\psi}_{+x}) + \eta(A'') \vdash P$, 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.* $A^{\psi} + A' + \eta(A'') \vdash P$. To conclude, note that $(A^{\psi} + A' \vdash \sigma(x_i) : \eta(\tau'_i))^{x_i : \tau'_i \in A''}$ by the collection of judgments (9). Thus, by lemma 6, we derive $A^{\psi} + A' \vdash P\sigma$.

Case Delta**:** The reduction is $\Vdash P \longrightarrow \Vdash P'$, where $(P, P') \in \delta$. The property straightforwardly follows from the fact that δ preserves typings, that is δ is a sub-relation of $\subset$. Indeed, this implies that $P \subset P'$, namely, that any typing of P is also a yping of P'.

Inductive cases. We factor the two cases Chemistry and Chemistry-Obj. Therefore, let $\mathcal{D}_1 \Vdash \mathcal{P}_1$ and $\mathcal{D}_2 \Vdash \mathcal{P}_2$ either

1. be $\Vdash P$ and $x.D \Vdash \mathcal{P}'$ taken in any order (rule Chemistry-Obj), or
2. be such that $\mathcal{D}_1 = \mathcal{D}_2$ (rule Chemistry).

We write $ov(\mathcal{D})$ for the set $\{x \mid x.D \in \mathcal{D}\}$ of object names defined in $\mathcal{D}$. Let us assume

$$A \vdash \mathcal{D}, \mathcal{D}_1 \Vdash \mathcal{P}, \mathcal{P}_1 \tag{1}$$

$$\mathcal{D}_1 \Vdash \mathcal{P}_1 \Rrightarrow \mathcal{D}_2 \Vdash \mathcal{P}_2 \tag{2}$$

$$(ov(\mathcal{D}_1) \cup ov(\mathcal{D}_2)) \setminus (ov(\mathcal{D}_1) \cap ov(\mathcal{D}_2)) \;\cap\; (fn(\mathcal{D}) \cup fn(\mathcal{P})) = \emptyset \tag{3}$$

Actually, the condition (3) only requires that $x \notin fn(\mathcal{D}) \cup fn(\mathcal{P})$ in the case of Chemistry-Obj. We must show that $A \vdash \mathcal{D}, \mathcal{D}_2 \Vdash \mathcal{P}, \mathcal{P}_2$ (4).

A derivation of (1) must end with Soup and premises:

$$A + (A_0 + A_1) \vdash \mathcal{D}, \mathcal{D}_1 :: A_0 + A_1 \qquad A + (A_0 + A_1) \vdash \mathcal{P}, \mathcal{P}_1 \tag{5}$$

where the domains of A_0 and A_1 are $ov(\mathcal{D})$ and $ov(\mathcal{D}_1)$, respectively. In turn, the derivations of (5) end with rules definitions and processes with premises:

$$A + A_0 + A_1 \vdash \mathcal{D} :: A \qquad A + A_0 + A_1 \vdash \mathcal{P} \tag{6}$$

$$A + A_0 + A_1 \vdash \mathcal{D}_1 :: A_1 \qquad A + A_0 + A_1 \vdash \mathcal{P}_1 \tag{7}$$

Applying Soup to (7), we get $A_0 + A \vdash (\mathcal{D}_1 \Vdash \mathcal{P}_1)$. Using the induction hypothesis with (2), we therefore obtain $A_0 + A \vdash (\mathcal{D}_2 \Vdash \mathcal{P}_2)$ (8). This latter judgment can only follow by Soup with premises:

$$(A + A_0) + A_2 \vdash \mathcal{D}_2 :: A_2 \qquad (A + A_0) + A_2 \vdash \mathcal{P}_2 \tag{9}$$

By lemma 2, applied to the two judgments of (6), thanks to the condition (3), we get:

$$A + (A_0 + A_2) \vdash \mathcal{D} :: A \qquad A + (A_0 + A_2) \vdash \mathcal{P} \tag{10}$$

The combination of (10) and (9) gives

$$A + (A_0 + A_2) \vdash \mathcal{D}, \mathcal{D}_2 :: A_0 + A_2 \qquad A + (A_0 + A_2) \vdash \mathcal{P}, \mathcal{P}_2 \tag{11}$$

We conclude (4) from (11) and Soup. $\square$

A.3 Progress at runtime and privacy

Progress at runtime. By Theorem 1, if $\mathcal{D} \Vdash \psi_{\#}x.\ell(\widetilde{u})), \mathcal{P}$ is a solution reachable from a well-typed (in the empty environment) solution then $\vdash (\mathcal{D} \Vdash \psi_{\#}x.\ell(\widetilde{u})), \mathcal{P}$. This judgment must follow by Soup and Definition, which require:

$$A^{\psi} \vdash x.\ell(\widetilde{u}) \ (1) \qquad A \vdash \mathcal{D} :: A \ (2) \qquad dom(A) = \{x \mid \psi' x.D \in \mathcal{D}\} \ (3)$$

Let us check that none of the cases of runtime failure can occur.

1. (**no undefined object**) We show that x is necessarily defined in $\mathcal{D}$. A derivation of (1) must end with

$$\dfrac{\dfrac{\cdots}{A^{\psi} \vdash x.\ell : ({\tau_i}^{i\in I})}\,(4) \qquad (A^{\psi} \vdash u_i : \tau_i)^{i\in I}}{A^{\psi} \vdash x.\ell({u_i}^{i\in I})}\ \textsc{Send}$$

   where (4) is either an instance of Message or Private-Message. This requires that A contains $x : \forall X.[\ell : ({\tau_i}^{i\in I}); \rho]$ or $x : \forall X.(\ell : ({\tau_i}^{i\in I}); B)$, respectively. In each case x is in the domain of A, and by (3), there must be some $\psi' x.D$ in $\mathcal{D}$.

2. (**no privacy leak**) Let ℓ be a private label f and let $\psi' x.D$ in $\mathcal{D}$. We prove that ψ is a prefix of ψ'.

   By the derivation of (1), the premise of (4) 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, a name can have a unique prefix. Therefore, since ψ' is already a prefix of x, then ψ must be of the form $\psi' x \psi''$.

3. (**no undefined label**) We now assume that $\psi' x.D$ in $\mathcal{D}$ (5) and we show that $\ell \in \mathit{dl}(D)$.

   The judgment (2) must be proved with rule Definitions. Given (5), we have at least the following premises:

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

$$X = \mathsf{Gen}\ (B, A^{\psi'}, W) \qquad x : \forall X.[\rho] \in A\ (8) \qquad x : \forall X.(B \upharpoonright \mathcal{F}) \in A\ (9)$$

   By (1), (8) and (9), either $\rho = [\ell : ({\tau_i}^{i\in I}); \rho']$ or $B = (\ell : ({\tau_i}^{i\in I}); B')$. In each case, using (7), ℓ is in $\mathit{dom}\,(B)$. We conclude by inspecting the derivation of (6), because B only contains labels defined in D.

4. (**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

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

   where W' is $\emptyset$ if $\mathit{card}\,(\mathit{dom}(B)) = 1$ and W' is $\mathit{dom}(B)$ otherwise. Moreover $W \subseteq \mathit{dom}(B)$. Therefore, the type of $\widetilde{y}$ in A' is $B(\ell)$. By rule Definitions, A contains a generalization of B; thus, the type of $x.\ell$ in A^{ψ} is a generalization of $B(\ell)$. The proof tree illustrated in the item 1 is required to prove (1). Then, as a consequence of rule (4), 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'.

$\square$

A.4 Proofs of subject reduction for the class semantics

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

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

$$C \ \mathsf{with}\ S \longmapsto C' \qquad A \vdash C :: \zeta(\rho)B^{W,V} \qquad A \vdash S :: \left|^{i\in I}(B''_i \Rightarrow {B'_i}^{W'_i})^{V'}\right. \\ \mathit{dl}(S) \cap \mathit{dom}(B) = \emptyset$$

and $B' = \bigoplus^{i\in I} B'_i$ *and* $B'' = \bigoplus^{i\in I} B''_i$, *then* $A \vdash C' :: \zeta(\rho)(B \oplus (B' \setminus L))^{W'',V''}$ *for some* $L \subseteq (\mathit{dl}(S) \setminus \mathit{dl}(C')) \cup (\mathit{dom}(B'') \setminus \mathit{dom}(B))$, $V'' \subseteq V \cup V'$, *and* $W'' \subseteq$ $\mathsf{match}\ B^W\ \mathsf{with}\ \left|^{i\in I}(B''_i \Rightarrow {B'_i}^{W'_i})\right.\ \mathsf{end}$.

In this proof, we abbreviate $\mathit{dom}(B)$ by $\overline{B}$, for sake of conciseness.

Basic cases.

Case Filter-Apply**:** Let us assume that

$$\begin{aligned} &K \mathbin{\&} K'' \triangleright P \ \mathsf{with}\ K'' \Rightarrow K' \triangleright Q \mid S \\ &\qquad \longmapsto K \mathbin{\&} K' \triangleright P \mathbin{\&} Q \ \mathsf{or}\ \mathit{dl}(K'') \setminus \mathit{dl}(K') && (1)\\ &\qquad A \vdash K \mathbin{\&} K'' \triangleright P :: \zeta(\rho){B_0}^{W_0,V} && (2)\\ &A \vdash K'' \Rightarrow K' \triangleright Q \mid S :: \left|^{i\in I}(B''_i \Rightarrow {B'_i}^{W'_i})^{V'}\right. && (3)\\ &\quad ((\mathit{dl}(K') \setminus \mathit{dl}(K'')) \cup \mathit{dl}(S)) \cap \overline{B_0} = \emptyset && (4)\end{aligned}$$

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

$$\textsc{Synchronization}\ \dfrac{\begin{array}{c} A' \vdash K'' :: {B''}^{W''}\ (10) \\ A' \vdash K :: B^W\ (9) \qquad \overline{B''} \cap \overline{B} = \emptyset\ (8)\end{array}}{A' \vdash K \mathbin{\&} K'' :: \zeta(\rho)(B \oplus B'')^{W_1}\ (11)}$$

$$\textsc{Reaction}\ \dfrac{A + A' \vdash P\ (7) \qquad \overline{A'} = \mathit{fn}(K'' \mathbin{\&} K)\ (6) \qquad W_1 \subseteq W_0\ (5)}{A \vdash K'' \mathbin{\&} K \triangleright P :: \zeta(\rho)(B \oplus B'')^{W_0,V}}$$

and,

$$\textsc{Modifier}\ \dfrac{\begin{array}{c} \cdots \qquad A'' \vdash K'' :: {B''}^{W''}\ (16) \qquad A'' \vdash K' :: {B'}^{W'}\ (15) \\ \overline{A''} = \mathit{fn}(K')\ (14) \qquad A + A'' \vdash Q\ (13) \qquad \overline{B''} \setminus \overline{B'} \subseteq V'\ (12)\end{array}}{A \vdash K'' \Rightarrow K' \triangleright Q \mid S :: (\left|^{i\in I}(B_i \Rightarrow (B'_i)^{W'_i})\right.)^{V'}}$$

where

- for some $k \in I$, $B_k = B''$ and $B'_k = B'$;
- $B_0 \triangleq B \oplus B''$, $B'_0 \triangleq \oplus^{i\in I} B_i$, and $B''_0 \triangleq \oplus^{i\in I} B'_i$;

- W_1 is W'' if K is $\mathsf{0}$, W_1 is W if K'' is $\mathsf{0}$, and $\overline{B''} \cup \overline{B}$ otherwise. (Remember, that K and K'' cannot be both $\mathsf{0}$).

From (10) and (16), A' and A'' coincide on $\mathit{fn}(K'')$ because they assign the same types to $\mathit{fn}(K'')$. Moreover, due to the scope rules of reaction rules and the selection operator, we may safely assume that $\overline{A'} \cap \overline{A''} = \mathit{fn}(K'')$. Thus, by lemma 2 applied to (9), and (15) we derive $A' + A'' \vdash K :: B^W$ (17) and $A' + A'' \vdash K' :: {B'}^{W'}$ (18). Similarly, by lemma 2 applied to (7) and (13), we also derive $A + A' + A'' \vdash P$ (19) and $A + A' + A'' \vdash Q$ (20).

Notice that, by definition, $\mathit{fn}(K') \setminus \mathit{fn}(K'') \subseteq \mathit{dl}(S)$ and, by (15) and (16), $\overline{B'} = \mathit{fn}(K')$ and $\overline{B''} = \mathit{fn}(K'')$. Hence $\overline{B'} \setminus \overline{B''} \subseteq \mathit{dl}(S)$. On the other hand, since $B_0 = B'' \oplus B$, we have $\overline{B} \subseteq \overline{B_0}$. It then follows from (4) that $(\overline{B'} \setminus \overline{B''}) \cap \overline{B} = \emptyset$. Combining this result with (8), we get $\overline{B'} \cap \overline{B} = \emptyset$ (21). By Synchronization with premises (18), (9), and (21), we derive

$$A' + A'' \vdash K \,\&\, K' :: (B \oplus B')^{W_3}. \tag{22}$$

where W_3 is W if K is $\mathsf{0}$, W' if K' is $\mathsf{0}$, and $\overline{B} \cup \overline{B'}$ otherwise.

Combining the judgments (19) and (20) yields $A + A' + A'' \vdash P \,\&\, Q$ (23) using rule Par. By premises (6) and (14) we have $\overline{A'} \cup \overline{A''} = \mathit{fn}(K) \cup \mathit{fn}(K') \cup \mathit{fn}(K'')$. By (14) and (16), we have $\mathit{fn}(K'') \subset \mathit{fn}(K')$. Hence $\overline{A' + A''} = \mathit{fn}(K \,\&\, K')$ (24). Therefore, by Reaction with premises (22), (23), and (24) we derive

$$A \vdash K \,\&\, K' \triangleright P \,\&\, Q :: \zeta(\rho)(B \oplus B')^{W_3, \emptyset} \qquad (25).$$

By rule Abstract, we also deduce

$$A \vdash L' :: \zeta(\rho)(B'' \upharpoonright L')^{\emptyset, L'} \tag{26}$$

where $L' = \mathit{dl}(K'') \setminus \mathit{dl}(K') = \overline{B''} \setminus \overline{B'}$ (27). Hence, Or allows to derive:

$$A \vdash K \,\&\, K' \triangleright P \,\&\, Q \;\mathsf{or}\; L' :: \zeta(\rho)(B \oplus B' \oplus (B'' \upharpoonright L'))^{W_3, L'} \qquad (28).$$

To conclude, we must

1. check that W_3 is a subset of $W_4 \triangleq \mathsf{match}\; B_0^{W_0} \;\mathsf{with}\; (B'' \Rightarrow (B')^{W'} \mid {|}^{i \in I} (B''_i \Rightarrow {B'_i}^{W'_i}))^V \;\mathsf{end}$ (29);
2. find L such that $B \oplus B' \oplus (B'' \upharpoonright L') = B_0 \oplus (B'_0 \setminus L)$ (30) and
$$L \subseteq (((\mathit{dl}(K') \setminus \mathit{dl}(K'')) \cup \mathit{dl}(S)) \setminus \mathit{dl}(C')) \;\cup\; (\overline{B''_0} \setminus \overline{B_0}) \tag{31}$$
3. show that $L' \subseteq V \cup V'$.

As regards 1, let J be $\{i \in I \mid \exists \ell \notin W, B''_i = \{\ell\}\}$. By definition,

$$W_4 \triangleq W_0 \cup \overbrace{\bigcup_{i \in J} W'_i}^{W^1} \cup \overbrace{\bigcup_{i \in I \setminus J} \overline{B'_i}}^{W^2} \qquad \underbrace{\overbrace{\cup \left(\overline{B_0} \setminus \bigcup_{i \in J} \overline{B'_i} \right)}^{W^3}}_{\mathsf{if}\ \exists i \in I, B''_i = \emptyset \wedge B'_i \neq \emptyset}$$

We have the following case analysis:

K	K''	K'	W_0	W_3	$W_3 \subseteq W_4$
0	–	–	W''	W'	$W' \subseteq \overline{B'_k} \subseteq (W^1 \cup W^2)$
–	0	0	W	W	$W_3 = W_0$
–	0	–	W	$\overline{B} \cup \overline{B'}$	$\overline{B}' \cup \overline{B}' \subseteq W_0$
–	–	–	$\overline{B} \cup \overline{B''}$	$\overline{B} \cup \overline{B'}$	$\overline{B} \subseteq W_0 \wedge \overline{B'} \subseteq W^2$

As regards 2, after inlining B_0, the equality (30) becomes $B \oplus B' \oplus (B'' \upharpoonright L') = B \oplus B'' \oplus (B'_0 \setminus L)$. Since all type environments agree, the equality is verified by taking

$$L = \overline{B'_0} \setminus (\overline{B} \cup \overline{B'} \cup \overline{B''}) \tag{32}$$

Let us check whether this solution satisfies (31). Replacing $dl(C')$ by $\overline{B'} \cup \overline{B}$, replacing $dl(K'')$ and $dl(K')$ by $\overline{B''}$ and $\overline{B}$, and inlining $\overline{B_0}$, the bound of L in (31) becomes:

$$\begin{aligned}
&((\overline{B'} \setminus \overline{B''}) \cup dl(S)) \setminus (\overline{B'} \cup \overline{B}) \\
&\cup\ \overline{B'_0} \setminus (\overline{B} \cup \overline{B''}) \\
= {}& dl(S) \setminus (\overline{B'} \cup \overline{B})\ \cup\ \overline{B'_0} \setminus (\overline{B} \cup \overline{B''})
\end{aligned} \tag{33}$$

Indeed, (33) is a superset of L defined in (32).

Finally, 3 follows from (27) and (12).

Case Filter-Next**:** Let us assume

$$\begin{array}{cr}
M \triangleright P \text{ with } K''_1 \Rightarrow K'_1 \triangleright P_1 \mid S \longmapsto C' & (1) \\
A \vdash M \triangleright P :: \zeta(\rho) B^{W,V} & (2) \\
A \vdash K''_1 \Rightarrow K'_1 \triangleright P_1 \mid S :: (\big|^{i \in 1..n} (B''_i \Rightarrow (B'_i)^{W'_i}))^{V'} & (3) \\
((dl(K'_1) \setminus dl(K''_1)) \cup dl(S)) \cap \overline{B} = \emptyset & (4) \\
S = \big|^{i \in 2..n} (K''_i \Rightarrow K'_i \triangleright P_i) &
\end{array}$$

A derivation of (3) must end with an instance of Modifier, with premises:

$$\left(\begin{array}{ccc} & A'_i \vdash K'_i :: (B'_i)^{W'_i} & A + A'_i \vdash P_i \\ \overline{A'_i} = fn(K'_i) & A'_i \vdash K''_i :: (B''_i)^{W''_i} & \overline{B''_i} \setminus \overline{B'_i} \subseteq V \end{array} \right)^{i \in 1..n}$$

where

$$\begin{aligned}
B'' &= \quad B''_1 \oplus B''_S \\
B' &= \quad B'_1 \oplus B'_S \\
B''_S &= \quad \textstyle\bigoplus_{i \in 2..n} B''_i \\
B'_S &= \quad \textstyle\bigoplus_{i \in 2..n} B'_i
\end{aligned}$$

Hence, by rule Modifier, we derive

$$A \vdash S :: (\big|^{i \in 2..n} (B''_i \Rightarrow (B'_i)^{W'_i}))^V \tag{5}$$

A derivation of (1) must end with an instance of rule Filter-Next, hence $M \triangleright P \text{ with } S \longmapsto C'$ (6). By induction hypothesis applied to (2), (5), (4), and

(6), we have

$$\begin{aligned}
A \vdash C' :: \zeta(\rho)(B \oplus (B'_S \setminus L_S))^{W_2, V''} \quad & (7)\\
L_S \subseteq (\mathit{dl}(S) \setminus \mathit{dl}(C')) \cup (\overline{B''_S} \setminus \overline{B}) \quad & (8)\\
V'' \subseteq V \cup V' \quad & (9)\\
W_2 \subseteq \mathsf{match}\ B^W\ \mathsf{with}\ \big|^{i\in 2..n}(B''_i \Rightarrow {B'_i}^{W'_i})\ \mathsf{end} \quad & (10)
\end{aligned}$$

We must prove that $A \vdash C' :: \zeta(\rho)(B \oplus (B' \setminus L))^{W_1, V''}$ such that

1. $L \subseteq ((\mathit{dl}(K'_1) \setminus \mathit{dl}(K''_1)) \cup \mathit{dl}(S)) \setminus \mathit{dl}(C') \cup (\overline{B''} \setminus \overline{B})$ (11) and
2. $W_1 \subseteq \mathsf{match}\ B^W\ \mathsf{with}\ \big|^{i\in I}(B''_i \Rightarrow {B'_i}^{W'_i})\ \mathsf{end}$.

To show the former, let $L = L_S \cup \overline{B'_1} \setminus (\overline{B''} \cup \overline{B'_S})$. We first prove that

$$B \oplus (B'_1 \oplus B'_S) \setminus L = B \oplus (B'_S \setminus L_S)$$

Since all type environments agree, it suffices to check that $\overline{B} \cup (\overline{B'_1} \cup \overline{B'_S}) \setminus L = \overline{B} \cup (\overline{B'_S} \setminus L_S)$. This follows by

$$\begin{aligned}
& (\overline{B'_1} \cup \overline{B'_S}) \setminus L\\
= \ & (\overline{B'_1} \cup \overline{B'_S}) \setminus (L_S \cup (\overline{B'_1} \setminus (\overline{B''} \cup \overline{B'_S})))\\
= \ & \overline{B'_1} \setminus (\overline{B'_1} \setminus (\overline{B''} \cup \overline{B'_S})) \setminus L_S \ \cup \ \overline{B'_S} \setminus L_S\\
= \ & (\overline{B'_1} \cap (\overline{B''} \cup \overline{B'_S})) \setminus L_S \ \cup \ \overline{B'_S} \setminus L_S\\
= \ & (\overline{B'_1} \cap \overline{B''}) \setminus L_S \cup \underbrace{(\overline{B'_1} \cap \overline{B'_S}) \setminus L_S}_{\subseteq \overline{B'_S} \setminus L_S} \ \cup \ \overline{B'_S} \setminus L_S\\
= \ & \underbrace{(\overline{B'_1} \cap \overline{B''}) \setminus L_S}_{\subseteq \overline{B}} \ \cup \ \overline{B'_S} \setminus L_S
\end{aligned}$$

and remarking that $\overline{B} \cup (\overline{B'_1} \cap \overline{B''}) \setminus L_S \cup \overline{B'_S} \setminus L_S = \overline{B} \cup \overline{B'_S} \setminus L_S$. Then, the containment (11) follows by remarking that $\mathit{dl}(C') \subseteq \overline{B} \cup \overline{B'_S}$ (12) and

$$\begin{aligned}
L &= L_S \ \cup \ \overline{B'_1} \setminus (\overline{B''} \cup \overline{B'_S}) & \\
&= L_S \ \cup \ (\overline{B'_1} \setminus (\overline{B''_1} \cup \overline{B''_S} \cup \overline{B'_S})) & \text{by (8)}\\
&= L_S \ \cup \ (\overline{B'_1} \setminus \overline{B''_1}) \setminus (\overline{B''_S} \cup \overline{B'_S}) & \\
&= L_S \ \cup \ (\mathit{dl}(K'_1) \setminus \mathit{dl}(K''_1)) \setminus (\overline{B''_S} \cup \overline{B'_S}) & \\
&= L_S \ \cup \ (\mathit{dl}(K'_1) \setminus \mathit{dl}(K''_1)) \setminus (\overline{B} \cup \overline{B''_S} \cup \overline{B'_S}) & \text{by (4)}\\
&\subseteq L_S \ \cup \ (\mathit{dl}(K'_1) \setminus \mathit{dl}(K''_1)) \setminus \mathit{dl}(C') & \text{by (12)}\\
&\subseteq (\mathit{dl}(S) \setminus \mathit{dl}(C') \ \cup \ \overline{B''_S} \setminus \overline{B}) \ \cup \ (\mathit{dl}(K'_1) \setminus \mathit{dl}(K''_1)) \setminus \mathit{dl}(C') & \text{by (8)}\\
&= ((\mathit{dl}(K''_1) \setminus \mathit{dl}(K_1)) \cup \mathit{dl}(S)) \setminus \mathit{dl}(C') \ \cup \ \overline{B''_S} \setminus \overline{B} & \\
&\subseteq ((\mathit{dl}(K''_1) \setminus \mathit{dl}(K_1)) \cup \mathit{dl}(S)) \setminus \mathit{dl}(C') \ \cup \ \overline{B''} \setminus \overline{B} &
\end{aligned}$$

To show 2, we let $W_1 = W_2$. Therefore we must verify that

$$\mathsf{match}\ B^W\ \mathsf{with}\ \big|^{i\in 2..n}(B''_i \Rightarrow {B'_i}^{W'_i})\ \mathsf{end} \subseteq \mathsf{match}\ B^W\ \mathsf{with}\ \big|^{i\in I}(B''_i \Rightarrow {B'_i}^{W'_i})\ \mathsf{end}$$

By definition, this containment reduces as follows. Let $I_1 = 1..n$; $I_2 = 2..n$; $J_1 = \{i \in I_1 \mid \exists \ell \notin W, \overline{B''_i} = \{\ell\}\}$ and $J_2 = \{i \in I_2 \mid \exists \ell \notin W, \overline{B''_i} = \{\ell\}\}$. Then

$$\begin{array}{ll}
W_0 \cup \bigcup_{i \in J_2} W'_i \cup \bigcup_{i \in I_2 \setminus J_2} \overline{B'_i} & \underbrace{\cup \left(\overline{B_0} \setminus \bigcup_{i \in J_2} \overline{B'_i} \right)}_{\text{if } \exists i \in I_2, B''_i = \emptyset \wedge B''_i \neq \emptyset} \\
\subseteq & \\
W_0 \cup \bigcup_{i \in J_1} W'_i \cup \bigcup_{i \in I_1 \setminus J_1} \overline{B'_i} & \underbrace{\cup \left(\overline{B_0} \setminus \bigcup_{i \in J_1} \overline{B'_i} \right)}_{\text{if } \exists i \in I_1, B''_i = \emptyset \wedge B''_i \neq \emptyset}
\end{array}$$

The containment is evident as regards the three leftmost components. Let us assume that $B''_k = \emptyset \wedge B''_k \neq \emptyset$ (13). Since $\mathit{dl}(K''_1) \not\subseteq \mathit{dl}(M)$, $\mathit{dl}(K''_1)$ is not empty and neither is $\overline{B''_1}$, then k cannot be 1. Therefore the condition (13) holds for the two sides of the above containment. A difference could be only if $1 \in J_1$. However, in this case $\overline{B_0} \cap \overline{B'_1}$ must be empty; otherwise, $\mathit{dl}(M)$ would be reduced $\mathit{dl}(K'_1)$ and rule Filter-Next would not apply.

Case Filter-End: Let us assume

$$\begin{array}{cr}
M \triangleright P \text{ with } 0 \longmapsto M \triangleright P & (1) \\
A \vdash M \triangleright P :: \zeta(\rho) B^{W,V} & (2) \\
A \vdash 0 :: \emptyset^{V'} & \\
\mathit{dl}(0) \cap \overline{B} = \emptyset &
\end{array}$$

We conclude from (2) by taking $V'' = V$ and $\emptyset$ for L. We remark that $W =$ match B^W with $\emptyset$ end.

Case Filter-Abstract: Let us assume

$$\begin{array}{c}
L \text{ with } S \longmapsto L \\
A \vdash L :: \zeta(\rho) B^{W,V} \\
A \vdash S :: (\big|^{i \in I} (B''_i \Rightarrow (B'_i)^{W_i})^{V'} \\
\mathit{dl}(S) \cap \overline{B} = \emptyset
\end{array} \qquad (1)$$

A derivation of (1) must end with an instance of rule Abstract, hence $\overline{B} = V = L$. Observe that $A \vdash L : \overline{B}^{\emptyset,V}$. The conclusion $A \vdash L : \zeta(\rho)(B \oplus (B' \setminus L'))^{\emptyset,V''}$ holds by taking $V'' = V$ and $L' = \overline{B}' \setminus L$.

Inductive case.

Case Filter-Or: Let us assume that:

$$\begin{array}{cr}
C_1 \text{ or } C_2 \text{ with } S \longmapsto C' & (1) \\
A \vdash C_1 \text{ or } C_2 :: \zeta(\rho) B^{W,V} & (2) \\
A \vdash S :: (\zeta(\rho) \mid^{i \in I} (B''_i \Rightarrow (B'_i)^{W'_i}))^{V'} & (3) \\
\mathit{dl}(S) \cap \overline{B} = \emptyset & (4)
\end{array}$$

A derivation of (2) must end with an instance DISJUNCTION, hence B is of the form $B_1 \oplus B_2$ and:

$$\begin{aligned} A \vdash C_1 :: \zeta(\rho)B_1^{W_1,V_1} && (5)\\ A \vdash C_2 :: \zeta(\rho)B_2^{W_2,V_2} && (6)\\ (V_1 \setminus I_2) \cup (V_2 \setminus I_1) \subseteq V && (7)\\ W_1 \cup W_2 \subseteq W && \end{aligned}$$

where $I_i = \overline{B_i} \setminus V_i$ for $i \in \{1,2\}$. The condition (4) implies that $dl(S) \cap \overline{B_i} = \emptyset$ for $i \in \{1,2\}$ (8). 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 $i \in \{1,2\}$ (9). By induction hypothesis applied to (9), (8), (5), (6) and (3) it follows that

$$\begin{aligned} A \vdash C'_j :: \zeta(\rho)(B_j \oplus (B' \setminus L_j))^{W^j,V'_j} && (10)\\ L_j \subseteq (dl(S) \setminus dl(C'_j)) \cup (\overline{B''} \setminus \overline{B_j}) && (11)\\ V'_j \subseteq V_j \cup V' && (12)\\ W^j \subseteq \mathsf{match}\ B^W\ \mathsf{with}\ \zeta(\rho) \mid^{i\in I}(B''_i \Rightarrow (B'_i)^{W'_i})\ \mathsf{end} && (13) \end{aligned}$$

for $j \in \{1,2\}$. By rule disjunction applied to the two cases of (10) we have:

$$A \vdash C'_1\ \mathsf{or}\ C'_2 :: \zeta(r)(B'_1 \oplus B'_2)^{W^3,V''} \qquad (14)$$

where

$$\begin{aligned} V'' &= (V'_1 \setminus I'_2) \cup (V'_2 \setminus I'_1)\\ I'_i &= \overline{B'_i} \setminus V'_i\\ B'_i &= B_i \oplus (B' \setminus L_i)\\ W^3 &= W^1 \cup W^2 \end{aligned}$$

On the one hand, remark that

$$\begin{aligned} &\mathsf{match}\ (B_1 \oplus B^2)^{W_1 \cup W_2}\ \mathsf{with}\ R\ \mathsf{end} =\\ &\qquad (\mathsf{match}\ (B_1)^{W_1}\ \mathsf{with}\ R\ \mathsf{end}) \bigcup (\mathsf{match}\ (B^2)^{W_2}\ \mathsf{with}\ R\ \mathsf{end}) \end{aligned}$$

Hence, $W^3 \subseteq \mathsf{match}\ B^W\ \mathsf{with} \Big|^{i\in I}(B''_i \Rightarrow (B'_i)^{W'_i})\ \mathsf{end}$.

On the other hand, remark that

$$\begin{aligned} V'' &= V'_1 \setminus (\overline{B'_2} \setminus V'_2)\ \cup\ V'_2 \setminus (\overline{B'_1} \setminus V'_1)\\ &\subseteq V'_1 \setminus (\overline{B_2} \setminus V'_2)\ \cup\ V'_2 \setminus (\overline{B_1} \setminus V'_1)\\ &= (V'_1 \setminus \overline{B_2}) \cup (V'_2 \setminus \overline{B_1})\ \cup\ (V'_1 \cap V'_2 \cap B_1 \cap B_2)\\ &\subseteq V'\ \cup\ V_1 \setminus \overline{B_2}\ \cup\ V_2 \setminus \overline{B_1}\ \cup\ V_1 \cap V_2 \cap B_1 \cap B_2\\ &= V'\ \cup\ V_1 \setminus (\overline{B_2} \setminus V_2) \cup V_2 \setminus (\overline{B_1} \setminus V_1)\\ &\subseteq V' \cup V && (15) \end{aligned}$$

Taking $L_1 \cap L_2$ for L, we have

$$B'_1 \oplus B'_2 = (B_1 \oplus B_2) \oplus (B' \setminus L) = B_1 \oplus (B' \setminus L_1)\ \oplus\ B_2 \oplus (B' \setminus L_2)$$

Moreover,

$$\begin{aligned} L_1 \cap L_2 &\subseteq ((dl(S) \setminus dl(C_1)) \cup (\overline{B'} \setminus \overline{B_1})) \cap\\ &\quad\ ((dl(S) \setminus dl(C_2)) \cup (\overline{B'} \setminus \overline{B_2})) && \text{by (11)}\\ &\subseteq (dl(S) \setminus dl(C_1)) \cap (dl(S) \setminus dl(C_2)) \cup\\ &\quad\ (\overline{B'} \setminus \overline{B_1}) \cap (\overline{B'} \setminus \overline{B_2}) && \text{by distributivity}\\ &\subseteq (dl(S) \setminus (dl(C_1) \cup dl(C_2)))\ \cup\\ &\quad\ (\overline{B'} \setminus (\overline{B_1} \cup \overline{B_2}))\\ &= (dl(S) \setminus dl(C)) \cup (\overline{B'} \setminus \overline{B}) && (16) \end{aligned}$$

The properties (16), (15) and (14) complete the proof. □

Theorem 5 (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 $A \vdash P'$.

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, whose premise has to be

$$A_x + z : [\rho], z.(B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V}$$

By the side condition of Self, x is not free in C. Therefore, by lemma 2, we may take $A \setminus x$ instead of A_x. Then, by lemma 6, we have:

$$(A \setminus x)\{x/z\} + x : [\rho] + x.(B \upharpoonright \mathcal{F}) \vdash C\{x/z\} :: \zeta(\rho)B^{W,V}$$

To conclude, observe that $(A \setminus x)\{x/z\} + x : [\rho] + x.(B \upharpoonright \mathcal{F})$ is equivalent to $A + x : [\rho], x.(B \upharpoonright \mathcal{F})$.

Case Or-Pat**:** Let $J_1 \ \mathsf{or}\ J_2 \rhd P \stackrel{x}{\longmapsto} J_1 \rhd P \ \mathsf{or}\ J_2 \rhd P$ and let $A_x \vdash J_1 \ \mathsf{or}\ J_2 \rhd P :: \zeta(\rho)B^{W,V}$ (1). The end of a derivation of this judgment must be

$$\textsc{Reaction}\ \dfrac{\begin{array}{c}\textsc{Alternative}\ \dfrac{\begin{array}{c} \mathit{fn}(J_1) = \mathit{fn}(J_2)\ (3)\\ A' \vdash J_1 :: B_1^{W_1} \qquad A' \vdash J_2 :: B_2^{W_2} \qquad W = \mathit{dom}(B)\end{array}}{A' \vdash J_1 \ \mathsf{or}\ J_2 :: B^{W}}\\ A_x + A' \vdash P \qquad \mathit{dom}(A') = \mathit{fn}(J_1 \ \mathsf{or}\ J_2)\ (2)\end{array}}{A_x \vdash J_1 \ \mathsf{or}\ J_2 \rhd P :: \zeta(\rho)B^{W,V}}$$

where B is $B_1 \oplus B_2$. From (3) and (2), we deduce $\mathit{dom}(A') = \mathit{fn}(J_1) = \mathit{fn}(J_2)$. Hence, we have:

$$\textsc{Disjunction}\ \dfrac{\textsc{Reaction}\ \dfrac{\begin{array}{c} A' \vdash J_k :: B_k^{W_k}\\ A_x + A' \vdash P \qquad \mathit{dom}(A') = \mathit{fn}(J_k)\end{array}}{A_x \vdash J_k \rhd P :: \zeta(\rho)B_k^{W_k,V}}\ k = 1, 2}{A_x \vdash J_1 \rhd P \ \mathsf{or}\ J_2 \rhd P :: \zeta(\rho)B^{W,V}}$$

Case Abstract-Cut**:** Let $C_0\ \mathsf{or}\ L \xrightarrow{x} C_0\ \mathsf{or}\ L'$ and $A_x \vdash C_0\ \mathsf{or}\ L :: \zeta(\rho)B^V$. The end of a derivation of this judgment must be:

$$\textsc{Disjunction}\ \dfrac{\begin{array}{c} \textsc{Abstract}\ \dfrac{dom(B_2) = L}{A_x \vdash L :: \zeta(\rho)W_2, {B_2}^{W_L, L}} \\ A_x \vdash C_0 :: \zeta(\rho)B_0^{W_0,V_0} \qquad I = dom(B_0) \setminus V_0 \\ V_0 \cup (L \setminus I) \subseteq V\ (2) \qquad W_L \cup W_0 \subseteq W\ (1) \end{array}}{A_x \vdash C_0\ \mathsf{or}\ L :: \zeta(\rho)(B_0 \oplus B_2)^{W,V}}$$

Thus B is $B_0 \oplus B_2$ and $B_2 = B \upharpoonright L$. From rule Abstract-Cut, we know that $L \setminus dl(C_0) \subseteq L' \subseteq L$ (3).

We can easily show, by induction on the structure of C_0, that $A \vdash C_0 :: \zeta(\rho){B_0}^{W_0,V_0'}$ where V_0' is $V_0 \setminus dl(C_0)$. Let I' be $dom(B_0) \setminus V_0'$, *i.e.* $I \cup dl(C_0)$, since $dl(C_0) \subseteq dom(B_0)$ and $dom(B_0) \setminus V_0 = I$. Then

$$\begin{aligned} V_0' \cup (L' \setminus I') &= V_0 \setminus dl(C_0)\ \cup\ L' \setminus (I \cup dl(C_0)) && \text{by (3)} \\ &\subseteq V_0\ \cup\ (L \setminus I) && \text{by (2)} \\ &\subseteq V (4) \end{aligned}$$

Moreover, since $L \setminus L' \subseteq dl(C_0) \subseteq dom(B_0)$, we have $dom(B_0) \cup L' = dom(B)$. Finally, we may derive:

$$\textsc{Disjunction}\ \dfrac{\begin{array}{c} \textsc{Abstract}\ \dfrac{dom(B \upharpoonright L') = L'}{A_x \vdash L_2' :: \zeta(\rho)(B \upharpoonright L')^{\emptyset,L'}} \\ Ax \vdash C_0 :: \zeta(\rho){B_0}^{W_0,V_0'} \qquad (4) \qquad (1) \end{array}}{A_x \vdash C_0\ \mathsf{or}\ L' :: \zeta(\rho)B^{W,V}}$$

Case Class-Abstract**:** Let $C\ \mathsf{or}\ \emptyset \xrightarrow{x} C$ and $A_x \vdash C\ \mathsf{or}\ \emptyset :: \zeta(\rho)B^{W,V}$. The end of a derivation of this judgment must be:

$$\textsc{Disjunction}\ \dfrac{\begin{array}{c} A_x \vdash \emptyset :: \zeta(\rho)\emptyset^{\emptyset} \\ A_x \vdash C :: \zeta(\rho)B^{W_0,V_0} \qquad V_0 \subseteq V \qquad W_0 \subseteq W \end{array}}{A_x \vdash C\ \mathsf{or}\ \emptyset :: \zeta(\rho)B^{W,V}}$$

Then, by lemma 5 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 $A \vdash \mathsf{class}\ c = \mathsf{self}(z)\,C\ \mathsf{in}\ P$ (1) and $\mathsf{class}\ c = \mathsf{self}(z)\,C\ \mathsf{in}\ P \longmapsto P\{\mathsf{self}(z)\,C/_x\}$. The final part of the derivation of (1) must have the form

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

By lemma 6 applied to (3) and (2), we derive $A \vdash P\{\mathsf{self}(z)\,C/_c\}$.

Inductive cases for classes.

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

Case Match**:** Let us assume that $A \vdash \mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end} : \zeta(\rho){B_0}^{W_0,V_0}$ (1) and $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end} \longrightarrow C'$ (2). We must prove that $A \vdash C' : \zeta(\rho){B_0}^{W_0,V_0}$ (3)

A derivation of (1) must end with an instance of the rule Refinement, hence B_0 has the form $B \oplus B'$ and

$$A \vdash C :: \zeta(\rho)B^{W,V} \tag{4}$$
$$A \vdash S :: \zeta(\rho) \mid^{i \in I} \rho(B''_i \Rightarrow (B'_i)^{W'_i})^{V'} \tag{5}$$
$$B'' \subseteq B \tag{6}$$
$$dl(S) \cap dom(B) = \emptyset \tag{7}$$
$$V \cup V' \subseteq V_0 \tag{8}$$

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

$$C\ \mathsf{with}\ S \longrightarrow C' \tag{9}$$
$$dl(S) \subseteq dl(C') \tag{10}$$
$$\mathsf{match}\ B^W\ \mathsf{with}\ \big|^{i \in I}(B''_i \Rightarrow (B'_i)^{W'_i})\ \mathsf{end} \subseteq W_0 \tag{11}$$

The lemma 9 applied to the (4), (5), (7), and (9) implies that

$$A \vdash C' :: \zeta(\rho)(B \oplus (B' \setminus L))^{W'',V''} \tag{12}$$
$$L \subseteq (dl(S) \setminus dl(C')) \cup (dom(B'') \setminus dom(B)) \tag{13}$$
$$V'' \subseteq V \cup V' \tag{14}$$
$$W'' \subseteq \mathsf{match}\ B^W\ \mathsf{with}\ \big|^{i \in I}(B''_i \Rightarrow (B'_i)^{W'_i})\ \mathsf{end} \tag{15}$$

The properties (6) and (10) combined with (13) imply that L is empty, hence $A \vdash C' :: \zeta(\rho)(B \oplus B')^{W'',V''}$ by (12) and the conclusion (3) follows by lemma 5 applied to (14) and (8) and by the combination of (11) and (15).

Inductive cases for processes.

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

A derivation of (1) has the shape

$$\textsc{Self-Binding}\ \dfrac{A + z : [\rho], z : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,\emptyset}\ (3)}{A \vdash \mathsf{self}(z)\, C :: \zeta(\rho)B^{W,\emptyset}}$$
$$\rho = B \upharpoonright \mathcal{M} \qquad X = \mathsf{Gen}\ (B, A, W)$$
$$\textsc{Object}\ \dfrac{A + x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F}) \vdash P \qquad A + x : \forall X.[\rho] \vdash Q}{A \vdash \mathsf{obj}\ x = \mathsf{self}(z)\, C\ \mathsf{init}\ P\ \mathsf{in}\ Q}$$

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

Note that if rule Process-Context allowed reductions under guards, then the proof would simply include another subcase for Process-Context: Assuming $A \vdash E[P]$ and $E[P] \longmapsto E[P']$ under the assumption that $P \longmapsto P'$, the judgment $A \vdash E[P']$ could easily be proved by induction on the structure of $E[\cdot]$.

A.5 Proofs of progress for the reduction of classes

Theorem 6 (Process-Progress) *A process that is well-typed in the empty environment and that cannot be reduced either is in the core object calculus or contains an erroneous refinement (a selection clause* $\mathsf{match}\ C\ \mathsf{with}\ S\ \mathsf{end}$ *such that* $C\ \mathsf{with}\ S \longmapsto C'$ *and* $\mathit{dl}(S) \not\subseteq C'$*).*

We say that P and C are *terminal* if either they are in the core calculus or they contain an erroneous refinement.

We show that for any context A that does not bind classes, P and C are terminal whenever $A \vdash P$ and $A \vdash C :: \zeta(\rho)B^{W,\emptyset}$, respectively. We reason by induction on the structure of P and C.

Processes.

Case P is $\mathsf{0}$ or $x.M$: Then P is in the core calculus, hence terminal.

Case P is $P_1 \,\&\, P_2$: Then, P_1 and P_2 are well-typed and irreducible in some environment that does not bind class variables. Therefore, by induction, both processes are terminal, and so is P.

Case P is $\mathsf{class}\ c = \mathsf{self}(z)\, C\ \mathsf{in}\ P'$: This process can be reduced.

Case P is $\mathsf{obj}{=}\ \mathsf{self}(z)\, C\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2$: A derivation of $A \vdash P$ must end with

$$\textsc{Self-Binding}\ \frac{A + x : [\rho],\ x.(B \restriction \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V}\,(4)}{A \vdash \mathsf{self}(x)\, C :: \zeta(\rho)B^{W,V}}$$

$$\textsc{Object}\ \frac{\rho = B \restriction \mathcal{M} \qquad X = \mathsf{Gen}\ (B, A, W) \atop A + x : \forall X.[\rho], x : \forall X.(B \restriction \mathcal{F}) \vdash P_1 \qquad A + x : \forall X.[\rho] \vdash P_2}{A \vdash \mathsf{obj}\ x = \mathsf{self}(z)\, C\ \mathsf{init}\ P_1\ \mathsf{in}\ P_2}$$

By induction P_1, P_2 and C are terminal, and so is P.

Definitions.

Case C is c: This is vacuous because A does not bind class variables.

Case C is L: This is also vacuous because the class type $\zeta(\rho)B^{\emptyset}$ has no virtual labels.

Case C is $J \triangleright P$: Then by rule Rule, $A + A' \vdash P$, for some environment A' that does not bind classes. Therefore P is terminal, and so is C.

Case C is $C_1\ \mathsf{or}\ C_2$: Then $A \vdash C_k :: \zeta(\rho)B_k^{\emptyset}$ and, by induction, C_k is terminal, for k in $\{1, 2\}$, and so is C.

Case C is $\mathsf{self}(x)\, C'$: Then $A + x : [\rho], x : (B \restriction \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,\emptyset}$ and, by induction, C' is terminal, and so is C.

Case C is $\mathsf{match}\ C'\ \mathsf{with}\ S\ \mathsf{end}$: A derivation of $A \vdash C :: \zeta(\rho)B^{W,V}$ must end with

$$\textsc{Refinement}\quad \frac{\begin{array}{c} A \vdash C' :: \zeta(\rho)B_1^{V_1} \qquad A \vdash S :: \zeta(\rho)(B_2 \Rightarrow B_2')^{V_2} \\ \mathit{dom}(B_1) \cap \mathit{dl}(S) = \emptyset\ (5) \qquad I = \mathit{dom}(B_2) \setminus V_2 \qquad (V_1 \setminus I) \cup V_2 \subseteq V \end{array}}{A \vdash \mathsf{match}\ C'\ \mathsf{with}\ S\ \mathsf{end} :: \zeta(\rho)(B_1 \oplus B_2')^{W,V}}$$

By induction, C' is terminal. If C' contains an erroneous refinement, then C also does and is terminal. Otherwise, C' is in the core calculus and there exists a class C'' such that $C'\ \mathsf{with}\ S \longmapsto C''$. Since C cannot be reduced, it must be the case that $\mathit{dl}(S) \not\subseteq \mathit{dl}(C'')$. Then, C is an erroneous refinement and thus it is terminal. $\square$