



Information Hiding, Inheritance and Concurrency

Qin Ma, Luc Maranget

► To cite this version:

Qin Ma, Luc Maranget. Information Hiding, Inheritance and Concurrency. [Research Report] RR-5631, INRIA. 2005, pp.74. inria-00070376

HAL Id: inria-00070376

<https://inria.hal.science/inria-00070376>

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Information Hiding, Inheritance and Concurrency

Qin Ma Luc Maranget

N° 5631

18th July 2005

THÈME 1



*apport
de recherche*

Information Hiding, Inheritance and Concurrency

Qin Ma Luc Maranget

Thème 1 — Réseaux et systèmes
Projet Moscova

Rapport de recherche n° 5631 — 18th July 2005 — 74 pages

Abstract: We aim to provide information hiding support in concurrent object-oriented programming languages. We study the issue of information hiding at the object level and class level, in the context of an object-oriented extension of the Join calculus — a process calculus in the tradition of the Pi-calculus. At the object level, we improve a privacy mechanism proposed in prior work by defining a simpler chemical semantics for privacy control. At the class level, we propose a hiding mechanism by designing a new operation on classes. We define its formal semantics in terms of alpha-converting hidden names to fresh names, and its typing in terms of eliminating hidden names from class types. We study the standard soundness property of the type system, as well as specific properties concerning hiding. Some, if not most, of our choices in designing our system are motivated by implementation. As an evidence of practical significance, we implement our model in a prototyping system. From that experience we draw guidelines for a full-scale implementation.

Key-words: language design, information hiding, access control, type abstraction, concurrency, object-oriented programming type system, join calculus

Visibilité des noms, héritage et concurrence

Résumé : Nous considérons la question du contrôle de la visibilité des noms dans les langages concurrents et orientés-objet. Nous procédons dans le cadre d’une extension objet du Join calcul — un calcul de processus dans la tradition du Pi-calcul, et traitons la question à la fois au niveau de l’objet et de la classe. Pour ce qui est des objets, nous améliorons les travaux précédents en proposant une sémantique chimique plus simple et plus réaliste pour le contrôle de la visibilité à l’exécution. Pour ce qui est des classes, nous ajoutons un nouvel opérateur de masquage (*hiding*) au calcul des classes, dans le but de rendre inaccessible partie des classes parentes lors de l’héritage. Nous définissons la sémantique du nouvel opérateur par le renommage des noms masqués en des noms frais, tandis que son typage entraîne la disparition des noms masqués des types des classes. Nous formulons et prouvons les propriétés usuelles de sûreté du système de types, ainsi que quelques propriétés supplémentaires liées au contrôle de visibilité. Certains choix de conceptions de notre modèle s’inspirent du souci d’implémentation, réalisée pour le moment sous la forme d’un prototype. De ce prototype nous tirons les grandes lignes d’une intégration en vraie grandeur de nos classes dans le langage JoCaml.

Mots-clés : conception de langages, contrôle de la visibilité des noms, abstraction dans les types, concurrence, programmation orientée-objet, join calcul

1 Introduction

Object-oriented programming concepts are often claimed to provide a practical view of concurrent systems. Basically, objects exchanging messages while managing their internal states in a private fashion seem a plausible model of computers or agents interacting over a network. To give solid semantical foundations to this idea, numerous fundamental studies [25, 3, 17, 34, 28, 26, 27, 37, 33, 19, 7, 24] propose calculi that combine objects and concurrency.

But there exists another connection between object-oriented design and concurrent systems. Concurrent programs are highly complex and extremely difficult to get right, while object-oriented design offers strong support for modular and incremental design, usually by the means of classes. Unfortunately, the idea of taming the complexity of concurrent systems by object-oriented design is hindered by *inheritance anomalies* [23], *i.e.*, the inability to program the objects internal synchronization behaviors by class inheritance. From that perspective, Fournet *et al.* make a significant progress with their work [15]. Fournet *et al.* supplement the the Join calculus [11, 12] with classes and objects, so as to actually propose a solution to all the inheritance anomalies of [23].

However, as we describe in our previous work [22], Fournet *et al.*'s model falls short in forming the basis of a practical implementation. Briefly, Fournet *et al.*'s class calculus does support the incremental programming of synchronization policies, but its type system is somehow counter-intuitive and significantly restricts the classes that can be produced by inheritance, when compared to those that can be written directly. In [22], we improve Fournet *et al.*'s model by designing a new type system, of which the main contribution is the inclusion of complete synchronization behavior in class types. Our more expressive class types actually support synchronization inheritance better. But, being quite verbose, they make even more visible another shortcoming of the original design, namely the lack of abstraction in class types, which in our design express how classes are seen from the outside world.

Generally, information hiding allows the separation between a restricted interface and a full implementation, and imposes limitations on how a name can be referenced. In a nutshell, this principle brings two advantages: first, irrelevant details are gotten rid of, favoring code comprehension, maintenance, and reuse; second, critical details are protected, favoring system robustness and invariant guarantee. However, few formalisms are reported in the literature that either address the implementation of hiding in concurrent settings, or investigate its impact on synchronization. In fact, Fournet *et al.* have supplied a rigid privacy policy (**private** annotations on names) to their model in [15] for this purpose. However, their design suffers from complexity, and more seriously, only supports limited information hiding. Specifically, we classify users of a class into two categories: *object users* who create objects from the class; and *inheritance users* who derive new class definitions by inheriting the class, and the privacy policy applies solely to object users while always leaves full access to inheritance users.

In this paper, we follow Fournet *et al.*'s model for concurrent object-oriented programming, and we study information hiding in it. Our main contribution is twofold. First, we define a simpler privacy control mechanism at the object level, which improves the one of [15]. Second, we introduce a new explicit hiding operation in the class calculus. This amounts to significant changes in both the semantics and the typing of class operations. As rewards, the benefit is also twofold: first, we gain abstraction in class types while still preserving safety; second and more importantly, we reach a more powerful concurrent object-oriented calculus, with flexible visibility control.

The rest of this paper is organized as follows. In Section 2, we present the ObJoin calculus — a variant of the Join calculus with concurrent objects proposed by Fournet *et al.* in [15], and meanwhile feature a simpler chemical semantics with privacy. In Section 3, we design a class language on top of ObJoin, called OJoin_H. This class language extends the one of [15] with the capability of hiding. The semantics of OJoin_H is defined in Section 4 in terms of evaluating classes into plain object definitions. A ML-style type system is provided for OJoin_H in Section 5. We state the standard soundness property of the type system in Section 6, as well as some other specific properties concerning hiding. Proof details are organized as an appendix at the end of this paper. Finally, a prototyping system is coded for OJoin_H in OCaml for experimental purpose. Discussions stimulated by the prototyping work can be found in Section 7, mainly concerning how to incorporate the improved model into the current JoCaml system [13].

2 The ObJoin calculus and privacy policy

2.1 A one-place buffer object

Objects arise naturally from the Join calculus when join-definitions are named and lifted to be the values of the calculus. For instance, a one-place buffer object is defined as follows:

```
obj buffer =
  put(n,r) & Empty() ▷ r.reply() & this.Some(n)
or get(r) & Some(n) ▷ r.reply(n) & this.Empty()
init this.Empty()
```

Just as join-definitions, objects are collections of channel definitions. However, values are now the collections of channels themselves, while channels become labels attached to objects. That is, channel names now behave as method names in object-oriented languages. As an important consequence, channel names are no longer lexically scoped. The basic operation of the calculus remains asynchronous message sending, but it is now expressed with object-oriented dot notation. For instance, process **buffer.put(n,r)** sends a **put(n,r)** message to the object **buffer**, where **put** is the destination channel and (**n,r**) is the content. Notice that the message possesses a **r** component, an object accepting messages on the **reply** label, so as to acknowledge **put** success.

Labels defined in one object are organized by several reaction rules that specify how messages sent on these labels will be synchronized and processed. For the one-place buffer example above, four labels are defined and arranged in two reaction rules. Following the join-pattern synchronization mechanism, object **buffer** will behave as follows:

- If the buffer is empty (*i.e.* an **Empty()** message is pending), and a put attempt is made (by sending a **put(n,r)** message), then the buffer will react by shifting itself into the **Some(n)** state.
- Symmetrically, the value **n** stored in the buffer (*i.e.* a **Some(n)** message is pending) can be retrieved by **get(r)**. The buffer sends back the value on label **reply** of the continuation object **r** and, concurrently, returns back to the empty state.
- Any **put** requests sent to a full buffer (or **get** request to an empty buffer) will be delayed until the object is changed into the complementary state by other messages.

- Finally, the (optional) **init** process initializes the object as an empty buffer.

A similar example with more implementation-oriented description is also used in [2].

Note that we use the keyword **this** for recursive “self” references. (Fournet *et al.* use object names themselves.) This is one of the modifications we introduce. In general, when an object is created, it can be referenced in two kinds of positions: either recursively from its guarded processes and the **init** process (*internal positions*); or otherwise (*external positions*). The usage of **this** allow us to distinguish the two kinds of positions syntactically, in the sense that in internal positions, references are always through **this**, while in external ones, references are always through object names.

2.2 Privacy policy

The usage of the two labels **Empty** and **Some** in the **buffer** object above is special. More precisely, the state of the one-place buffer, either empty or full, is encoded as a message pending on either **Empty** or **Some**. Moreover, to keep the buffer consistent, such messages should never appear simultaneously. We define **buffer** carefully to respect this. Object **buffer** is initially empty and switches alternatively between states full and empty according to the operations applied to it. However, this kind of guarantee is fragile: a single message sent on either **Empty** or **Some** from outside suffices to break the invariant. To avoid this, Fournet *et al.* adopt a rigid privacy policy in their model [15], to restrict the views of object users. Conventionally, labels starting with an uppercase letter (such as **Empty** and **Some**) are *private* and messages sent on those labels should always originate from internal positions. Other labels are unrestricted (such as **put** and **get**). They are *public* and work as ports offered by the object to the external world.

It is worth noticing that the privacy policy is enforced statically, while typing. Namely, typing environments bind object names to *object types* which list public labels only, while the types associated to **this** are complete and list all labels.

2.3 Formalization

2.3.1 Syntax

The formal syntax of the ObJoin calculus is defined in Figure 1. We assume two disjoint sets of identifiers: for variables (or object names) $x, y, z, o \in \mathcal{O}$, and for labels $l \in \mathcal{L}$. For privacy purpose, the set of labels $\mathcal{L}$ is further partitioned into two disjoint subsets, with $f \in \mathcal{F}$ for private labels and $m \in \mathcal{M}$ for public labels. In general, we write u for either a name from $\mathcal{O}$ or the keyword **this**, which refers to the self object.

Three syntactic categories are defined: processes P , join-definitions D , and join-patterns M . A process can be either a null process 0 , a (group of) message destined to the same object $x.M$, a parallel composition of processes $P_1 \ \& \ P_2$, or an object defining process **obj** $o = D \ \mathbf{init} \ P \ \mathbf{in} \ Q$. A join-definition D is a disjunction of reaction rules of form $M \triangleright P$ that associate a join-pattern M with a guarded process P . A join-pattern defines a set of synchronized ($\&$) labels with formal arguments, and the corresponding guarded process specifies the behavior when messages on these labels are consumed simultaneously. Note that although we write $l(\tilde{u})$ for message patterns in Figure 1, we in fact additionally state that **this** can only appear in message sending but never as formal arguments of join-patterns.

$P, Q ::=$	Processes
0	null process
$x.M$	message sending
this . M	recursive message sending
$P_1 \& P_2$	parallel composition
obj $o = D$ init P in Q	object definition
$D ::=$	Join-definitions
$M \triangleright P$	reaction rule
D_1 or D_2	disjunction of definitions
$M ::=$	Join-patterns
$l(\tilde{u})$	message pattern
$M_1 \& M_2$	synchronization

Figure 1: Syntax of the ObJoin calculus

STR-NUL	STR-PAR
$\vdash \phi \# 0 \equiv \vdash$	$\vdash \phi \# (P \& Q) \equiv \vdash \phi \# P, \phi \# Q$
STR-JOIN	
$\vdash \phi \# u.(M_1 \& M_2) \equiv \vdash \phi \# u.M_1, \phi \# u.M_2$	
STR-OBJ	
$\frac{x \notin \text{fv}[D] \cup \text{fv}[P]}{\vdash \phi \# \mathbf{obj} \ x = D \ \mathbf{init} \ P \ \mathbf{in} \ Q \equiv x.(D \ \mathbf{or} \ \mathbf{init}() \triangleright P) \vdash \Phi \# x.\mathbf{init}(), \phi \# Q}$	
THIS-COMM	OBJ-COMM
$\vdash \phi \# \mathbf{this}.l(\tilde{u}) \longrightarrow \vdash \Phi \# \phi(\mathbf{this}).l(\phi(\tilde{u}))$	$\vdash \phi \# x.m(\tilde{u}) \longrightarrow \vdash \Phi \# x.m(\phi(\tilde{u}))$
REACT	
$x.(... M \triangleright P ...) \vdash \Phi \# x.M\sigma \longrightarrow x.(... M \triangleright P ...) \vdash (\mathbf{this} \mapsto x) \# P\sigma$	
CHEMISTRY	CHEMISTRY-OBJ
$\frac{\mathcal{D}_0 \vdash \mathcal{P}_1 \Longrightarrow \mathcal{D}_0 \vdash \mathcal{P}_2}{\mathcal{D}, \mathcal{D}_0 \vdash \mathcal{P}_1, \mathcal{P} \Longrightarrow \mathcal{D}, \mathcal{D}_0 \vdash \mathcal{P}_2, \mathcal{P}}$	$\frac{\vdash \phi \# P \equiv x.D \vdash \mathcal{P}' \quad x \notin \text{fv}[\mathcal{D}] \cup \text{fv}[\mathcal{P}]}{\mathcal{D} \vdash \phi \# P, \mathcal{P} \equiv \mathcal{D}, x.D \vdash \mathcal{P}', \mathcal{P}}$

Figure 2: RCHAM of ObJoin with privacy

Join-patterns bind formal arguments in corresponding guarded processes. In addition, the object definition **obj** $o = D$ **init** P **in** Q binds the object name o in Q and the keyword **this** to o in P and in the guarded processes in D . We denote the set of free variables as $\text{fv}[\cdot]$, the set of formal arguments (or received variables) as $\text{rv}[\cdot]$, and the set of defined labels as $\text{dl}[\cdot]$. See Figure 4 for the formal definitions of these sets in the context of $\text{OJoin}_{\mathcal{H}}$ — the richer model with classes. As in Join, we require every join-pattern M to be linear in the

strong sense that all the label defined ($\text{dl}[M]$) and all the formal arguments bound ($\text{rv}[M]$) are pairwise distinct. The definitions of Figure 4 specify the linearity of join-patterns by using the disjoint union operator “ $\uplus$ ”. Non linear join-pattern would significantly improve expressiveness, by providing a way to check equality of channels (through formal arguments). Accordingly, program equivalence and implementation would become more difficult.

2.3.2 Chemical semantics with privacy

We design the operational semantics of ObJoin as a *reflexive chemical abstract machine* [12], refined with a notion of privacy. Such a notion is critical to stating the properties of type systems, namely, to demonstrating that the privacy policy enforced by static typing is more restrictive than the one checked by the implementation.

A *chemical solution* $\mathcal{D} \Vdash \mathcal{P}$ denotes the state of the machine, where $\mathcal{D}$ is a set of active objects (named join-definitions), ranged over by $x.D$, and $\mathcal{P}$ is a multiset of running processes prefixed with privacy annotations, ranged over by $\phi \# P$. A *privacy annotation* ϕ is a function that maps **this** to either an object name or **this**. Such a function can also be undefined, written (**this** $\mapsto \perp$). We call this last annotation empty, and we usually omit it. That way, the ObJoin process P corresponds to a chemical solution $\Vdash P$.

Formal chemical rewriting rules appear in Figure 2. They are of two kinds: structural rules $\equiv$ represent the syntactical rearrangement of the terms, and reduction rules $\longrightarrow$ represent the computation steps. Privacy annotations are used in rules to limit communications on private labels. (See discussion below on how communications are implemented.) Generally speaking, privacy annotations serve two purposes: first, to pass around the binding of **this** (cf. rule REACT); second, to indicate the success of privacy check by the trivial function $\Phi = (\text{this} \mapsto \text{this})$.

Rules STR-NUL and STR-PAR make parallel composition of process associative and commutative, with unit 0. Rule STR-JOIN organizes messages sent to the same object. And rule STR-OBJ activates an object definition with a fresh name x . Note that we use the side condition $x \notin \text{fv}[D] \cup \text{fv}[P]$ to guarantee freshness. During activation, we treat **init** process P as a special guarded process by appending it into the object body D in terms of **init**() $\triangleright P$, and explicitly sending a message **init**() to launch P . To ensure that initializations run just once, we reserve the definition of label **init** to semantics only, *i.e.*, **init** not accessible by programmers. This somehow indirect way to initialize objects reflects implementation, where initializers are compiled code, as guarded processes are. From the point of view of privacy control, the message on channel **init** should always be authorized, hence is prefixed with Φ . By contrast, the process Q inherits the original privacy annotation ϕ .

We implement communications in two steps: first privacy check then message consumption. Rules THIS-COMM and OBJ-COMM are here to check privacy. We substitute real object name for **this** in messages according to the original privacy annotation ϕ , and prefix any messages sent through **this** or any public messages sent through named objects with a new annotation Φ , indicating authorization. Rule REACT delivers authorized messages from senders to receivers (the success annotation Φ is checked), and triggers a copy of the appropriate guarded process $P\sigma$, whose formal arguments are replaced by message contents, and whose privacy annotation binds **this** to the name of the receiver object x . The latter reflects implementation, where a binding for the current object is created.

By convention, chemical rules mention only the components that take part in the rewriting, while in fact apply to any solutions that contain those. This is made explicit by the two context

rules: CHEMISTRY-OBJ (for rule STR-OBJ) and CHEMISTRY (for other rules). In CHEMISTRY, we write $\Rightarrow$ for either $\equiv$ or $\longrightarrow$. In CHEMISTRY-OBJ, the side condition $x \notin \text{fv}[\mathcal{D}] \cup \text{fv}[\mathcal{P}]$ precludes the fresh name x of the newly activated object from being captured. (The sets $\text{fv}[\mathcal{D}]$ and $\text{fv}[\mathcal{P}]$ are defined in a member-wise manner Figure 4.)

Fournet *et al.* also supply a privacy control mechanism in [15]. Compared to their design, our design is much simpler and mainly syntactical thanks to the straight distinction of internal and external positions through keyword **this**. By contrast, Fournet *et al.* use object names in all references to objects, including recursive (or “internal”) ones. To express privacy, they equip processes P in the chemical soup with sophisticated privacy annotations, which basically are lists of active object names. Thereby, they express that P is in internal position w.r.t. the objects in the list. When process P is some private message sending $x.f(\dots)$, a check that x occurs in the privacy annotation is performed by a dedicated chemical rule. Overall, the setting of Fournet *et al.* is more tolerant than ours, as illustrated by the following example:

obj $x = \text{Ch}() \triangleright P$ **or** $m(o) \triangleright o.\text{Ch}()$ **in** $x.m(x)$

Following Fournet *et al.*, message sending $o.\text{Ch}()$ is legal when o is object x itself, which happens to be the case above. By contrast, if o is bound to some object y other than x that possesses a private **Ch** label¹, message sending $o.\text{Ch}()$ is rejected at runtime because of a privacy violation. Our model cannot emulate such a situation, since all message sendings on private labels are performed through the special binding **this**. It is clear that the model of Fournet *et al.* is more expressive than ours, and one may find that it is legitimate for x to access its own private labels in a slightly indirect fashion. However, the typing system of [15] is unable to make a distinction between objects x and y . As a result, the above example is rejected at typing, leaving the additional expressiveness unexploited.

3 The OJoin_H calculus and hiding mechanism

3.1 A one-place buffer class

Classes act as templates for sets of objects with the same behavior. For instance, the following class defines one-place buffers:

```
class c_buffer =
  put(n,r) & Empty()  $\triangleright$  r.reply() & this.Some(n)
or get(r) & Some(n)  $\triangleright$  r.reply(n) & this.Empty()
init this.Empty()
```

And to instantiate an object from the class, we do:

obj **buffer** = **c_buffer**

The **init** process is an inseparable part of an object definition, hence is also lifted into the class definition, called *initializer* (analog to *constructors* or *makers* in other languages). Initializers will be triggered whenever objects are created from the class. Compared with the design of Fournet *et al.*, where initialization remains at the object level with each object defining its own independently, uniform initialization given at the class level enjoys the following advantages. First, our way supports more code reuse: objects of the same class share their initialization code, and during inheritance, the initializer of the derived class is implicitly composed in parallel with the ones of all the original classes. Moreover, initialization is also

¹Such a situation arises naturally when x and y are built from the same class

an internal position. Gathering it with other internal positions, namely guarded processes, is more practical than having internal positions scattered at both class and object levels.

3.2 Class operations

In order to derive new definitions from existing ones, two operations on classes are provided in [15]: disjunction for accumulation or reaction rules, and selective refinement for rewriting.

For example, a possible modification of the previous one-place buffer may be to log buffer content on the terminal for debugging purpose. We implement in terms of the *disjunction* of class `c_buffer` and two new reaction rules ²

```
class c_log_buffer =
  c_buffer
  or log() & Some(n) ▷ this.Some(n) & out.print_int(n)
  or log() & Empty() ▷ this.Empty() & out.print_string("Empty")
```

Class `c_log_buffer` inherits `c_buffer`, defines a new label `log`, and “overrides” the two inherited labels `Some` and `Empty`. However, in contrast to overriding in sequential settings, where new definition replaces old one, here, definitions cumulate, yielding competing behaviors for messages on that channel (*e.g.* label `Some` synchronized with both `log` and `get`). The order among disjunction components does not matter. Note that the initializer of class `c_buffer` is also implicitly inherited, meant to be composed in parallel with the one of class `c_log_buffer`. Since class `c_log_buffer` has no definition for its own initializer, the inherited one becomes its initializer.

Another interesting debugging requirement may be to log every `put` and `get` operation. We implement this in terms of the *selective refinement* of class `c_buffer` against two *refinement clauses*:

```
class c_log_buffer_bis =
  match c_buffer with
  | put(n,r) ⇒ put(n,r) ▷ out.print_int(n)
  | get(r) ⇒ get(r) ▷ out.print_string("get")
  end
```

The **match-with-end** construct can be understood by analogy with ML-style pattern matching. The *selective pattern* (to the left of $\Rightarrow$) of the first clause is `put(n,r)`. It matches any reaction rule whose join-pattern has the form $\dots \& \text{put}(\dots) \& \dots$, and replaces the sub-pattern `put(n,r)` by the refinement pattern (to the right of $\Rightarrow$). Here, because the refinement pattern is exactly the same as the selective pattern, the join-pattern remains the same during refinement. However, the guarded process becomes the parallel composition of the original one and the *refinement process* `out.print_int(n)`. Similarly, the second clause matches reaction rules containing the `get` label, and adds the printing process. The reaction rules of `c_buffer` are matched following the order of refinement clauses. They are either rewritten according to the first matching clause if it exists, or remain unchanged. As a consequence, `c_log_buffer` actually behaves as:

```
class c_log_buffer_bis =
  put(n) & Empty() ▷ r.reply() & this.Some(n) & out.print_int(n)
  or get(r) & Some(n) ▷ r.reply(n) & this.Empty() & out.print_string("get")
  init this.Empty()
```

²The terminal is implemented as an object named `out`, with labels `print_int`, `print_string` etc.

3.3 Inheritance and hiding

At the moment, all labels defined in a class are visible during inheritance, since our privacy policy only applies to message sendings to objects. However, this complete knowledge of class behavior may not be needed for building a given class by inheritance. For instance, the designer of the previous class `c_log_buffer_bis` needs no information on the private labels `Empty` and `Some`. Moreover, exposing full details during inheritance sometimes puts program safety at risk, and designers of parent classes may legitimately wish to restrict inheritance users freedom.

As an example, an inheritance user may attempt to extend the class `c_buffer` with a new channel `put2` for putting two elements:

```
class c_put2_buffer =
  c_buffer
  or put2(n,m,r) & Empty() ▷
    r.reply() & this.(Some(n) & Some(m))
```

Unfortunately, this naïve implementation breaks the invariant of a one-place buffer. More specifically, the `put2` attempt, once it succeeds, sends two messages on label `Some` in parallel. Semantically, this means turning a one-place buffer into an invalid state where two values are stored simultaneously.

In order to protect classes from (deliberate or accidental) integrity-violating inheritance, we introduce a new operation on classes to hide critical labels. We reach a more robust definition using hiding:

```
class c_hidden_buffer = c_buffer hide {Empty, Some}
```

The hiding clause `hide {Empty, Some}` hides the critical channels `Empty` and `Some`. They are now absent from the class type and become inaccessible during inheritance. As a result, the previous invariant-violating definition of channel `put2` will be rejected by a “name unbound” static error. Nevertheless, programmers still can supplement one-place buffers with a `put2` operation as follows:

```
class c_buffer_bis =
  c_buffer_hidden
  or put2(n,m,r) ▷
    class c_join =
      reply() & Next() ▷ r.reply()
      or reply() & Start() ▷ this.Next()
      init this.Start() in
      obj k = c_join in
      this.(put(n,k) & put(m,k))
```

In the code above, the reactions rules of the (inner) class `c_join` serve the purpose of consuming two acknowledgements from the one-place buffer before acknowledging the success of the `put2` operation to the appropriate object `r`. One may remark that the order in which values `n` and `m` are stored remains unspecified.

It is important to understand that hiding does not reduce to erasing the hidden labels from class types. Hiding also is an operation of the class semantics, whose design is governed by two concerns. On the one hand, once hidden, labels disappear; for instance, redefining a new label homonymous to a previously hidden label yields a totally new label, and selective patterns can no longer match the hidden label. On the other hand, hidden labels still exist

$P, Q ::=$	Processes
0	null process
$x.M$	message sending
this . M	recursive message sending
$P_1 \& P_2$	parallel composition
obj $o = C$ in P	object definition
class $c = C$ hide F in P	class binding with hiding
$C ::=$	Classes
c	class name
L	abstract class
$M \triangleright P$	reaction rule
C_1 or C_2	disjunction
match C with S end	selective refinement
C init P	initializer
$S ::=$	Refinement Sequences
$\emptyset$	empty sequence
$K_1 \Rightarrow K_2 \triangleright P \mid S$	refinement clause
$M ::=$	Join-Patterns
$l(\tilde{u})$	message pattern
$M_1 \& M_2$	synchronization
$K ::=$	Selection-Patterns
0	empty pattern
M	join-pattern

Figure 3: Syntax of the OJoin_H calculus

inside the class that underwent hiding; for instance, objects created by instantiating the class `c_buffer_hidden` must somehow possess labels to encode the state of a one-place buffer.

3.4 Syntax of OJoin_H

The OJoin_H calculus provides a class layer on top of ObJoin. It enriches the class language of [15] with a hiding mechanism. Equipped with this support, class designers thus can combine at will privacy policy and hiding mechanism, to achieve more precise and flexible visibility control of their classes. We use an additional set of identifiers for class names $c \in \mathcal{C}$. The grammar of the OJoin_H calculus is defined in Figure 3.

OJoin_H extends ObJoin with class definitions, thus objects are now created from classes: **obj** $o = C$ **in** P . Class definitions C are built from a full variety of constructs. Reaction rules are basic classes. We write $L \subseteq \mathcal{L}$ for a set of labels. Such L stands for abstract classes, whose labels are declared but not defined. Abstract labels are useful to force inheritance users to define certain labels, but they are also necessary to the semantics of selective refinement which may erase a given label from all the join-patterns of a class. Three operations are possible to manipulate classes: disjunction C_1 **or** C_2 to combine two definitions; selective refinement **match** C **with** S **end** to rewrite reaction rules; and C **init** P to add and initializer. We extend join-patterns M to selective-patterns K with empty patterns 0, which serve as the

For join-patterns :

$$\begin{array}{ll}
 \text{rv}[l(\tilde{u})] & \stackrel{\text{def}}{=} \{\tilde{u}\} \\
 \text{rv}[M_1 \& M_2] & \stackrel{\text{def}}{=} \text{rv}[M_1] \uplus \text{rv}[M_2] \\
 \text{rv}[0] & \stackrel{\text{def}}{=} \emptyset
 \end{array}
 \qquad
 \begin{array}{ll}
 \text{dl}[l(\tilde{u})] & \stackrel{\text{def}}{=} \{\tilde{u}\} \\
 \text{dl}[M_1 \& M_2] & \stackrel{\text{def}}{=} \text{dl}[M_1] \uplus \text{dl}[M_2] \\
 \text{dl}[0] & \stackrel{\text{def}}{=} \emptyset
 \end{array}$$

For processes :

$$\begin{array}{ll}
 \text{fv}[0] & \stackrel{\text{def}}{=} \emptyset \\
 \text{fv}[x.M] & \stackrel{\text{def}}{=} \{x\} \cup \text{rv}[M] \\
 \text{fv}[\mathbf{this}.M] & \stackrel{\text{def}}{=} \{\mathbf{this}\} \cup \text{rv}[M] \\
 \text{fv}[P_1 \& P_2] & \stackrel{\text{def}}{=} \text{fv}[P_1] \cup \text{fv}[P_2] \\
 \text{fv}[\mathbf{obj } o = C \text{ in } P] & \stackrel{\text{def}}{=} (\text{fv}[C] \setminus \{\mathbf{this}\}) \cup (\text{fv}[P] \setminus \{o\}) \\
 \text{fv}[\mathbf{class } c = C \text{ in } P] & \stackrel{\text{def}}{=} \text{fv}[C] \cup (\text{fv}[P] \setminus \{c\})
 \end{array}$$

For classes :

$$\begin{array}{ll}
 \text{fv}[c] & \stackrel{\text{def}}{=} \{c\} \\
 \text{fv}[L] & \stackrel{\text{def}}{=} \emptyset \\
 \text{fv}[M \triangleright P] & \stackrel{\text{def}}{=} \text{fv}[P] \setminus \text{rv}[M] \\
 \text{fv}[C_1 \text{ or } C_2] & \stackrel{\text{def}}{=} \text{fv}[C_1] \cup \text{fv}[C_2] \\
 \text{fv}[\mathbf{match } C \text{ with } S \text{ end}] & \stackrel{\text{def}}{=} \text{fv}[C] \cup \text{fv}[S] \\
 \text{fv}[C \text{ init } P] & \stackrel{\text{def}}{=} \text{fv}[C] \cup \text{fv}[P]
 \end{array}$$

$$\begin{array}{ll}
 \text{dl}[c] & \stackrel{\text{def}}{=} \emptyset \\
 \text{dl}[L] & \stackrel{\text{def}}{=} \emptyset \\
 \text{dl}[M \triangleright P] & \stackrel{\text{def}}{=} \text{dl}[M] \\
 \text{dl}[C_1 \text{ or } C_2] & \stackrel{\text{def}}{=} \text{dl}[C_1] \cup \text{dl}[C_2] \\
 \text{dl}[\mathbf{match } C \text{ with } S \text{ end}] & \stackrel{\text{def}}{=} \text{dl}[C] \cup \text{dl}[S] \\
 \text{dl}[C \text{ init } P] & \stackrel{\text{def}}{=} \text{dl}[C]
 \end{array}$$

For refinement sequences :

$$\begin{array}{ll}
 \text{fv}[\emptyset] & \stackrel{\text{def}}{=} \emptyset \\
 \text{fv}[K_1 \Rightarrow K_2 \triangleright P \mid S] & \stackrel{\text{def}}{=} (\text{fv}[P] \setminus \text{rv}[K_2]) \cup \text{fv}[S]
 \end{array}$$

$$\begin{array}{ll}
 \text{dl}[\emptyset] & \stackrel{\text{def}}{=} \emptyset \\
 \text{dl}[K_1 \Rightarrow K_2 \triangleright P \mid S] & \stackrel{\text{def}}{=} (\text{dl}[K_2] \setminus \text{dl}[K_1]) \cup \text{dl}[S]
 \end{array}$$

For solutions :

$$\text{fv}[\mathcal{D}] \stackrel{\text{def}}{=} \bigcup_{x.D \in \mathcal{D}} (\{x\} \cup \text{fv}[D]) \qquad \text{fv}[\mathcal{P}] \stackrel{\text{def}}{=} \bigcup_{\phi \# P \in \mathcal{P}} \text{fv}[P]$$

Figure 4: Free variables $\text{fv}[\cdot]$, received variables $\text{rv}[\cdot]$, and defined labels $\text{dl}[\cdot]$ in $\text{OJoin}_{\mathcal{H}}$

wild-card and match any reaction rules during selective refinement. Finally, inheritance is expressed by referring to the names of parent classes.

Process **class** $c = C$ **hide** F **in** P is new. It is the only binder for class name c . We apply hiding at class binding time, where F denotes a set of private labels defined in C . The result of hiding is a new class definition, whose implementation is the same as the one of C , but exports a restricted interface where the labels in F are absent. When F is empty we omit it, getting the simple class binding construct **class** $c = C$ **in** P .

Besides class name binders, object name binders include: object definitions (binding defined objects) and join-patterns (binding formal arguments). In particular, we follow the scoping of reaction rules for refinement clauses $K_1 \Rightarrow K_2 \triangleright P$, *i.e.* the formal arguments of K_2 are bound in process P . Moreover, a well-formedness condition is imposed on the formal arguments of K_1 and K_2 : $\text{rv}[K_1] \subseteq \text{rv}[K_2]$. This condition is needed so as to prevent selective refinement from introducing free variables in guarded processes. Formal scoping rules appear in Figure 4.

4 Evaluating OJoin _{$\mathcal{H}$} into ObJoin

The semantics of the class language is expressed by evaluating class definitions to plain object definitions, *i.e.* join-definitions together with an optional initializer: D **init** P . Such classes are the only ones that can be instantiated. However, abstract labels introduce a slight complication. In general, classes are evaluated to the following *class normal forms*³:

$$C_v ::= (D \text{ or } L) \text{ init } P_v$$

Semantic rules that transform OJoin _{$\mathcal{H}$} terms into ObJoin terms appear in Figure 5. They are deterministic big-step evaluation rules [29], expressing call-by-value reduction. Note that the choice between call-by-value and call-by-name strategies is not critical because both yield the same results as observed in [21]. However, motivated by reality, we prefer to avoid repetitive evaluation of the same class definition, based upon the assumption that all classes in a program are useful. Compared with the reduction semantics proposed in [15], our evaluation semantics provides a more direct formalism of implementation.

4.1 Evaluation judgments

The following judgments are used in this big-step semantics.

Process evaluation	$\Downarrow_P : \Gamma \models P \Downarrow_P P_v$
Class evaluation	$\Downarrow_C : \Gamma \models C \Downarrow_C C_v$
Filter evaluation	$\Downarrow_S : D \text{ with } S \Downarrow_S C$

Note that we use evaluation environment Γ (in stead of substitution as in [15]) in process and class judgements, which again reflects implementation better.

Denoting the set of class normal forms as $\mathcal{C}_v$, an *evaluation environment* Γ is a function from class names to class normal forms, $\Gamma : \mathcal{C} \rightarrow \mathcal{C}_v$. We say an environment Γ is empty if it is defined nowhere on $\mathcal{C}$, that is: $\forall x \in \mathcal{C}, \Gamma(x) = \perp$. Empty environment is written $[\]$, and

³Processes of ObJoin are explicitly denoted by P_v or Q_v when there is an potential ambiguity.

Rules for processes

$$\begin{array}{c}
\text{EVAL-NUL} \\
\Gamma \models 0 \Downarrow_P 0 \\
\\
\text{EVAL-HIDE} \\
\frac{\Gamma \models C \Downarrow_C C_v \quad (f_i \in \text{dl}[C_v], h_i \text{ fresh})^{i \in I} \quad \Gamma + (c \mapsto C_v \{h_i / f_i\}_{i \in I})_{\mathcal{H}} \models P \Downarrow_P P_v}{\Gamma \models \mathbf{class } c = C \mathbf{ hide } \{f_i\}_{i \in I} \mathbf{ in } P \Downarrow_P P_v} \\
\\
\text{EVAL-OBJECT} \quad \text{EVAL-PARALLEL} \\
\frac{\Gamma \models C \Downarrow_C (D \text{ or } \emptyset) \mathbf{ init } Q_v \quad \Gamma \models P \Downarrow_P P_v}{\Gamma \models \mathbf{obj } x = C \mathbf{ in } P \Downarrow_P \mathbf{obj } x = D \mathbf{ init } Q_v \mathbf{ in } P_v} \quad \frac{\Gamma \models P \Downarrow_P P_v \quad \Gamma \models Q \Downarrow_P Q_v}{\Gamma \models P \& Q \Downarrow_P P_v \& Q_v} \\
\\
\text{EVAL-SEND} \\
\Gamma \models u.M \Downarrow_P u.M
\end{array}$$

Rules for classes

$$\begin{array}{c}
\text{EVAL-CNAME} \quad \text{EVAL-REACTION} \\
\frac{\Gamma(c) = C_v \quad \{h_i\}_{i \in I} = \text{dl}[C_v] \upharpoonright \mathcal{H} \quad (h'_i \text{ fresh})^{i \in I}}{\Gamma \models c \Downarrow_C C_v \{h'_i / h_i\}_{i \in I} \mathcal{H}} \quad \frac{\Gamma \models P \Downarrow_P P_v}{\Gamma \models M \triangleright P \Downarrow_C M \triangleright P_v} \\
\\
\text{EVAL-ABSTRACT} \quad \text{EVAL-DISJUNCTION} \\
\frac{\Gamma \models L \Downarrow_C L \quad \Gamma \models C_1 \Downarrow_C (D_1 \text{ or } L_1) \mathbf{ init } P_v \quad \Gamma \models C_2 \Downarrow_C (D_2 \text{ or } L_2) \mathbf{ init } P'_v \quad L = (L_1 \setminus \text{dl}[D_2]) \cup (L_2 \setminus \text{dl}[D_1])}{\Gamma \models C_1 \text{ or } C_2 \Downarrow_C (D_1 \text{ or } D_2 \text{ or } L) \mathbf{ init } (P_v \& P'_v)} \\
\\
\text{EVAL-REFINEMENT} \\
\frac{\Gamma \models C \Downarrow_C (D \text{ or } L) \mathbf{ init } P_v \quad D \text{ with } S \Downarrow_S C' \quad \Gamma \models C' \text{ or } L \Downarrow_C D' \text{ or } L'}{\Gamma \models \mathbf{match } C \text{ with } S \mathbf{ end } \Downarrow_C (D' \text{ or } L') \mathbf{ init } P_v} \\
\\
\text{EVAL-INITIALIZER} \\
\frac{\Gamma \models C \Downarrow_C (D \text{ or } L) \mathbf{ init } Q_v \quad \Gamma \models P \Downarrow_P P_v}{\Gamma \models C \mathbf{ init } P \Downarrow_C (D \text{ or } L) \mathbf{ init } (Q_v \& P_v)}
\end{array}$$

Rules for filters

$$\begin{array}{c}
\text{FILTER-OR} \quad \text{FILTER-NEXT} \\
\frac{D_1 \text{ with } S \Downarrow_S C_1 \quad D_2 \text{ with } S \Downarrow_S C_2}{D_1 \text{ or } D_2 \text{ with } S \Downarrow_S C_1 \text{ or } C_2} \quad \frac{M \triangleright P_v \text{ with } S \Downarrow_S C \quad \text{dl}[K_1] \not\subseteq \text{dl}[M]}{M \triangleright P_v \text{ with } K_1 \Rightarrow K_2 \triangleright Q \mid S \Downarrow_S C} \\
\\
\text{FILTER-END} \\
D \text{ with } \emptyset \Downarrow_S D \\
\\
\text{FILTER-APPLY} \\
\frac{M \equiv K_1 \& K \quad \text{rv}[K_2] \cap \text{rv}[K] = \emptyset \quad \text{dl}[K_2] \cap \text{dl}[K] = \emptyset \quad L = \text{dl}[K_1] \setminus \text{dl}[K_2]}{M \triangleright P_v \text{ with } K_1 \Rightarrow K_2 \triangleright Q \mid S \Downarrow_S K_2 \& K \triangleright P_v \& Q \text{ or } L}
\end{array}$$

Figure 5: Class evaluation rules of the OJoin_H calculus

we simply write $\models P \Downarrow_P P_v$ ($\models C \Downarrow_C C_v$) for $[] \models P \Downarrow_P P_v$ ($[] \models C \Downarrow_C C_v$). We define the extension of environments as follows:

$$(\Gamma + (c \mapsto C_v))(c') = \begin{cases} \Gamma(c') & c' \neq c \\ C_v & c' = c \end{cases}$$

4.2 Evaluation rules

Rules for processes Rule EVAL-HIDE describes how class binding **class** $c = C$ **hide** $\{f_i^{i \in I}\}$ **in** P is performed. Definition C is first evaluated to normal form C_v . Then a hiding procedure $C_v\{h_i/f_i^{i \in I}\}_{\mathcal{H}}$ is called for before we bind the resulting class value to c . Represented by the indexed set $\{f_i^{i \in I}\}$, hidden labels are required be private. Moreover, the condition in the premise $(f_i \in \text{dl}[D])^{i \in I}$ requires them also to be concrete. We implement $\{h_i/f_i^{i \in I}\}_{\mathcal{H}}$ by α -converting the hidden channels $\{f_i^{i \in I}\}$ to fresh labels $\{h_i^{i \in I}\}$. In order to guarantee the freshness, we isolate a subset of $\mathcal{F}$, called $\mathcal{H}$, and we require $\mathcal{H}$ not be accessible by programmers. Whenever a channel is to be hidden, we pick a fresh label from $\mathcal{H}$, and do the α -conversion. Here $\{h_i^{i \in I}\}$ are the fresh labels picked up for $\{f_i^{i \in I}\}$ respectively.

A formal definition of our hiding procedure is given in Figure 6. We simply write $\{./\}_{\mathcal{H}}$ for $\{h_i/f_i^{i \in I}\}_{\mathcal{H}}$ in inductive cases. The α -conversion applies to both definition occurrences (in join-patterns) and reference occurrences (in guarded processes and the **init** process) of the f_i s in C_v . It is important to notice that we can rename *all* reference occurrences of hidden labels because hidden labels are private. Moreover, in the last rule, we do not recursively go into new object definitions because they rebind **this**. Additionally, abstract labels L are not influenced by renaming because only concrete labels can be hidden. To give some intuition, the normal form of class **c_hidden_buffer** (Section 3.3) looks as follows:

```
class c_hidden_buffer =
  get(r) & Some'(n) ▷ r.reply(n) & this.Empty'()
or put(n,r) & Empty'() ▷ r.reply() & this.Some'(n)
init this.Empty'()
```

Here, we assume **Empty'** and **Some'** to be the two fresh labels replacing **Empty** and **Some** respectively.

Rule EVAL-OBJECT describes how objects are created from classes. We explicitly require the abstract part of the corresponding class normal form be empty in the premise, in order to rule out any attempt of instantiating abstract classes.

Rules for classes These rules evaluate a class definition to its normal form. Classes may refer to some class name in definitions as in Rule EVAL-CNAME, and following the call-by-value strategy, the definitions listed in the environment Γ is already in normal form. However, these normal forms can not be return directly. As we have discussed, hidden names should never be touched by later inheritance and overriding. Consider the following situation: we have a class c whose normal form C_v contains a hidden name h and other two classes c_1 and c_2 both derived from class c , namely referring to c in their definitions. If we returned C_v directly, then both c_1 and c_2 would also have the hidden name h in their normal forms. As a consequence, when we combine c_1 and c_2 in disjunction, the two occurrences of h will have an influence on each other, which is against the spirit of hiding. Therefore, we require the re-freshening of the hidden names whenever a class is inherited in the EVAL-CNAME rule to avoid the collision.

$((D \text{ or } L) \text{ init } P_v)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$(D\{\cdot/\cdot\}_{\mathcal{H}} \text{ or } L) \text{ init } P_v\{\cdot/\cdot\}_{\mathcal{H}}$
$f(\tilde{u})\{h_i/f_i\}_{i \in I}\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$\begin{cases} h_j(\tilde{u}) & f = f_j, j \in I \\ f(\tilde{u}) & \text{otherwise} \end{cases}$
$(M_1 \& M_2)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$M_1\{\cdot/\cdot\}_{\mathcal{H}} \& M_2\{\cdot/\cdot\}_{\mathcal{H}}$
$(M \triangleright P)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$M\{\cdot/\cdot\}_{\mathcal{H}} \triangleright P\{\cdot/\cdot\}_{\mathcal{H}}$
$(D_1 \text{ or } D_2)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$D_1\{\cdot/\cdot\}_{\mathcal{H}} \text{ or } D_2\{\cdot/\cdot\}_{\mathcal{H}}$
$0\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	0
$(\text{this}.M)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$\text{this}.M\{\cdot/\cdot\}_{\mathcal{H}}$
$(x.M)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$x.M$
$(P_1 \& P_2)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$P_1\{\cdot/\cdot\}_{\mathcal{H}} \& P_2\{\cdot/\cdot\}_{\mathcal{H}}$
$(\text{obj } x = D \text{ init } P \text{ in } Q)\{\cdot/\cdot\}_{\mathcal{H}}$	$\stackrel{\text{def}}{=}$	$\text{obj } x = D \text{ init } P \text{ in } Q\{\cdot/\cdot\}_{\mathcal{H}}$

Figure 6: α -converting hidden names to fresh names in class normal forms

Rule EVAL-DISJUNCTION describes how two definitions are cumulated. We evaluate both sub-definitions into normal forms and combine the results together. Abstract labels that are defined elsewhere are discarded, and the two initializers from both sub-definitions are composed in parallel. Selective refinement applies to class normal forms $(D \text{ or } L) \text{ init } P_v$. Rule EVAL-REFINEMENT evaluates selective refinements by means of evaluating *filters* of the form $D \text{ with } S$ to C' , defined below. Another round of evaluation of $C' \text{ or } L$ is subsequently called for, in order to combine the new class with original abstract part and to evaluate the newly-added processes to normal forms. Note that selective refinement has no impact on initializer. Finally, rule EVAL-INITIALIZER tells how new **init** process is composed in parallel with the existing one.

Rules for filters The extra bunch of rules presented at the bottom of Figure 5 evaluates filters. A filter refines a join-definition D against a sequence of refinement clauses S . Each reaction rule in D is refined independently from the others according to the leftmost matching clause if any (rules FILTER-NEXT and FILTER-END), and the resulting classes are combined by class disjunction (rule FILTER-OR). We say a clause $(K_1 \Rightarrow K_2 \triangleright Q)$ matches a reaction rule $(M \triangleright P_v)$ when the selective pattern K_1 is a sub-pattern of the join pattern M , namely, $\text{dl}[K_1] \subseteq \text{dl}[M]$. More precisely, with necessary α -conversion of formal arguments, and commutativity and associativity of the operator “&” in join-patterns, it is equivalent to require M be structurally congruent to $K_1 \& K$ as stated in rule FILTER-APPLY, where K is another pattern. Once a matching clause is found, the reaction rule is rewritten by substituting K_2 for K_1 in M , and composing P_v in parallel with Q . To preserve the linearity of formal arguments and labels in join-patterns, two other side conditions are also required, *i.e.* (1) $\text{rv}[K_2] \cap \text{rv}[K] = \emptyset$, (2) $\text{dl}[K_2] \cap \text{dl}[K] = \emptyset$, as stated in the premise. Condition (1) can be guaranteed by α -conversion, and the type system will check for condition (2). The replacement of K_1 by K_2 has two effects: some new channels may be introduced into this reaction

rules by K_2 , they are $\text{dl}[K_2] \setminus \text{dl}[K_1]$; and on the contrary, the channels in $\text{dl}[K_1] \setminus \text{dl}[K_2]$ loose their definitions and become abstract. For more detailed discussion on new labels and abstract labels, please refer to our previous work [22]. Finally, if no matching clause is found, the reaction rule remains the same (rule FILTER-END).

4.3 Evaluation errors

We enforce certain conditions in the premises of some evaluation rules in Figure 5. For example, we require the abstract label set L be empty when a class is instantiated in rule EVAL-OBJECT. However, any of these conditions may fail and we then say that an “error” occurs in those cases. By convention, as soon as an error is reported, the whole evaluation terminates and returns the error as result. In order to specify those errors, we give some **error** rules in the following, to accompany the correct ones given in Figure 5.

- Abstract class instantiation:

$$\frac{\text{EVAL-OBJECT-ERROR} \quad \Gamma \models C \Downarrow_C (D \text{ or } L) \text{ init } Q_v \quad L \neq \emptyset}{\Gamma \models \text{obj } x = C \text{ in } P \Downarrow_P \text{ error}}$$

- Unbound class name:

$$\frac{\text{EVAL-CNAME-ERROR} \quad c \notin \text{dom}[\Gamma]}{\Gamma \models c \Downarrow_C \text{ error}}$$

- Non-linear join-pattern generation:

$$\frac{\text{FILTER-APPLY-ERROR} \quad M \equiv K_1 \& K \quad \text{rv}[K_2] \cap \text{rv}[K] = \emptyset \quad \text{dl}[K_2] \cap \text{dl}[K] \neq \emptyset}{M \triangleright P_v \text{ with } K_1 \Rightarrow K_2 \triangleright Q \mid S \Downarrow_S \text{ error}}$$

- Hiding public labels:

$$\frac{\text{EVAL-HIDE-ERROR1} \quad \exists j \in I, \text{ such that } l_j \in \mathcal{M}}{\Gamma \models \text{class } c = C \text{ hide } \{l_i^{i \in I}\} \text{ in } P \Downarrow_P \text{ error}}$$

- Hiding abstract labels:

$$\frac{\text{EVAL-HIDE-ERROR2} \quad \Gamma \models C \Downarrow_C C_v \quad \exists j \in I, \text{ such that } f_j \notin \text{dl}[C_v]}{\Gamma \models \text{class } c = C \text{ hide } \{f_i^{i \in I}\} \text{ in } P \Downarrow_P \text{ error}}$$

5 The type system of OJoin_H

We define a ML-style type system for OJoin_H. It enriches our previous design [22] to type the hiding operation, and consequently, features more abstract class types.

5.1 Type algebra

The grammar of type expressions appears in Figure 7.

Types:		
$\tau ::=$	$\alpha \mid [\rho]$	Object type
$\rho ::=$	$\emptyset \mid \varrho \mid m : \tilde{\tau}; \rho$	Row type
$\tilde{\tau} ::=$	$(\tau_i^{i \in I})$	Tuple type
$\tau^c ::=$	$\zeta(\tau) B^{W,V}$	Class type
$B ::=$	$\emptyset \mid l : \tilde{\tau}; B$	Internal type
$\tau^s ::=$	B^R	Refinement sequence type
$R ::=$	$\emptyset \mid \pi_1 \Rightarrow \pi_2; R$	Refinement rules
Types schemes:		
$\zeta ::=$	$\forall X. \tau$	Object type scheme
$\zeta^c ::=$	$\forall X. \tau^c$	Class type scheme

Figure 7: Type algebra

Types There are two kinds of type variables: object type variables, ranged over by α ; and row type variables, ranged over by ϱ . We use θ for type variables, regardless their kinds, and denote the set of all possible type variables as Θ . We use X and Y to range over sets of type variables, namely, we have $X, Y \subseteq \Theta$. As in the ML type system, polymorphism is parametric polymorphism, obtained essentially by generalizing the free type variables.

Object types $\tau = [\rho]$ list the types of public channels. They may end with a row variable, which means that, besides channels listed in ρ , there may be some other channels. Such trailing row variables enable a useful degree of subtyping polymorphism by structure.

Class types $\tau^c = \zeta(\rho) B^{W,V}$ consist of four parts. Objects created from the class have type $[\rho]$. Internal type B collects the types of all (non-hidden) channels in the class, defined or declared, public or private. We shall describe V later. Finally, W reflects existing synchronization amongst channels. Component W is a set of sets of channel names, with one member set $w \subseteq \mathcal{L}$ corresponding to one join-pattern, and all together representing the whole structure of join-patterns in the class normal form. Note that with the effect of hiding, hidden labels are eliminated from B and W . However, wild elimination endangers safe polymorphism. As an example, the type of class `c_buffer` from Section 3.1 is:

```
class c_buffer:
  object
    label get: ([reply: ( $\theta$ );  $\varrho$ ]) ;
    label put: ( $\theta$ , [reply: ();  $\varrho'$ ]) ;
    label Some: ( $\theta$ ) ; label Empty: () ;
  end W = {{get, Some}, {put, Empty}}
```

As detailed in [22, 14, 6], labels from the same join-patterns are identified as *correlated* and any free type variables shared by correlated labels cannot be generalized in object types. In this example, because θ is shared by two correlated labels `get` and `Some` (from the same member set of W), it should not be generalized. By contrast, class `c_hidden_buffer` from Section 3.3 hides labels `Some` and `Empty` and has type:

```
class c_hidden_buffer:
```

```

object
  label get: ([reply: ( $\theta$ );  $\rho$ ]) ;
  label put: ( $\theta$ , [reply: ();  $\rho'$ ]) ;
end  $W = \{\{\mathbf{get}\}, \{\mathbf{put}\}\}$ 

```

where the hidden labels disappear from both the label list and W . Without other restriction, this type allows the generalization of θ in object types because labels **get** and **put** are not correlated (coming from two different member sets of W). Generalized θ then could be instantiated incompatibly for **get** and **put**, for instance, as integer and string, which would result in a runtime type error: attempting to retrieve a string when an integer is present.

To tackle the problem, we keep track of such *dangerous type variables* in class types, denoted by V . Type variables appearing in V are prohibited to be generalized. As a result, the complete type of class `c_hidden_buffer` is:

```

class c_hidden_buffer:
  object
    label get: ([reply: ( $\theta$ );  $\rho$ ]) ;
    label put: ( $\theta$ , [reply: ();  $\rho'$ ]) ;
  end  $W = \{\{\mathbf{get}\}, \{\mathbf{put}\}\}$   $V = \{\theta\}$ 

```

In spite of the discrete technicality of component V , this type obviously achieves information hiding towards the inheritance users of class `c_hidden_buffer`.

Which of the channels appearing in B are abstract can easily be inferred from our types, since abstract channels never appear in join-pattern. Hence we can compute the set of abstract channels as $\text{dom}[B] \setminus \overline{W}$, where $\text{dom}[B]$ is the set of channels listed in B , and $\overline{W}$ is the flattening of W , namely, union of all member sets.

Refinement sequence types $\tau^s = B^R$ abstract the external features of selective refinement sequences S , in which the internal type B lists the types of all the labels appearing in S , and R sketches refinement clauses into *refinement rules*.

Usually, we omit the trailing $\emptyset$ in ρ , in B or in R . Note that names bound in ρ or B are pair-wise distinct, and the order of them does not matter. Thus, they can also be seen as sets of (label, tuple type) pairs. However, the order of refinement rules in R does matter. It reflects the order of refinement clauses.

Type schemes For class types, we generalize all the free type variables and get the corresponding class type scheme. While for object types, only those are neither annotated as dangerous, nor shared by any correlated channels can be made polymorphic.

Usually, we omit the empty set of generalized variables in a type scheme, namely we abbreviate $\forall \emptyset. \tau$ ($\forall \emptyset. \tau^c$) as τ (τ^c).

5.2 Type checking processes and classes

The following typing judgments are used in the type system. Recall that u stands for either object names or the keyword **this**.

$A \vdash u : \tau$	object u has type τ in A
$A \vdash u.l :: \tilde{\tau}$	label l of object u has type $\tilde{\tau}$ in A
$A \vdash P$	process P is well-typed in A
$A \vdash K :: B$	pattern K has type B in A
$A \vdash C :: \tau^c$	class C has type τ^c in A
$A \vdash S :: \tau^s$	refinement sequence S has type τ^s in A
$\vdash W \text{ with } R :: W'$	refining W against R gives W'

Typing judgments rely on *type environments* A that bind class names, object names or **this** to corresponding type schemes:

$$A ::= [] \mid c : \varsigma^c; A \mid u : \varsigma; A \mid u : \forall X. B; A$$

An object u may have two complementary bindings in A , namely, $u : \varsigma$ (external scheme) for public labels, and $u : \forall X. B$ (internal scheme) for private labels, where $\text{dom}[B] \subseteq \mathcal{F}$. For simplicity, we omit empty environments $[]$ in typing judgments.

Typing rules appear in Figure 8 and Figure 9. Before looking at them, we first summarize the used notations as follows.

- $\{\gamma_\theta / \theta^{\theta \in X}\}$ expresses the substitution of type variables of domain X by types. Note that γ_θ is either an object type (τ) or a row type (ρ).
- $\text{dom}[A]$ is the set of identifiers bound in A . $A + A'$ equals $(A \setminus \text{dom}[A']) \cup A'$, where $A \setminus \text{dom}[A']$ removes the bindings for $\text{dom}[A']$ from A . More specifically, $A + u : \forall X. [\rho], u : \forall X. B$ means $A \setminus \{u\} \cup u : \forall X. [\rho] \cup u : \forall X. B$.
- $B \upharpoonright L$ restrict B to a set of labels L . And $B(l)$ refers to the tuple type bound to l in B . $B_1 \uplus B_2$ is the union of B_1 and B_2 , provided $\text{dom}[B_1] \cap \text{dom}[B_2] = \emptyset$ holds. Predicate $B_1 \uparrow B_2$ expresses that B_1 and B_2 coincide on the types of their common channels. And $B_1 \oplus B_2$ is the union of B_1 and B_2 , provided $B_1 \uparrow B_2$ holds.
- $\text{ctv}[B^W]$ computes the free type variables in B that are common to types of at least two correlated channels according to W . Namely,

$$\begin{aligned} \text{ctv}[B^W] &= \bigcup_{w_i \in W} \text{ctv}[B^{\{w_i\}}] \\ \text{ctv}[B^{\{w\}}] &= \bigcup_{l \in w, l' \in w, l \neq l'} \text{ftv}[B(l)] \cap \text{ftv}[B(l')] \end{aligned}$$

where $\text{ftv}[\cdot]$ denotes the set of free type variables.

- $\text{Gen}(\rho, B, A)$ returns the set of type variables free in ρ or B , but not free in type environment A . Namely, $\text{Gen}(\rho, B, A) = (\text{ftv}[\rho] \cup \text{ftv}[B]) \setminus \text{ftv}[A]$.

Rules for object names and labels

$$\frac{\text{TYPE-ONAME} \quad u : \forall X. \tau \in A}{A \vdash u : \tau\{\gamma_\theta / \theta^{\theta \in X}\}}$$

$$\frac{\text{TYPE-PUBLAB} \quad A \vdash u : [m : \tilde{\tau}; \rho]}{A \vdash u.m :: \tilde{\tau}}$$

$$\frac{\text{TYPE-PRILAB} \quad u : \forall X. (f : \tilde{\tau}; B) \in A}{A \vdash u.f :: \tau\{\gamma_\theta / \theta^{\theta \in X}\}}$$

Rules for processes

$$\frac{\text{TYPE-NULL}}{A \vdash 0}$$

$$\frac{\text{TYPE-SEND} \quad A \vdash u.l :: \tau_i^{i \in I} \quad (A \vdash u_i : \tau_i)^{i \in I}}{A \vdash u.l(u_i^{i \in I})}$$

$$\frac{\text{TYPE-JOIN} \quad A \vdash u.M_1 \quad A \vdash u.M_2}{A \vdash u.(M_1 \ \& \ M_2)}$$

$$\frac{\text{TYPE-HIDE} \quad \begin{array}{l} A + \mathbf{this} : [\rho]; \mathbf{this} : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V} \\ B' = B \setminus F \end{array} \quad \begin{array}{l} \rho = B \upharpoonright \mathcal{M}; \varrho \\ W' = W \setminus F \\ F \subseteq \overline{W} \end{array}}{\text{TYPE-PARALLEL} \quad \frac{A \vdash P \quad A \vdash Q}{A \vdash P \ \& \ Q} \quad \frac{A + c : \forall \text{Gen}(\rho, B', A). \zeta(\rho)B'^{W',V \cup \text{ftv}[B \upharpoonright F]} \vdash P}{A \vdash \mathbf{class} \ c = C \ \mathbf{hide} \ F \ \mathbf{in} \ P}}$$

$$\frac{\text{TYPE-OBJECT} \quad \begin{array}{l} A + \mathbf{this} : [\rho]; \mathbf{this} : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V} \\ X = \text{Gen}(\rho, B, A) \setminus \text{ctv}[B^W] \setminus V \\ A + x : \forall X. [\rho] \vdash P \end{array} \quad \begin{array}{l} \rho = B \upharpoonright \mathcal{M} \\ \text{dom}[B] = \overline{W} \end{array}}{A \vdash \mathbf{obj} \ x = C \ \mathbf{in} \ P}$$

Figure 8: Typing rules for processes in the OJoin_H calculus

Typing processes Processes are typed following the second bunch of rules in Figure 8. In rules TYPE-HIDE and TYPE-OBJECT, we explicitly extend the environment A with bindings for the recursive self reference **this** when typing the class definitions. As a side note, it should perhaps be noticed that the conditions $\rho = B \upharpoonright \mathcal{M}; \varrho$ and $\rho = B \upharpoonright \mathcal{M}$ in these rules enforce that all public labels are listed in the B component of class types and allows us to omit component ρ in our examples of class types.

With the notion of dangerous type variables, the polymorphism control of object type elaborates into two parts in rule TYPE-OBJECT for object definitions. Besides the common free type variables that are shared by correlated channels (*i.e.* computed by $\text{ctv}[B^W]$), the set of dangerous type variables V is also prevented from generalization.

Rule TYPE-HIDE types class binding with hiding. Because $\overline{W}$ lists all the defined labels, the condition $F \subseteq \overline{W}$ in the premise checks whether all the hidden private names are actually defined. The type of objects created from the class (object user interface) $[\rho]$ remains the same, because ρ contains only public labels. However, the interface for inheritance users, namely B^W is restricted to $B'^{W'}$, where $B' = B \setminus F$ removing from B the binding of labels in F , and $W' = W \setminus F$ removing from W labels in F . $W \setminus F$ is defined as $\{w_i \setminus F \mid w_i \in W\}$, where $w_i \setminus F$ refers to usual set difference. Moreover, to assure safe polymorphism, all the free type variables of the hidden channels ($\text{ftv}[B \upharpoonright F]$) are considered as dangerous, and are appended into V . One might wonder why we did not add those from $\text{ctv}[B^W] \setminus \text{ctv}[B'^{W'}]$,

which at first glance seems feasible and would allow more polymorphism. Unfortunately, the answer is “we can’t” and the reason is of no surprise: this would not be safe.

We temporarily replace $\text{ftv}[B \upharpoonright \mathcal{F}]$ by $\text{ctv}[B^W] \setminus \text{ctv}[B'^{W'}]$ in rule TYPE-HIDE and demonstrate why this change may impair safety by the following examples. Consider a class definition:

```
class  $c_1$  =
   $a(x) \triangleright 0$ 
  or  $b(y) \ \& \ \text{Ch}(n_1, n_2) \triangleright \text{this}.(a(n_1) \ \& \ b(n_2))$ 
  hide {Ch}
```

Its normal form is:

```
 $a(x) \triangleright 0$ 
or  $b(y) \ \& \ \text{Ch}'(n_1, n_2) \triangleright \text{this}.(a(n_1) \ \& \ b(n_2))$ 
```

where the label **Ch** is α -converted to $\text{Ch}' \in \mathcal{H}$ during hiding. The type system gives the following types respectively for the definition (on the left) and the normal form (on the right):

<pre>object label a: (θ); label b: (θ') end $W = \{\{a\}, \{b\}\} \ V = \{\theta'\}$</pre>	<pre>object label a: (θ); label b: (θ'); label Ch': (θ, θ') end $W = \{\{a\}, \{b, \text{Ch}'\}\} \ V = \emptyset$</pre>
---	---

If we create an object from class c_1 , both types render the same object type: $\forall \theta.[a : (\theta); b : (\theta')]$, in which the type variable θ' is monomorphic. (The type on the left restricts the generalization of θ' because $\theta' \in V$, and the type on the right does so because θ' is shared by two correlated labels **b** and Ch' .) We then modify class c_1 by selective refinement as follows:

```
class  $c_2$  = match  $c_1$  with
   $b(y) \Rightarrow b(y) \ \& \ d(z) \triangleright \text{this}.a(z)$  end
```

This yields the following class in normal form:

```
class  $c_2$  =
   $a(x) \triangleright 0$ 
  or  $b(y) \ \& \ d(z) \ \& \ \text{Ch}'(n_1, n_2) \triangleright$ 
     $\text{this}.(a(n_1) \ \& \ b(n_2) \ \& \ a(z))$ 
```

The type system infers the left type for the selective refinement based on the type of class c_1 , and the normal form has the type on the right:

<pre>object label a: (θ); label b: (θ'); label d: (θ) end $W = \{\{a\}, \{b, d\}\} \ V = \{\theta'\}$</pre>	<pre>object label a: (θ); label b: (θ'); label Ch': (θ, θ'); label d: (θ) end $W = \{\{a\}, \{b, \text{Ch}', d\}\} \ V = \emptyset$</pre>
--	--

We create an object from class c_2 , and here comes the problem. The left hand type gives the polymorphic type: $\forall \theta.[a : (\theta); b : (\theta'); d : (\theta)]$ to the object. However, the type variable θ is apparently forbidden to be generalized according to the normal form, because Ch' and d correlated and they share θ . The wrong polymorphism authorized by the typing is safety-violating.

Rules for patterns

TYPE-EMPTYPATTERN $A \vdash 0 :: \emptyset$	TYPE-MESSAGE $\frac{(A \vdash x_i : \tau_i)_{i \in I}}{A \vdash l(x_i^{i \in I}) :: (l : \tau_i^{i \in I})}$	TYPE-SYNCHRONIZATION $\frac{A \vdash M_1 :: B_1 \quad A \vdash M_2 :: B_2}{A \vdash M_1 \& M_2 :: B_1 \uplus B_2}$
---	--	--

Rules for classes

TYPE-CNAME $\frac{c : \forall X. \zeta(\rho) B^{W,V} \in A}{A \vdash c :: (\zeta(\rho) B^{W,V}) \{\gamma_\theta / \theta^{\theta \in X}\}}$	TYPE-ABSTRACT $\frac{\text{dom}[B] = L}{A \vdash L :: \zeta(\rho) B^{\emptyset, \emptyset}}$
TYPE-REACTION $\frac{A' \vdash M :: B \quad A + A' \vdash P \quad \text{dom}[A'] = \text{rv}[M]}{A \vdash M \triangleright P :: \zeta(\rho) B^{\{\text{dl}[M]\}, \emptyset}}$	
TYPE-DISJUNCTION $\frac{A \vdash C_1 :: \zeta(\rho) B_1^{W_1, V_1} \quad A \vdash C_2 :: \zeta(\rho) B_2^{W_2, V_2}}{A \vdash C_1 \text{ or } C_2 :: \zeta(\rho) (B_1 \oplus B_2)^{W_1 \cup W_2, V_1 \cup V_2}}$	
TYPE-REFINEMENT $\frac{A \vdash C :: \zeta(\rho) B^{W,V} \quad A \vdash S :: B'^R \quad \vdash W \text{ with } R :: W' \quad B \uparrow B'}{A \vdash \text{match } C \text{ with } S \text{ end} :: \zeta(\rho) ((B' \uparrow \overline{W'}) \oplus B)^{W', V}}$	TYPE-INITIALIZER $\frac{A \vdash C :: \zeta(\rho) B^{W,V} \quad A \vdash P}{A \vdash C \text{ init } P :: \zeta(\rho) B^{W,V}}$

Rules for refinement clauses

TYPE-CLAUSE $\frac{A' \vdash K_1 :: B_1 \quad A' \vdash K_2 :: B_2 \quad \text{dom}[A'] = \text{rv}[K_2] \quad A + A' \vdash P \quad A \vdash S :: B^R}{A \vdash K_1 \Rightarrow K_2 \triangleright P \mid S :: (B_1 \oplus B_2 \oplus B)^{(\text{dl}[K_1] \Rightarrow \text{dl}[K_2]) \mid R}}$	TYPE-EMPTYCLAUSE $A \vdash \emptyset :: \emptyset^\emptyset$
--	--

Rules for filters

TYPE-APPLY $\vdash w_1 \uplus w \text{ with } w_1 \Rightarrow w_2 \mid R :: w_2 \uplus w$	TYPE-END $\vdash w \text{ with } \emptyset :: w$	TYPE-NEXT $\frac{w_1 \not\subseteq w \quad \vdash w \text{ with } R :: w'}{\vdash w \text{ with } w_1 \Rightarrow w_2 \mid R :: w'}$
TYPE-OR $\frac{(\vdash w_i \text{ with } R :: w'_i)_{i \in I}}{\vdash \{w_i^{i \in I}\} \text{ with } R :: \{w'_i^{i \in I}\}}$		

Figure 9: Typing rules for patterns, classes, and filters in OJoin_H

TYPE-SOLUTION $\frac{(A^{\mathcal{M}} \vdash P)^{P \in \mathcal{P}} \quad A = \bigcup_{x.D \in \mathcal{D}} A_x \quad (A^{\mathcal{M}} \vdash x.D :: A_x)_{x.D \in \mathcal{D}}}{(A^{\mathcal{M}} \vdash P)^{\Phi \# P \in \mathcal{P}} \quad (A \vdash P)^{\Phi \# P \in \mathcal{P}} \quad (A^{\mathcal{M}} + A_x\{\mathbf{this}/x\} \vdash P)^{(\mathbf{this} \mapsto x) \# P \in \mathcal{P}}}$ $\vdash (\mathcal{D} \Vdash \mathcal{P})$	
TYPE-DEFINITION $\frac{A + \mathbf{this} : [\rho], \mathbf{this} : (B \upharpoonright \mathcal{F}) \vdash D :: \zeta(\rho)B^{W, \emptyset} \quad X = \text{Gen}(\rho, B, A) \setminus \text{ctv}[B^W] \quad \rho = B \upharpoonright \mathcal{M} \quad \text{dom}[B] = \overline{W}}{A \vdash x.D :: x : \forall X. [\rho], x : \forall X. (B \upharpoonright \mathcal{F})}$	

Figure 10: Typing rules for chemical solutions in $\text{OJoin}_{\mathcal{H}}$

Typing classes The typing rules of class definitions appear in the middle of Figure 9. Rule TYPE-INITIALIZER is new for typing initializers. It is straightforward. We require both the class definition and the initializer be typable. Not surprisingly, the rest rules are almost kept the same as before in [22], except for the parts managing V . Dangerous type variables only come from hiding at class binding time (*i.e.* outside class definitions). However, for safe polymorphism reason, they should be preserved during class operation. Moreover, in rule TYPE-CNAME, the substitution of type variables in class types is redefined to deal with V as $\eta(\zeta(\rho)B^{W,V}) = \zeta(\eta(\rho))\eta(B)^{W, \eta(V)}$ where:

$$\eta(V) = (V \setminus \text{dom}[\eta]) \cup \bigcup_{\theta \in (V \cap \text{dom}[\eta])} \text{ftv}[\eta(\theta)]$$

Intuitively, this means when a dangerous type variable is replaced by a type, all the free type variables in that type are dangerous.

5.3 Type checking chemical solutions

The typing is finally extended to chemical solutions in Figure 10 in order to illustrate the properties of the type system. The judgment $\vdash (\mathcal{D} \Vdash \mathcal{P})$ states that the chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is well typed. And the auxiliary judgment $A \vdash x.D :: x : \forall X. [\rho], x : \forall X. (B \upharpoonright \mathcal{F})$ means that in the environment A , the active object $x.D$ defines an object x with polymorphic public type $\forall X. [\rho]$, and polymorphic private type $\forall X. (B \upharpoonright \mathcal{F})$.

In rule TYPE-SOLUTION, we require all the active objects from $\mathcal{D}$ and all the processes from $\mathcal{P}$ are well-typed, and we record the public and private types of all the active objects in A . A_x refers to the types of object x . We simply write $A^{\mathcal{M}}$ for $A \upharpoonright \mathcal{M}$, namely the public part of A and $A\{u'/u\}$ for renaming u to u' in A . Processes with different privacy annotations are typed differently. Authorized processes, namely those with annotation Φ , are typed in the complete environment A defined by $\mathcal{D}$. Internal processes of some object x , namely those with annotation $(\mathbf{this} \mapsto x)$ are typed in the public part of A plus the private type of object x referred through $\mathbf{this}$. And all the other kind of processes with an empty annotation $(\mathbf{this} \mapsto \perp)$ are typed only in the public part of A .

The typing rule TYPE-DEFINITION for active objects $x.D$ is similar to the TYPE-OBJECT rule for typing $\mathbf{obj} \ x = D \ \mathbf{in} \ 0$, but we return the polymorphic types for typing the solution. Note that the V part is always empty because no hiding happens in chemical solutions.

6 Properties of the Type System

6.1 The type system is sound

One of the main goal of static typing is to exclude programs that will cause errors at runtime. We discuss the soundness of our type system with respect to both class evaluation semantics and chemical reduction semantics.

Soundness w.r.t. class evaluation The main result is: no closed, well-typed process can evaluate to an error during the class evaluation, moreover, the result process is well-typed, too.

Theorem 1 (Soundness w.r.t. class evaluation). *Let P be a process. If $\vdash P$, then $\models P \Downarrow_P P_v$, P_v is not an error and $\vdash P_v$.*

The empty evaluation environment and the empty type environment in the theorem above are not sufficient to allow an inductive proof. We prove a stronger result in Lemma 1. For that purpose, we should first extend the typing to type class normal forms and evaluation environments with polymorphism, as we did for active objects in chemical solutions in rule TYPE-DEFINITION. However, because of hiding, the relation between the type inferred for a class definition and the type of its normal form is different from the one between the type inferred for an object definition and the type of its runtime value (*i.e.* the active object). In the later case, these two types are the same, while in the former case, a class definition and its runtime class normal form have different types. More specifically, when a definition for class c is typed and bound in the type environment A as $(c : \zeta^c)$, its corresponding normal form C_v bound in the evaluation environment Γ do not have exactly the polymorphic type ζ^c . Instead, C_v reveals ζ^c , written $A \vdash C_v :> \zeta^c$.

Intuitively, the notion of revealing conveys the same idea of hiding but in the opposite direction: $(C \text{ **hide** } F)$ “hides” C *i.e.* C “reveals” $(C \text{ **hide** } F)$. As a consequence, we define the revealing typing rule for $A \vdash C_v :> \zeta^c$ similar to the hiding typing rule (TYPE-HIDE) as follows:

$$\frac{\begin{array}{l} \text{TYPE-REVEAL} \\ A + \text{this} : [\rho]; \text{this} : (B \upharpoonright \mathcal{F}) \vdash C_v :: \zeta(\rho)B^{W,V} \\ B' = B \setminus H \\ W' = W \setminus H \end{array} \quad \begin{array}{l} \rho = B \upharpoonright \mathcal{M}; \varrho \\ H \subseteq \overline{W} \end{array}}{A \vdash C_v :> \forall \text{Gen}(\rho, B', A). \zeta(\rho)B'^{W', V \cup \text{ftv}[B \upharpoonright H]}}$$

where the revealed type $\forall \text{Gen}(\rho, B', A). \zeta(\rho)B'^{W', V \cup \text{ftv}[B \upharpoonright H]}$ is just the polymorphic type of class $(C_v \text{ **hide** } H)$ for some $H \subseteq \mathcal{H}$. Notice that revealing brings a very controlled form of subtyping.

Lifting the revealing relation to evaluation environments, we define $A \vdash \Gamma :> A'$ in a member-wise manner as:

$$\frac{\begin{array}{l} \text{TYPE-}\Gamma :> \\ (A \vdash \Gamma(c) :> A'(c))^{c \in \text{dom}[\Gamma]} \end{array}}{A \vdash \Gamma :> A'}$$

For any type environment A , we write $A^{\mathcal{O}}$ for its segment of all object bindings, and A^c for its segment of all class bindings. The stronger result for the soundness w.r.t. class evaluation with hiding is stated as the following lemma:

Lemma 1 (Strong soundness w.r.t. class evaluation). *Let A be a type environment and Γ be a class evaluation environment, such that $A^O \vdash \Gamma :> A^C$. If $A \vdash P$, then $\Gamma \models P \Downarrow_P P_v$, P_v is not an error and $A^O \vdash P_v$.*

Proof. See Appendix A.2. □

Soundness w.r.t. chemical reduction We go on to discuss the soundness with respect to chemical reduction in terms of subject reduction and safety properties.

Theorem 2 (Subject reduction). *Let $\mathcal{D} \Vdash \mathcal{P}$ be a chemical solution. If $\vdash (\mathcal{D} \Vdash \mathcal{P})$ and $\mathcal{D} \Vdash \mathcal{P} \Rightarrow \mathcal{D}' \Vdash \mathcal{P}'$, then $\vdash (\mathcal{D}' \Vdash \mathcal{P}')$.*

Proof. See Appendix A.3. □

As far as safety is concerned, we first specify chemical reduction failure in the following definition.

Definition 1 (Chemical reduction failure). *We say a chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is a chemical reduction failure, if one of the following holds:*

- *Free this:* $(\mathbf{this} \mapsto \perp) \# P \in \mathcal{P}$ (briefly $P \in \mathcal{P}$), and $\mathbf{this} \in \text{fv}[P]$.
- *Undefined object name:* $\phi \# P \in \mathcal{P}$, $x \in \text{fv}[P]$ or $\phi = (\mathbf{this} \mapsto x)$, and x is not defined in $\mathcal{D}$.
- *Failed privacy:* $\phi \# x.f(\tilde{u}) \in \mathcal{P}$, and $\phi \neq \Phi$.
- *Undefined channel name:* $\Phi \# x.l(\tilde{z}) \in \mathcal{P}$, $x.D \in \mathcal{D}$, and $l \notin \text{dl}[D]$.
- *Arity mismatch:* $\Phi \# x.l(\tilde{z}) \in \mathcal{P}$, $x.D \in \mathcal{D}$, $l(\tilde{y})$ appears in a join-pattern of D , and $\tilde{y}$ and $\tilde{z}$ have different arities.

The following theorem guarantees that the type system statically rejects any failures mentioned above.

Theorem 3 (Safety). *Well-typed chemical solution is never a failure as defined in Definition 1.*

Proof. See Appendix A.4. □

Finally, to combine class evaluation and chemical reduction together, we have the following theorem stating the overall soundness property of the type system:

Theorem 4 (Soundness). *Let P be a process. If $\vdash P$, then $\models P \Downarrow_P P_v$, P_v is not an error and the chemical reduction of $\vdash P_v$ never fails.*

Proof. Because $\vdash P \iff \vdash (\Vdash P)$ by typing rules, this theorem holds as a corollary of Theorem 1, Theorem 2, and Theorem 3. □

6.2 Our hiding is “hiding”

Besides the basic soundness property, the type system and the hiding semantics conform to what common sense suggests about hidden names. We argue informally in this section, through examples.

First, the type system rules out any inheritance that tries to access a hidden name. For example, in the following definitions:

```
class c1 =
  a() ▷ this.Ch() or Ch() ▷ P
  hide {Ch}
class d1 =
  c1
  or b() ▷ this.Ch()
```

Although channel `Ch` is defined in class `c1`, it is hidden. Hence in the derived class `d1`, `Ch` is not accessible. The type system will report a “unbound name” error when typing class `d1`.

Moreover, once a name is hidden, it is reasonable to define an unrelated channel that happens to have the same name. For example, in the following code:

```
class c2 =
  a(x) ▷ this.Ch(x)
  or Ch(i) ▷ out.print_int(i)
  hide {Ch}
class d2 =
  c2
  or Ch(s) ▷ print_string s
  or b(x) ▷ this.Ch(x)
```

Both classes `c2` and `d2` are well-typed in spite of the fact that the two definitions for channel named `Ch` have incompatible types and behaviors: one receives and prints an integer and the other receives and prints a string. Besides, our hiding semantics guarantees that late-binding is not applicable during the inheritance. Namely, if `o` is an object created from class `d2`, then `o.a(x)` actually sends a message to the fresh name corresponding to the hidden `Ch`, which requires `x` be an integer, and will print out this integer. While on the other hand, `o.b(x)` calls the newly defined channel `Ch`, which waits for a string argument `x`, and will print out this string.

Finally, the exact name of the hidden channel does not matter. For example, if we define a variant class for class `c2` as follows:

```
class c'2 =
  a(x) ▷ this.A'(x)
  or A'(i) ▷ out.print_int(i)
  hide {A'}
```

Then both `c2` and `c'2` are well-typed, and provide the same interfaces to their users. Replacing `c2` by `c'2` in the definition of class `d2` makes no difference.

7 Prototyping OJoin_H

As a proof of concepts, we coded a prototyping system for OJoin_H. This prototype is not to be confused with JoCaml, our implementation of the join-calculus on top of the OCaml sys-

tem [13, 20]. Shortly, the JoCaml runtime system is a minimal extension of the thread systems of OCaml— There are three different thread libraries in OCaml. It is mostly implemented by calls to a specific library referred to as “J” in the following. Library J is written in OCaml. It relies upon the essential features of ML, such as full functionality and pattern matching, and upon libraries. Those libraries are both user-level ones, such as the thread library, and system-level ones, such as the `Obj` library that handles runtime values in an untyped manner.

Apart from showing that our design is of practical significance, our prototype aims at demonstrating how object semantics and class operations can be integrated into the JoCaml system. Writing the prototype gave us some insight on how we can perform such an extension of JoCaml. In the following, we present those ideas. We pay significant attention to issues such as static typing and separate compilation. As a matter of fact, JoCaml (like OCaml) is a multi-module system that features separate compilation of implementations and interfaces, and a fully static type system. More concretely, separate compilation implies that not all the class definitions are available at compile time. Therefore, at least some of the class operations have to be performed at runtime. Types are totally absent at runtime, this means that compiled code has no access of any kind to type information. However the code may assume some properties, since it results from the compilation of well-typed source. This context should be kept in mind while considering our design.

7.1 Type inference

Type checking takes place at the early stage of compiling. The type system of $OJoin_{\mathcal{H}}$ has been carefully designed to allow type inference. We claim that, given a typing environment A and a process P (or a class C), it is decidable whether P (or C) is typable in A . Moreover, we claim that if C is typable then it has a principal type. In the prototyping system, the type inference has been implemented on top of using in-place graph unification. Additionally, we adopt some of the efficient techniques from [30].

7.2 Compilation of message sending

From a runtime perspective, objects are very similar to join-definitions. In JoCaml [20] join-definitions are essentially implemented as data structures that, amongst other things, hold an array of message queues. Abstractly, channels are pairs of such a data structure and of a *slot number*. Channels defined conjointly (in the same join-definition) have the same first component, and the slot number is an index into the sub-structures of the join-definition — for instance, into message queues. In practice, the only operation performed on channels is sending messages on them and, for that reason, channels are compiled into functions⁴:

```
let d = ... in (* join-definition *)
let c = (fun arg → J.send d i_c arg) (* channel *)
```

Sending a message m on channel c amounts to applying function c to m . The library function `J.send` performs the actual sending of m to the channel whose slot number in definition d is i_c . Observe that the variable d occurs free in the function above. Library function `J.send` will first record message m into the appropriate queue (the one whose index is i_c), then depending on the current status of message queues, it may fire the appropriate guarded process, or return immediately. Notice that all operations on the join-definition data structures (such

⁴We present the compiled code in OCaml syntax. In the actual JoCaml system, such code is “lambda-code”, *i.e.* loosely typed enriched λ -calculus.

as queues) are protected by a mutual exclusion lock (in short a mutex), which is part of the join-definition data structure. One of the first step of `J.send` consists in locking this *object mutex*.

We will not elaborate on the implementation of `J.send`, that is, on how the join-pattern matching can be performed efficiently by using the automata of [20]. Instead, we wish to stress on the important point that, in JoCaml, the slot number i_c is computed at compile time, since the definition is known completely at compile time, with all its join-patterns. By contrast, in the object-oriented extension, the slot numbers result from class operations and can no longer be known by the compiler. Thus a new compilation strategy is called for.

As regards message sending, we make a distinction between internal message sending (*i.e.* through explicit **this**, as in `this.c(m)`), and external message sending (*i.e.* through a variable, as in `o.c(m)`). In this section, we treat external message sending. Every object `o` holds a field referring to its class that, amongst other things, includes a mapping from labels to slot numbers. We will abstract on the details of this mapping and represent it as another library function `J.get_slot` that takes an object and a label as arguments. And we compile `o.c(m)` by using `J.get_slot` as:

```
let slot = J.get_slot o "c" in
J.send o slot m
```

Basically, when compared with message sending in join, there are two changes:

- In plain join, the binding `d` for join-definition is a compiler convenience that expresses the sharing of a data structure by all channels. By contrast, in the object-oriented extension, the join-definitions (*i.e.* the objects) are first class values, and bindings for them originate from source.
- The mapping from labels to slot numbers must now be performed at runtime, at least in the case of external message sending. The efficiency penalty is not as high as it may seem at first. Faced with a similar problem, the developers of OCaml tested various techniques, which achieve reasonable efficiency. To some extent, this efficiency comes from replacing string comparisons by integer comparison. Labels get translated into integers by a first, global, injective mapping, which is performed once at runtime [35, Chapter 4] by means of a hash table, or even at compile time by using compile-time hashing [16]. A second, per-class, mapping then translates those integers into slot numbers. Some unpublished experiments seems to show that binary search is a reasonable implementation of this second mapping. As usual, caching at call sites, called in-line caching in [8], is also an option.

By contrast, retrieving slot numbers for internal message sending can be done more efficiently, by one memory indirection. We explain the idea in the next section along with our discussion about the compilation of class definitions.

7.3 The class layer

7.3.1 Basic classes

A basic class, *i.e.* one whose source trivially yields a class already in normal form, can be seen as a join-definition with an optional set of abstract labels and an optional initializer. The simplest form of a basic class is a single reaction rule. However, even the compilation of basic classes cannot produce the automata of [20] immediately. The reason is that class definitions

are open to class operations, while automata are too low-level data structures to allow such operations. Instead, the runtime value of class definitions in normal form is some structure which attempts a good balance between abstraction (for efficiency) and expressiveness (to allow class operations). More specifically, on the one hand, to carry out class disjunction and selective refinement, we require the join-patterns to be kept concrete in the runtime data structure for classes. On the other hand, guarded processes have to be compiled into functions now, for the sake of separate compilation. However, because of the possibility of a later hiding of some private labels — that causes α -conversion of those labels, the code of guarded processes has to be abstracted w.r.t. private labels. Moreover, additional implementation concerns lead us into considering abstraction w.r.t. all labels (including public ones) and channel formal arguments. Based on these considerations, we compile reaction rules as a list of records. Those records possess three fields: field “**jpt**” is the join-pattern (which we write in source syntax and enclose in single quotes, see below), field “**tbl**” is an array of labels and channel formal arguments (or received variables), and field “**clo**” is the guarded process compiled as a function (a closure in runtime terms). We illustrate our ideas by the following example:

```
class c =
  a(x) ▷ this.A(x)
  or A(y) ▷ out.print_int(y)
```

We compile the reaction rules of class **c** as the following list of two records:

```
1 let rules =
2   [{jpt = 'a(x)';
3     tbl = [| 'x'; 'A' |];
4     clo = (fun dict this →
5             let x = J.get_queue this dict.(0) in
6             J.unlock this;
7             J.spawn (fun () → J.send this dict.(1) x))}];
8   {jpt = 'A(y)';
9     tbl = [| 'y' |];
10    clo = (fun dict this →
11           let y = J.get_queue this dict.(0) in
12           J.unlock this;
13           J.spawn (fun () →
14                   let slot = J.get_slot out "print_int" in
15                   J.send out slot y))}]
```

Functions “**clo**” have an uniform interface so that they can be called easily by generic code. They take two arguments, the dictionary **dict**, and **this**, a binding for self. A dictionary is an array of slot numbers that exists at runtime, of which the array “**tbl**” is the representation at the class level. While dictionaries associate indices to slot numbers, “**tbl**” arrays associate indices to formal arguments and labels, which we write in single quotes, as ‘**x**’, ‘**A**’ etc. so as to avoid confusion with program variables. formal arguments are here for the purpose of fetching arguments from message queues, while labels are for performing internal message sending. The OCaml notation for the array *t* of which elements are $x_0, x_1, \dots, x_n$ is [| x_0 ; x_1 ; ...; x_n |] and array access is *t*.(*i*). Internal message sending are performed through dictionaries. For instance here, the first guarded process performs an internal message sending **this.A(x)**, In the corresponding compiled code (line 7), the slot number of channel **A** is

retrieved as the second element of dictionary `dict`. Such compiled code that directly originates from the source of guarded processes is collected into functions that take argument “()” (void of type `unit`). Those functions are used as argument to the library function `J.spawn` of type $(\text{unit} \rightarrow \text{unit}) \rightarrow \text{unit}$. When applied to function f , `J.spawn` creates a new thread to execute f applied to void. Before guarded processes are spawned, two tasks are performed: first, the values of guarded process arguments (x and y above) are consumed from the appropriate message queues (by library function `J.get_queue` at lines 5 and 11); second, the object mutex is released (by library function `J.unlock` at lines 6 and 12). In other words, closures “`clo`” are responsible for releasing the mutex taken by the library call `J.send` that fired them. Most of the compilation of guarded processes is the one of the JoCaml system. The key addition is the extra dictionary argument.

Basic classes are compiled into another kind of records with a first field “`rules`” holding reaction rules, a second field “`hidden`” keeping the sets of hidden labels. Additionally, we compile the initializer process as a special reaction rule `init()` $\triangleright P$. However, we omit the trivial join-pattern `init()` and only keep the remaining two fields of the reaction rule: “`tbl`” and “`clo`” in the third “`init`” field of the class structure. In case there is no initializer (class initializer are optional), the “`init`” field holds a distinguished constant `null`. To sum up, the class `c` above is compiled into the following record:

```
let c =
  {rules = rules;
   hidden =  $\emptyset$ ; init = null}
```

Note that, in our example, the last two fields are empty. Without ambiguity, we refer to this data structure as a *class value*. It may be surprising that we do not record abstract labels in class values, in spite of the fact that class normal forms actually comprise abstract labels (as L in $(D \text{ or } L) \text{ init } P_v$). As a matter of fact, L parts in class normal forms are there only to express errors: an attempt to instantiate a class whose labels are not all concrete, or an attempt to hide abstract names (rules EVAL-OBJECT-ERROR and EVAL-HIDE-ERROR2 in Section 4.3). While all those errors cannot occur at runtime because of static typing, hence no check is performed at runtime. As a consequence, there is no need to keep abstract labels in class values. This does not mean that abstract labels are totally absent from class values: some of the labels in “`tbl`” fields may in fact be abstract, when guarded processes try to send messages on them.

7.3.2 Class initializers

Class initializers are simplified reaction rules in two aspects. First there is no join-pattern. Second, guarded process compilation is simplified, because such guarded processes take no argument. As a consequence, they do not modify the object data structures directly, and no part of them need to execute in critical section. More specifically, initializer P is compiled into the function:

```
(fun dict this  $\rightarrow$   $\llbracket P \rrbracket$ )
```

where $\llbracket P \rrbracket$ is the informal notation for “compilation of process P ”. By contrast, the general compilation of guarded processes (as shown in the previous subsection) has to introduce calls to `J.unlock` and `J.spawn`. Nevertheless, initializers must execute in their own thread. Later, at the creation time of an object of some class c , the runtime support will spawn a new thread to call function `c.init.clo` with appropriate dictionary and self arguments.

7.3.3 Hiding and class variable reference

Hiding is implemented by giving the ‘hidden’ field some non-empty value. Freshening of hidden labels is performed by the library function `J.freshen` at the time when the classes are referred to by their names (cf. rule EVAL-CNAME in Figure 5). Notice that, since we maintain an explicit set of hidden labels, the initial freshening of hidden labels performed by rule EVAL-HIDE is not necessary. As an example, we slightly alter our previous example of a class `c`, so that label `A` is hidden.

```
class c' =
  a(x) ▷ this.A(x)
  or A(y) ▷ out.print_int(y)
  hide {A}
```

```
... c' ...
```

Then, compilation simply yields:

```
let c' = {rules=rules; hidden={'A'}; init=null}
```

```
...J.freshen c'...
```

The library function `J.freshen` performs the freshening of the hidden labels in the class value given as argument. The operation of freshening is defined in Figure 6. In practice, `J.freshen` will first build a mapping from hidden labels to fresh names, or return immediately if the set of hidden labels is empty. Notice that it is very easy to get fresh names in an implementation, by means of a global counter. Actual freshening is not performed over guarded processes as described in Figure 6, but rather on the “tbl” field of reaction rules. Here, assuming that label `A` is mapped to the fresh label `A'`, the class value resulting from freshening, will be:

```
{rules =
  [{jpt='a(x)'; tbl = [| 'x'; 'A' |];
    clo = (fun dict this → ...)};
  {jpt = 'A'(y)'; tbl = [| 'y' |];
    clo = (fun dict this → ...)}];
  hidden={'A'}; init=null}
```

7.3.4 Class disjunction

In a separate compilation setting, class operations may apply to some free class names that are defined in another module and whose normal forms will thus not be accessible until runtime. As a consequence, performing complex class operations at compilation time is impossible. Instead, we compile class operations as code which, when executed at runtime, will compute the resulting class values. In practice, because of the complexity of class operations, most of class operations will be performed by calling generic library functions. We do not believe that inlining and specializing those calls is a good idea. Instead, we think this would probably yield little benefit as regards execution time, but cause a significant penalty as regards code size. In our view of application programs, class operations are less frequent than object layer operations (*i.e.* object creation and message sending). Hence we more focus on producing efficient code for objects.

Class disjunction aims at combining two compiled class values, essentially by accumulation of reaction rules. Following the evaluation rule EVAL-DISJUNCTION, we mostly perform

list concatenation (written “@”) of the “rule” fields of the argument classes. Compilation transforms class disjunctions into calls to the library function `J.disjunct`, which takes two class values as arguments.

```
let disjunct c1 c2 =
  { rules=c1.rules@c2.rules;
    hidden=Set.union c1.hidden c2.hidden;
    init=conjunct_init c1.init c2.init; }
```

At runtime, variables `c1` and `c2` are bound to the class values that we now operate on. Function `Set.union` refers to standard library function for sets, with obvious semantics.

Initializers require a slightly more complicated treatment during disjunction. The computation of new initializers is encapsulated in another library function `J.conjunct_init`.

```
let conjunct_init i1 i2 =
  if i1 == null then i2
  else if i2 == null then i1
  else
    let n1 = Array.length i1.tbl
    and n2 = Array.length i2.tbl in
    { tbl=Array.append i1.tbl i2.tbl;
      clo=(fun dict this →
        spawn (fun () → i1.clo dict this) ;
        i2.clo (Array.sub dict n1 n2) this); }
```

Function `conjunct_init` takes two initializers as arguments, which are the compilation output of `init` processes P_1 and P_2 , and returns a new initializer that should act as process $P_1 \& P_2$. Following the JoCaml system, $P_1 \& P_2$ is compiled as

```
J.spawn (fun () → [[P1]] ; [[P2]])
```

That is, the parallel composition “&” gets translated into explicit thread creation and sequence. As regards `conjunct_init`, in case either initializer is empty, then the result is the other initializer. Otherwise, both initializers are non-empty, and the dictionary of the resulting initializer is the array built by appending dictionaries, as reflected by the new “tbl” field. The new “clo” function simply calls both initializers with appropriate dictionaries (`Array.sub t p ℓ` computes the subarray of array t whose length is ℓ , starting at position p). It is worth noticing that such a composition of initializers is possible because initializers do not use the object mutex. Would it be the case, the above code would release the object mutex twice. Such a simple composition will not be adequate for composing guarded processes during selective refinement, which we examine now.

7.3.5 Selective refinement

Selective refinement is by far the most sophisticated amongst class operations. In fact, it is a real challenge in compiler design. We illustrate our ideas by means of an example. We first consider the example of one refinement clause.

```
...
| a(z) ⇒ a(y) & b(z) ▷ this.B(z+y)
...
```

Observe that the clause above operates a change in formal arguments. Before the refinement, variable z refers to the message sent on label a , while, after refinement, variable z is the formal argument of label b . Furthermore, a new formal argument y is introduced.

We first review how reaction rules are rewritten by matching clauses (rule FILTER-APPLY of Figure 5). The general form of a refinement clause is $K_1 \Rightarrow K_2 \triangleright Q$, and a matched reaction rule is of the form $M \triangleright P$, with $M \equiv K_1 \& K$, where the congruence “ $\equiv$ ” on join-patterns means structural equality up to the commutativity and associativity of the parallel composition “ $\&$ ”, and α -conversion of formal arguments. Then, the resulting rule is $K_2 \& K \triangleright Q \& P$. In practice, the commutativity and associativity of parallel composition are not very annoying: all involved join-patterns can be represented as sorted lists of message patterns. By contrast, α -conversion deserves attention. For instance, we assume some matched reaction rule $'a(x)' \& K \triangleright P$, then the rewritten rule with explicit formal arguments shown is $'a(y) \& b(z)' \& K \triangleright Q \& (P\{'z'/'x'\})$. That is, we substitute $'z'$ for $'x'$ in process P . Additionally α -conversion should guarantee that no name clash of formal arguments occurs, that is, neither $'z'$ nor $'y'$ are formal arguments in K . In the following, for the sake of simplicity, we shall assume that this last condition holds. It can be implemented by a prior freshening of all formal arguments in refinement clauses, whenever rewriting is performed.

Rewriting of matched reaction rules will be performed by code compiled from clauses. More specifically, given a clause, the compiler produces a “*class builder*” function. Here is for instance the class builder resulting from the compilation of the clause above:

```

1  let builder rule =
2    let n = Array.length rule.tbl in
3    let new_rule =
4      { jpt=J.refine_jpt ['a'] 'a(y) & b(z)' rule.jpt;
5        tbl=
6          Array.append
7            (J.subst_tbl
8              [('z', J.get_formal rule.jpt 'a'])
9              rule.tbl)
10         [| 'z'; 'y'; 'B' |];
11        clo=(fun dict this →
12          let z = J.peek_queue this dict.(n+0)
13            and y = J.get_queue this dict.(n+1) in
14          J.spawn (fun () → J.send this dict.(n+2) (z+y)) ;
15          rule.clo dict this) } in
16    {rules=[new_rule] ;
17      hidden=∅; init=null;}

```

The new join-pattern is built by some external function `J.refine_jpt` (line 4). Function `J.refine_jpt` takes a list of labels ($dl[K_1]$), a join pattern (K_2) and another pattern (M , the join-pattern of the matched reaction rule) as arguments. It returns a new join-pattern M' , which is M with all message patterns whose label is in the list removed, and with K_2 added. The new dictionary is built by appending some additional slots at the end of the α -converted dictionary of the original reaction rule (lines 6–10). The performed α -conversion is the one we wrote as $P\{'z'/'x'\}$ in the previous paragraph. However, we cannot change anything in the “`clo`” function resulting from the compilation of P . Thus, α -conversion operates on dictionaries, using the library function `J.subst_tbl`. The formal argument of label $'a'$ in M

(x in our example) is extracted from M by another external function `J.get_formal`. One should remark that the existence of such α -conversions of formal arguments explains why we introduce formal arguments in dictionaries.

We now describe the new “`clo`” function, which should act as $Q \& P$ (lines 11–15). According to the structure of the new dictionary, process Q (which is `this.B(z+y)`) is compiled with an explicit offset `n` over dictionary accesses (lines 12–14). The exact value of `n` will be known at runtime, as the dictionary size of the matched rule (line 2). It is worth noticing that the values of the formal arguments of K_2 (`'z'` and `'y'`) are fetched in different ways depending upon whether they occur in both K_1 and K_2 , or in K_2 only. In the former case, (formal argument `'z'`, line 12) we use a new library function `J.peek_queue`, while, in the latter case (formal argument `'y'`, line 13), we use the normal `J.get_queue`. Function `J.peek_queue` returns the value that stands first in the queue, without removing it from the queue. By contrast, `J.get_queue` removes the value from the queue. We do so because the value of argument `'z'` is consumed from the appropriate queue by function `rule.clo`, while function `rule.clo` knows nothing about argument `'y'`. The closure of the matched reaction rule (*i.e.* `rule.clo`) is simply called as the last operation of the new “`clo`” function (line 15). One should remember that the object mutex of `this` is released by `rule.clo`. The mutexes of OCaml have the following behavior: when a thread attempt to lock a mutex owned by another thread, then this thread suspends, until re-activated when the current owner of the mutex releases the mutex. As a consequence, the call to `J.spawn` at line 14 is enough to prevent deadlock.

Application of refinement clauses is performed by the library function `J.filter`. This function takes two arguments. The first argument is a list of pairs, of which first components are the selective patterns K_1 and second components are the class builders compiled from clauses $K_1 \Rightarrow K_2 \triangleright Q$. The second argument is a reaction rule. The task of `J.filter` is to identify the matching clause and to return the rewritten reaction rule:

```
let rec filter clauses rule = match clauses with
| [] → (* cf. rule FILTER-END *)
  { rules = [rule] ; hidden=0; init=null; }
| (K1, builder)::rem →
  if Set.subset (dl_jpt K1) (dl_jpt rule.jpat) then
    builder rule
  else (* cf. rule FILTER-NEXT *)
    filter rem rule
```

The function above makes use of OCaml pattern matching on lists of pairs to find the first clause that matches reaction rule `rule` in the sense of Section 4.2 (subsection **Filter evaluation**). The matching condition is that $dl[K_1]$ is included in the labels defined by the rule pattern. We here make use of the trivial external function `J.dl_pat` to compute labels defined by a join-patterns (see Figure 4 for the definition of $dl[\cdot]$ over join-patterns). If no matching clause is found, the argument reaction rule `rule` is returned, packed as a simple class value.

Finally, selective refinement is performed by a call to another library function `J.refine`. Function `J.refine` takes a list of compiled clauses and a class value as arguments. It returns the new class value resulting from selective refinement.

```
let refine clauses c =
  let cs = List.map (filter clauses) c.rules in
  let r = disjuncts cs in
```

```

{ rules=r.rules;
  hidden=c.hidden;
  init=c.init; }

```

The code above applies `J.filter` to all the reaction rules of the argument class `c`, yielding a list of rewritten classes. The library function `List.map`, when given a function f and a list $[x_0; x_1; \dots; x_n]$ as arguments, returns the new list $[f\ x_0; f\ x_1; \dots; f\ x_n]$. Function `J.refine` then combines the results of filtering with class disjunction (cf. rule `FILTER-OR` in Figure 5). Function `J.disjuncts` is a simple extension of the previously introduced function `J.disjunct`. Function `J.disjuncts` takes a list of class values as argument, while `J.disjunct` operated on two class values only. (cf. previous section **Class disjunction**). Note that because selective refinement has no influence to hidden names and initializers, the `hidden` field and the `init` field remain the same.

7.4 The object layer

7.4.1 Classes in objects

Class values turned out to be adequate for class operations. However, they are not adequate to be used as the class field in objects. First, for code to execute, the arrays of formal arguments and labels in the “`tbl`” field have to be replaced by arrays of slot numbers. Second, keeping the concrete join-patterns in the “`jpt`” field would yield inefficient computations for deciding whether to fire a guarded process or not when receiving messages.

Objects will thus hold a pointer to a different kind of class value, a *run class value*. Computing run class values from class values is conceptually simple. One should first collect all defined labels and allocate slot numbers for them as successive integers, starting from zero. This yields a mapping σ from labels to slot numbers. Then, a more low level representation of reaction rules is computed. For each reaction rule, we extend σ with mappings from formal arguments to slot numbers in the sense that: given pattern $l(u)$ we add the the binding $u \mapsto \sigma(l)$. This augmented mapping is then applied to the “`tbl`” field of the reaction rule, yielding the final dictionary. Join-patterns are translated into bitfields that represent sets of slot numbers, a compact representation for sets of labels. Closures in the “`clo`” fields are of course left unchanged. The same transformation from labels to slot numbers is performed on the initializer, when present. Finally, run class values need an extra “`slots`” field that holds the mapping from public labels to their slot numbers.

As an exemple, we consider the class `c` of Section 7.3.1 and the selective refinement of Section 7.3.5:

```

class c =
  a(x) ▷ this.A(x)
  or A(y) ▷ out.print_int(y)

class d =
  match c with
  | a(z) ⇒ a(y) & b(z) ▷ this.B(z+y)
  end
  or B(x) ▷ out.print_int(x)

```

Following the rules of Figure 5, class evaluation yields the class normal form:

```

a(y) & b(z) ▷ this.B(z+y) & this.A(z)

```

```

or A(y) ▷ out.print_int(y)
or B(x) ▷ out.print_int(x)

```

And at runtime, the corresponding class value is:

```

{ rules=[
  { jpt='a(y) & b(z)';
    tbl=[| 'z'; 'A'; 'z'; 'y'; 'B' |] ;
    clo=... } ;
  { jpt='A(y)'; tbl=[| 'y' |] ; clo=... } ;
  { jpt='B(z)'; tbl=[| 'z' |] ; clo=... } ] ;
  hidden=∅; init=null }

```

We assume the slot allocation is 'a' $\mapsto$ 0, 'b' $\mapsto$ 1, 'A' $\mapsto$ 2, 'B' $\mapsto$ 3. Then the resulting run class value is:

```

{ rules=[
  { jpt=0B0011; tbl=[| 1; ; 2; 1; 0; 3 |] ;
    clo=... } ;
  { jpt=0B0100; tbl=[| 2 |] ; clo=... } ;
  { jpt=0B1000; tbl=[| 3 |] ; clo=... } ] ;
  init=null;
  slots=[("a", 0); ("b", 1)]; }

```

In the code above, bitfields are written as integers in binary notation. The mapping from public labels to slot numbers is represented as a list of pairs of strings and integers. However, more efficient implementations are possible (see the end of Section 7.2 on external message sending). Without further detail, we state that the computation of run class values from class values is performed by the library function `J.create_runclass`.

7.4.2 Object instantiation

The computation of run class values is performed at object instantiation time. In order to perform this computation at most once, run class values are cached in a new, yet unshown, field "runclass" of the class value structure. We assume a default value of `null` for the field "runclass". More concretely, the creation of objects is performed by the library function `J.create_instance`, which takes a class value as argument.

```

let create_instance c =
  if c.run_class == null then
    c.runclass ← create_runclass c ;
  let runclass = c.runclass in
  let o =
    { status=0 ;
      mutex=Mutex.create () ;
      queues=... ;
      class=runclass } in
  let i = runclass.init in
  if i != null then
    spawn (fun () → i.clo i.tbl o) ;
  o

```

First, if necessary, the run class value is computed. Then the object value, shown here as another kind of record, is computed. The record for objects holds four fields: field “**status**” expresses the current status of message queues as a bitfield; fields “**mutex**” holds the object mutex; field “**queues**” holds an array of message queues; and field “**class**” holds the run class value of the object. This last field is shared by all objects created from the same class. Before returning the created object, the initializer process is spawned, when present.

We now have produced the automata of the JoCaml system [20]. They are represented by the bitfields in the “**jpt**” fields of run class values and the “**status**” field of objects values. We interpret those as sets of slot numbers. In the case of the “**jpt**” fields, such sets simply encode join-patterns in a compact way. By contrast, the “**status**” field evolves over time and expresses the current status of message queues: the queue of slot i_c is empty if and only if the value of bit number i_c in the “**status**” field is zero. The behaviour of objects is abstracted by automata. The states of those automata are possible values of the “**status**” field, while the transitions of them express **status** evolving upon message receiving. During transitions, some actions may be performed.

As a concrete example of a join-matching automaton, we show the one of an object of the previous class **d**, as the diagram in Figure 11. The mapping from labels to slot numbers is recalled at the bottom of the figure.

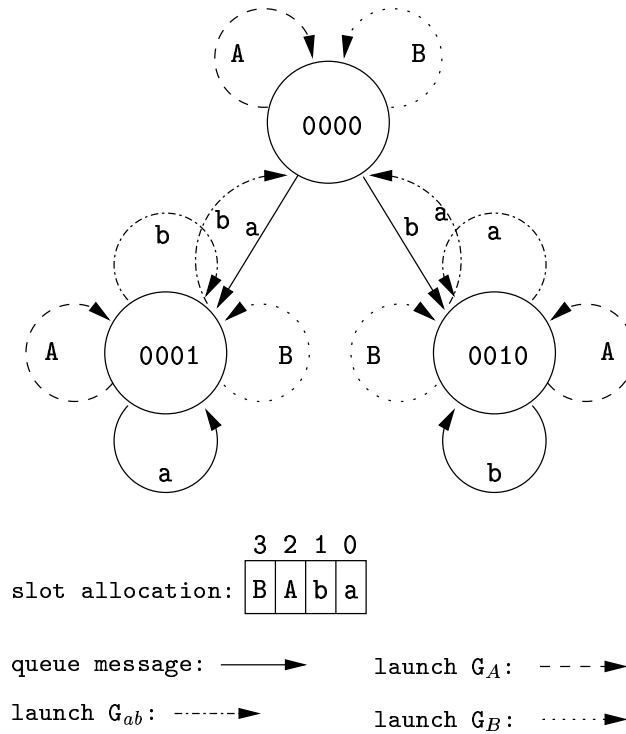


Figure 11: The automaton of objects of class **d**

Observe that there are only three states present in the automaton but not 2^4 . This is due to the policy of firing guarded processes as soon as possible, which in practice removes all the matching **status** from the automata. Transitions are decorated by channel names, meant to pointing out messages received on which channel actually cause the transitions. Four different

kinds of arrows denote four different kinds of actions taking place along with transitions. The action attached to a solid arrow is simply to store the message in an appropriate queue, while the other three kinds of arrows express the firing of either of the three possible guarded processes as shown in Figure 11. We denote the guarded process of the first reaction rule as G_{ab} , the second as G_A and the last as G_B . As an example of automata behavior, consider the situation when all queues are empty (state 0000). Messages received on either label A or B will immediately trigger either G_A or G_B , and the object status does not change. By contrast, messages on either label a or b will result in queueing the message and changing the state to 0001 or 0010 respectively. Note that the non-determinism from the state 0001 upon message receiving on channel b is a “faked” one. Although it seems to have two possible transitions, the behavior will always be to trigger guard G_{ab} , and the resulting state is decided deterministically according whether some messages are still present in the queue of channel a. Similarly, the two possible transitions starting from state 0010 upon message receiving on channel a do not introduce non-determinism, either.

7.4.3 Relations between class and object layers

In our semantical model, which we use for proving type soundness, there is a clear separation between the evaluation of classes and objects. Classes are first rewritten to class normal forms (Section 4), yielding valid input for the object chemical machine (Section 2.3.2). It is obvious that the simple model of chemical semantics does not account for the functional part of our implementation (*i.e.* ML). Nor does it account for more basic computations, such as computing on numbers. Although models that combine concurrency and functional programming [4] or concurrency and ordinary values [1] exist, we do not attempt such a combination. Instead, we think that our relatively simple framework is sufficient to guide a sound implementation, given that functional and concurrent aspects do not interfere much.

However, a noticeable discrepancy exists between our model and our proposed implementation. In the semantical model, the class layer and the object layer are clearly separated, while, in the implementation, the execution of class and object operations mix. The discrepancy is particularly salient when class operations occur inside guarded processes. Consider the following example, where C is a class definition and P a process:

```
class c =
  a(x) ▷ class d = b(y) ▷ this.A(x+y) or C in P
```

In the semantical model, class d is reduced to class normal form once, prior to chemical evaluation; whereas, in the implementation, class d is computed whenever a message is sent on label a. The model behavior stems from rule EVAL-REACTION in Figure 5. The implementation behavior stems from the compilation of class operations in guarded process: the process guarded by pattern $a(x)$ is compiled into a function f that performs class operations and class d will be rebuilt whenever function f is called.

In fact, we can explain the discrepancy between model and implementation more precisely. In the model, object variables can occur free in class normal forms, while in the implementation, class values, like all values, resemble ground terms. A clean solution is to restrict class operations and to enforce that they occur at toplevel only. In OCaml, toplevel bindings are computed once only, and this solution accords with semantics better. But then, to recover expressiveness, we should allow class definition parameterized by ordinary (*i.e.* non-class) variables.

```
class d(x) = b(y) ▷ this.A(x+y) or C
```

class $c = a(x) \triangleright P\{d(x)/d\}$

We could even have the compiler to change such “inner classes” into toplevel classes, so as to allow inner classes in users programs. However, we have no theory of parameterized classes, nor have attempted to implement them. We expect difficulties, especially as regards typing. Nevertheless, it is worth noticing that the bindings of OCaml are immutable, that is, variables cannot be assigned. As a consequence, we do not face the same difficulties with free variables in inner classes as Java does.

8 Related work

The idea of using join-patterns for class synchronization abstraction in object-oriented programming is also followed by other language design, such as polyphonic C[#] [2] and Join Java [18]. However, classes in those only support limited inheritance of Join abstractions. Fournet *et al.* study this problem based on a theoretical foundation in [15]. They extend the Join calculus with a class language, in which various operations are designed to support a variety of inheritance paradigms. Our previous work [22] improves their model by proposing a more expressive type system. This paper introduces further improvement from another angle. We enrich Fournet *et al.*’s calculus with information hiding. To draw a comparison, the model presented in this paper on one hand allows more precise and flexible visibility control of classes than in [15], on the other hand allows more degree of type abstraction than in [22].

Our hiding mechanism is inspired by the design of its counterpart for sequential classes in OCaml, which is not described in its theoretical calculus [31] but is present in its real system. Briefly, in the sequential case, hiding amounts to freezing method names, while our extension additionally performs a similar action on synchronizations policies defined by a class. From typing point of view, hiding method name in OCaml also amounts to removing the hidden names from class types. However, hiding in OCaml (and in MOBY [10]) is performed implicitly by specifying restricted class types. Given the sophisticated class types of OJoin_H, such an option would not be convenient for concurrent classes. In particular, it seems impractical to deprive programmers from compiler help in figuring out the impact of hiding on synchronization and, above all, polymorphism. Thus, in contrast to OCaml, we provide an explicit class operator for hiding, and the type of the resulting class is inferred automatically.

Fisher and Reppy design a ML style module system to take care of the visibility control of classes in MOBY [10, 9]. One significant difference between MOBY and our design is in the hiding of public members of a class. Such a difference originates in the problems between hiding public names and supporting advanced features, such as *selftype* (also known as *mytype*) and *binary methods* [5]. As observed by Rémy and Vouillon [31, 36], and also by Fisher and Reppy [10], these two aspects do not trivially get along without endangering type soundness. A simple solution to these problems is to support either. We choose to support the notion of selftype while limit hiding to private channels, as is the case with OCaml. By contrast, Fisher and Reppy choose complete visibility control over selftype in MOBY. Notice however that Vouillon proposes a comprehensive solution [36].

More specifically, problem manifests itself when selftype is assumed outside the class and we hide a public name afterwards. As an example, consider the following class definitions:

class $c_0 = f(x) \triangleright x.b(1)$

```

class c1 =
  a() ▷ obj x = c0 in x.f(this)
  or b(n) ▷ out.print_int(n)

```

Label **f** of class **c₀** expects an object with a label **b** of type **int**. This condition is satisfied when typing the guarded process of label **a** in class **c₁**, because the self object **this** does have a label **b** of type **int**. However, later inheritance may hide the label **b** (in class **c₂**), and then define a new label also named **b** but with a different type **string** (in class **c₃**).

```

class c2 = c1 hide {b}
class c3 =
  c2
  or b(s) ▷ out.print_string(s)

```

Apparently, although the above code is type correctly, the following process will cause a runtime type error: providing an integer when a string is expected.

```
obj o = c3 in o.a()
```

Furthermore, another kind of problem may happen when selftype appearing as the argument type of some labels, *i.e.* binary methods, as in the following example:

```

class c0 =
  a(x) ▷ x.m()
  or b() ▷ this.a(this)
  or m() ▷ ...

```

Because of the second reaction rule, the argument of label **a** is of selftype, and moreover, with a label **m**, as expected by the guarded process of label **a**. If we allow to hide the public name **m** during inheritance as in:

```
class c1 = c0 hide {m}
```

the selftype changes to without label **m** *a posteriori*. As a consequence, messages can be sent to label **a** with an object that may not have a **m** label. This is clearly unsound.

A simple solution to these problems is to support either. We choose to support the notion of selftype while limit hiding to private channels, as is the case with OCaml. By contrast, Fisher and Reppy choose complete visibility control over selftype in MOBY. However, Vouillon proposes a comprehensive solution by making use of *views* [36]. All those works [10, 9, 36] use Riecke-Stone style *dictionaries* [32] to capture the dynamic semantics of hiding. It is worth noticing that dictionaries, which basically are bindings from labels to labels, appear explicitly in the cited calculi, but also in the language of [36], thereby adding significant complexity. Compared with their approach, our semantics of hiding by α -conversion is simpler, and suffices to the purpose of hiding private names only. Note that we also employ dictionaries, but only at the implementation level for efficiency.

Another way of hiding is by subtyping, structural or inheritance-based, at object level. Namely, an object of sub-type can be used as an object of super-type. This kind of hiding is also referenced as *coercion* (or *subsumption*). However, as we address parametric polymorphism rather than subtyping polymorphism in our calculus, this kind of hiding is not discussed in this paper.

9 Conclusion and future work

We extended the hiding mechanism from sequential to concurrent object-oriented settings. Along with the privacy mechanism, the hiding mechanism provides a flexible way to control

class accessibility, both at object level and class level. We designed the hiding mechanism as yet another operation on classes. The semantics is formally defined by α -converting hidden names to fresh names, which exploits the usage of the keyword **this**. We believe our semantics of hiding could also be easily adapted to formalize the corresponding mechanism in OCaml, and thus could be applied to the theoretical model of OCaml [31].

We also designed a type system in the tradition of ML to accompany the hiding operation. Hiding has been achieved by eliminating the hidden names from class types. However, wild elimination endangers safety. More precisely, the eliminated class types can only express partially the set of free type variables shared by correlated labels, thus falls short in manifesting the impact of synchronization on polymorphism [22, 14, 6] to the full extent. This deficiency is critical to type safety. As a solution, we equipped class types with a set of “dangerous variables” to recover some of the polymorphism impaired by hiding. We claim that types for hidden classes can be automatically inferred. Although lacking a formal treatment, we prototyped a type inferer to demonstrate the idea.

We proved the soundness of our type system, both at the class evaluation level and the chemical machine level. The use of big-step semantics with evaluation environment demands changes to the proof techniques of [15, 22]. More specifically, additional typing rules are needed to type evaluation environments polymorphically. Moreover, the different effects of hiding to the semantics (*i.e.* α -converting the hidden names) and to the type system (*i.e.* eliminating the hidden names) causes further impact to the soundness proof. An extra typing rule called TYPE-REVEAL was actually called for to bridge the gap, where revealing intuitively reflects the reverse of hiding. Besides the standard soundness property, we informally argued that our mechanism deserves the name “hiding”.

We have achieved significant improvements over the original of Fournet *et al.* [15], that is, [22] as regards the class system expressiveness and this paper as regards visibility control and simplification of runtime semantics. We claim that those improvements yield a calculus mature enough to act as the model of a full-scale implementation, which we plan as the integration of our class-based design into the JoCaml system [13]. We have not yet performed this extension, but rather wrote a prototype implementation from which we draw some precise implementation guidelines. More precisely, we explain how to perform class operations in a separate compilation setting and how to implement message sendings to objects at a small additional price over ordinary message sendings in Join.

Future work. We believe our hiding mechanism can be easily adapted to module systems, where modules become the boundary of hiding. In analogy to OCaml, a module will consist of two parts: the implementation and the interface. Implementations are *.ml* files containing definitions, and interfaces are the corresponding *.mli* files. Then, to carry out hiding, we just list the hidden channels in the *.mli* file and let the type system infer the restricted class types and replace the hiding clause by the inferred types.

Hiding also makes it possible for several classes implemented differently to have a unified external interface. Extending the semantics of inheritance so that we can support a generalized version of selective refinement will be a challenge. Generalized selective refinement will apply to class types instead of class definitions, thereby acting more like an autonomous class-to-class operator.

References

- [1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *the Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001.
- [2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C[‡]. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.
- [3] P. D. Blasio and K. Fisher. A calculus for concurrent objects. In *the Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, LNCS 1119, pages 655–670, 1996.
- [4] G. Boudol. The π -calculus in direct style. In *the Proceedings of the 24rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 228–241, 1997.
- [5] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.
- [6] G. Chen, M. Odersky, C. Zenger, and M. Zenger. A functional view of join. Technical Report ACRC-99-016, University of South Australia, 1999.
- [7] S. Dal-Zilio. An interpretation of typed concurrent objects in the blue calculus. In *the Proceedings of the International Conference IFIP TCS 2000*, LNCS 1872, pages 409–424, 2000.
- [8] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *the Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'84)*, pages 297–302, 1984.
- [9] K. Fisher and J. Reppy. Foundations for Moby classes. Technical memorandum, Bell Labs, 1998.
- [10] K. Fisher and J. Reppy. The design of a class mechanism for MOBY. In *the Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 37–49, 1999.
- [11] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Nov. 1998.
- [12] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *the Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385, 1996.
- [13] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. The JoCaml system. Software and documentation available at <http://pauillac.inria.fr/jocaml>, 2001.
- [14] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ML for the join-calculus. In *the Proceedings of 8th International Conference on Concurrency Theory (CONCUR'97)*, LNCS 1243, pages 196–212, 1997.

- [15] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.
- [16] J. Garrigue. Programming with polymorphic variants. In *the Proceedings of ML workshop*, 1998.
- [17] A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *the Proceedings of 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, ENTCS 16(3), 1998.
- [18] G. S. Itzstein and D. Kearney. Join Java: An alternative concurrency semantics for Java. Technical Report ACRC-01-001, University of South Australia, 2001.
- [19] J. Kleist and D. Sangiorgi. Imperative objects as mobile processes. *Science of Computer Programming*, 44(3):293–342, 2002.
- [20] F. Le Fessant and L. Maranget. Compiling join-patterns. In *the Proceedings of 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, ENTCS 16(3), 1998.
- [21] Q. Ma. Prototyping on object oriented join calculus. Master's thesis, D.E.A. Programation: Sémantique, Preuves et Langages, Université Paris 7-Denis Diderot, Sept. 2001.
- [22] Q. Ma and L. Maranget. Expressive synchronization types for inheritance in the join calculus. In *the Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS'03)*, LNCS 2895, pages 20–36, 2003.
- [23] 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*, pages 107–150. MIT Press, 1993.
- [24] M. Merro, J. Kleist, and U. Nestmann. Mobile objects as mobile processes. *Information and Computation*, 177(2):195–241, 2002.
- [25] O. Nierstrasz. Towards an object calculus. In *the Proceedings of the ECOOP'91 Satellite Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 1–20, 1991.
- [26] M. Odersky. Functional nets. In *the Proceedings of the 9th European Symposium on Programming (ESOP'00)*, LNCS 1782, pages 1–25, 2000.
- [27] M. Papathomas. A unifying framework for process calculus semantics of concurrent object-oriented languages. In *the Proceedings of the ECOOP'91 Satellite Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 53–79, 1991.
- [28] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *the Proceedings of the International Workshop on Theory and Practice of Parallel Programming (TPPP 94)*, LNCS 907, pages 187–215, 1995.
- [29] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

- [30] D. Rémy. Extending ML type system with a sorted equational theory. Research Report RR-1766, INRIA-Rocquencourt, 1992.
- [31] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998.
- [32] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 172(1):2–28, 2002.
- [33] D. Sangiorgi. An interpretation of typed objects into typed π -calculus. *Information and Computation*, 143(1):34–73, 1998.
- [34] V. T. Vasconcelos. Typed concurrent objects. In *the Proceedings of ECOOP'94 Workshop on Object-Based Concurrent Computing*, LNCS 821, pages 100–117, 1994.
- [35] J. Vouillon. *Conception et réalisation d'une extension du langage ML avec des objets*. PhD thesis, Université Paris 7-Denis Diderot, Oct. 2000.
- [36] J. Vouillon. Combining subsumption and binary methods: an object calculus with views. In *the Proceedings of the 28rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 290–303, 2001.
- [37] D. J. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.

A Typing proofs

We present the detailed soundness proofs of the $\text{OJoin}_{\mathcal{H}}$ calculus in this appendix.

A.1 Basic properties

Let us first establish some auxiliary lemmas. Let Δ range over any right hand side of a typing judgment except for chemical typing judgements. (Because for chemical typing judgments, the type environments are always empty.) We have that the following lemmas hold.

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

$$A \vdash \Delta \iff A + A' \vdash \Delta$$

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

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

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

Definition 2 (Instance of type scheme). *We say that a type τ (τ^c) is “an instance of” of a type scheme $\forall X.\tau_0$ ($\forall X.\tau_0^c$), denoted by $\tau \preceq \forall X.\tau_0$ ($\tau^c \preceq \forall X.\tau_0^c$), if and only if there exists an substitution η whose domain is a subset of X and $\eta(\tau_0) = \tau$ ($\eta(\tau_0^c) = \tau^c$).*

Definition 3 (More general type scheme). *Given two type schemes, we say that one is “more general” than the other if and only if any instance of the latter is also an instance of the former.*

and we have the following lemma:

Lemma 4 (Generalization). *If $A \vdash \Delta$, if A' replaces some assumptions in A by more general ones, then $A' \vdash \Delta$.*

Proof. A formal proof can be derived by induction on the structure of Δ . Here we give some intuition. Let N denote the set of names bound in A (A'). If judgment $A \vdash \Delta$ holds, there must exist a typing tree which has this judgment as root. Then replacing A by A' , and $A(n)$ by $A'(n)$ for any name $n \in N$, in this tree, we get a new tree with root $A' \vdash \Delta$. To demonstrate that this new tree is also a valid typing tree, *i.e.* $A' \vdash \Delta$ holds, the only tricky point worth elaborating is in cases when the type of some name $n \in N$ is referred to. For any name n , if $A \vdash n : \tau$, following typing rule TYPE-ONAME or TYPE-PRILAB or TYPE-CNAME , we must have $\tau \preceq A(n)$. Then because A' is more general than A , following the definition, we also have $\tau \preceq A'(n)$, hence $A' \vdash n : \tau$. The replacement of the new type environment A' in other places in the tree preserves typing trivially. $\square$

A.2 Strong soundness w.r.t. class evaluation with hiding

We first show the soundness with respect to the filter evaluation $\Downarrow_S$.

Lemma 5 (Soundness w.r.t. filter evaluation). *Given a filter D with S , if for some type environment A the following conditions hold:*

$$A \vdash D :: \zeta(\rho)B^{W,V} \quad A \vdash S :: B'^R \quad \vdash W \textbf{ with } R :: W' \quad B \uparrow B'$$

then we have:

$$D \textbf{ with } S \Downarrow_S C$$

where C is not an error and:

$$A \vdash C :: \zeta(\rho)((B' \uparrow \overline{W'}) \oplus B)^{W',V}$$

Proof. We reason by induction on the depth of the evaluation of $D \textbf{ with } S$. We distinguish the root rule we use according to the structure of $D \textbf{ with } S$.

Case (Filter-End): We assume $S = \emptyset$. Then for any D , we always have:

$$D \textbf{ with } \emptyset \Downarrow_S D$$

By assumption, we also have:

$$A \vdash D :: \zeta(\rho)B^{W,V} \tag{1}$$

$$A \vdash \emptyset :: B'^R \tag{2}$$

$$\vdash W \textbf{ with } R :: W' \tag{3}$$

$$B \uparrow B'$$

and we should show $A \vdash D :: \zeta(\rho)((B' \uparrow \overline{W'}) \oplus B)^{W',V}$.

In (2), by typing rule TYPE-EMPTYCLAUSE, we have $B' = \emptyset$, $R = \emptyset$, hence in (3) we have $W' = W$, hence we have $\zeta(\rho)((B' \uparrow \overline{W'}) \oplus B)^{W',V} = \zeta(\rho)B^{W,V}$. We conclude by (1).

Case (Filter-Apply): We assume $D = K_1 \& K \triangleright P_v$ and $S = K_1 \Rightarrow K_2 \triangleright Q \mid S'$. By assumption, we have the following typing derivation for D . Note that we are going to number some premises in the derivation tree and some formulas in-lined in order to be able to refer to them later and this style will be followed throughout this appendix.

$$\frac{\text{TYPE-SYNCHRONIZATION} \quad \frac{(6) A_1 \vdash K_1 :: B_1 \quad (5) A_1 \vdash K :: B_k}{A_1 \vdash K_1 \& K :: B_1 \uplus B_k}}{\text{dom}[A_1] = \text{rv}[K_1 \& K] \quad (4) A + A_1 \vdash P_v} \text{TYPE-REACTION} \quad \frac{}{A \vdash K_1 \& K \triangleright P_v :: \zeta(\rho)B^{W,V}}$$

where $B = B_1 \uplus B_k$, (7) $W = \{\text{dl}[K_1 \& K]\} = \{\text{dl}[K_1] \uplus \text{dl}[K]\}$, and $V = \emptyset$.

Meanwhile, we have the following typing derivation for S :

$$\frac{\text{TYPE-CLAUSE} \quad \begin{array}{l} (8) A_2 \vdash K_1 :: B_{k1} \\ (9) A_2 \vdash K_2 :: B_{k2} \quad \text{dom}[A_2] = \text{rv}[K_2] \\ (10) A + A_2 \vdash Q \quad A \vdash S' :: B_s^{R_s} \end{array}}{A \vdash K_1 \Rightarrow K_2 \triangleright Q \mid S' :: B'^R}$$

where $B' = (B_{k1} \oplus B_{k2} \oplus B_s)$, (11) $R = \text{dl}[K_1] \Rightarrow \text{dl}[K_2] \mid R_s$, and $\text{rv}[K_1] \subseteq \text{rv}[K_2]$.

By (7) and (11), the typing derivation of $\vdash W$ **with** $R :: W'$ must look like:

$$\frac{\text{TYPE-APPLY} \quad \vdash \text{dl}[K_1] \uplus \text{dl}[K] \text{ **with** } \text{dl}[K_1] \Rightarrow \text{dl}[K_2] \mid R_s :: \text{dl}[K_2] \uplus \text{dl}[K]}{\vdash W \text{ **with** } \text{dl}[K_1] \Rightarrow \text{dl}[K_2] \mid R_s :: W'} \text{TYPE-OR}$$

where $W' = \{\text{dl}[K_2] \uplus \text{dl}[K]\} = \{\text{dl}[K_2 \& K]\}$. Note that the disjoint union $\uplus$ in W' guarantees the linearity of the derived join-pattern $K_2 \& K$ as regards labels. In other words, we have:

$$\text{dl}[K_2] \cap \text{dl}[K] = \emptyset \quad (12)$$

Moreover, by the linearity of received variables in join-patterns and α -conversion, we can always have:

$$\text{rv}[K_2] \cap \text{rv}[K] = \emptyset \quad (13)$$

Hence following rule FILTER-APPLY, we have:

$$K_1 \& K \triangleright P_v \text{ **with** } K_1 \Rightarrow K_2 \triangleright Q \mid S' \Downarrow_S K_2 \& K \triangleright P_v \& Q \text{ **or** } L \quad (14)$$

where $L = \text{dl}[K_1] \setminus \text{dl}[K_2]$.

We go on to show:

$$A \vdash K_2 \& K \triangleright P_v \& Q \text{ **or** } L :: \zeta(\rho)((B' \upharpoonright \overline{W'}) \oplus B)^{W',V}$$

According to the typing rules of join-patterns, if $A \vdash K :: B$, we must have the following: $\text{dom}[B] = \text{dl}[K]$; $\text{rv}[K] \subseteq \text{dom}[A]$; and assuming $K = \dots \& l(\dots, x, \dots) \& \dots$ and $x : \tau \in A$, $B(l) = (\dots, \tau, \dots)$. Then from (6) and (8), we have $\text{rv}[K_1] \subseteq \text{dom}[A_1]$ and $\text{rv}[K_1] \subseteq \text{dom}[A_2]$. And because $\text{dom}[A_1] = \text{rv}[K_1 \& K]$ and $\text{dom}[A_2] = \text{rv}[K_2]$, by (13), we thus have $\text{dom}[A_1] \cap \text{dom}[A_2] = \text{rv}[K_1]$. Moreover, we have $\text{dom}[B_1] = \text{dl}[K_1] = \text{dom}[B_{k1}]$ also from (6) and (8). And because $B \upharpoonright B'$, we have $B_1 = B_{k1}$. Therefore, in (6) and (8), A_1 and A_2 intersect and coincide on $\text{rv}[K_1]$. By applying Lemma 2 to (5) and to (9), we have:

$$A_1 + A_2 \vdash K :: B_k \quad (15)$$

$$A_2 + A_1 \vdash K_2 :: B_{k2} \quad (16)$$

where $\text{dom}[B_k] = \text{dl}[K]$, $\text{dom}[B_{k2}] = \text{dl}[K_2]$. Remark that we have $A_1 + A_2 = A_2 + A_1$. Let A_3 denote the resulting type environment. By (12), we have $\text{dom}[B_{k2}] \cap \text{dom}[B_k] = \emptyset$. Applying rule TYPE-SYNCHRONIZATION to (15) and (16), we thus have:

$$A_3 \vdash K_2 \& K :: B_k \uplus B_{k2} \quad (17)$$

Besides, by α -conversion of received variables, we always have $\text{dom}[A] \cap \text{dom}[A_i] = \emptyset$, for $i = 1, 2$. Then similarly, by applying Lemma 2 to (4) and to (10), we get:

$$A + A_1 + A_2 \vdash P_v \quad (18)$$

$$A + A_2 + A_1 \vdash Q \quad (19)$$

Applying typing rule TYPE-PARALLEL to (18) and (19), we have:

$$A + A_3 \vdash P_v \& Q \quad (20)$$

Moreover, we also have:

$$\begin{aligned} \text{dom}[A_3] &= \text{dom}[A_1] \cup \text{dom}[A_2] \\ &= \text{dom}[A_1] \setminus \text{rv}[K_1] \cup \text{dom}[A_2] && \text{because } \text{rv}[K_1] \subseteq \text{dom}[A_2] \\ &= \text{rv}[K_1 \& K] \setminus \text{rv}[K_1] \cup \text{rv}[K_2] \\ &= \text{rv}[K] \cup \text{rv}[K_2] \\ &= \text{rv}[K_2 \& K] \end{aligned} \quad (21)$$

Applying typing rule TYPE-REACTION to (17), (20), and (21), we thus have:

$$A \vdash K_2 \& K \triangleright P_v \& Q :: \zeta(\rho)(B_k \uplus B_{k2})^{\{\text{dl}[K_2 \& K]\}, \emptyset}$$

Because $L = \text{dl}[K_1] \setminus \text{dl}[K_2]$ and $\text{dom}[B_{k1} \setminus \text{dl}[K_2]] = \text{dl}[K_1] \setminus \text{dl}[K_2]$, by typing rule ABSTRACT, we have:

$$A \vdash L :: \zeta(\rho)(B_{k1} \setminus \text{dl}[K_2])^{\emptyset, \emptyset}$$

Because $\text{dom}[B_{k1}] \cap \text{dom}[B_k] = \text{dl}[K_1] \cap \text{dl}[K] = \emptyset$ by join-pattern linearity, we have $\text{dom}[B_k \uplus B_{k2}] \cap \text{dom}[B_{k1} \setminus \text{dl}[K_2]] = \emptyset$, hence $(B_k \uplus B_{k2}) \oplus (B_{k1} \setminus \text{dl}[K_2])$ makes sense. Then by typing rule TYPE-DISJUNCTION, we derive:

$$A \vdash K_2 \& K \triangleright P_v \& Q \text{ or } L :: \zeta(\rho)((B_k \uplus B_{k2}) \oplus (B_{k1} \setminus \text{dl}[K_2]))^{\{\text{dl}[K_2 \& K]\}, \emptyset}$$

To conclude, we check the following:

$$\begin{aligned} (B' \upharpoonright \overline{W'}) \oplus B &= ((B_{k1} \oplus B_{k2} \oplus B_s) \upharpoonright \text{dl}[K_2 \& K]) \oplus B \\ &= (((B_{k1} \setminus \text{dom}[B_{k2}]) \oplus B_{k2} \oplus (B_s \setminus \text{dom}[B_{k2}])) \upharpoonright \text{dl}[K_2 \& K]) \oplus B \\ &= (((B_{k1} \setminus \text{dl}[K_2]) \oplus B_{k2} \oplus (B_s \setminus \text{dl}[K_2])) \upharpoonright \text{dl}[K_2 \& K]) \oplus B \end{aligned}$$

Because $\text{dom}[B_{k1} \setminus \text{dl}[K_2]] \cap \text{dl}[K_2 \& K] = \emptyset$, $\text{dom}[B_{k2}] \subseteq \text{dl}[K_2 \& K]$, and $\text{dl}[K_2] \cap \text{dom}[B_s \setminus \text{dl}[K_2]] = \emptyset$, we have:

$$\begin{aligned} &= B_{k2} \oplus ((B_s \setminus \text{dl}[K_2]) \upharpoonright \text{dl}[K]) \oplus B \\ &= B_{k2} \oplus (B_s \upharpoonright \text{dl}[K]) \oplus B && \text{by (12), i.e. } \text{dl}[K_2] \cap \text{dl}[K] = \emptyset \\ &= B_{k2} \oplus (B_s \upharpoonright \text{dl}[K]) \oplus (B_1 \uplus B_k) \\ &= B_{k2} \oplus (B_s \upharpoonright \text{dl}[K]) \oplus (B_{k1} \uplus B_k) \\ &= B_{k2} \oplus (B_s \upharpoonright \text{dl}[K]) \oplus B_{k1} \oplus B_k \end{aligned}$$

Because $\text{dl}[K] \subseteq \text{dom}[B]$, we have $\text{dom}[B \upharpoonright \text{dl}[K]] \supseteq \text{dom}[B' \upharpoonright \text{dl}[K]]$, in other words $\text{dom}[B_k] \supseteq \text{dom}[B' \upharpoonright \text{dl}[K]] = \text{dom}[(B_{k1} \oplus B_{k2} \oplus B_s) \upharpoonright \text{dl}[K]]$. Because by linearity, $\text{dom}[B_{ki}] \cap \text{dl}[K] = \emptyset$, for $i = 1, 2$, we thus have $\text{dom}[B_k] \supseteq \text{dom}[B_s \upharpoonright \text{dl}[K]]$. Therefore:

$$\begin{aligned}
&= B_{k2} \oplus B_{k1} \oplus B_k \\
&= B_{k2} \oplus (B_{k1} \setminus \text{dom}[B_{k2}]) \oplus B_k \\
&= B_{k2} \oplus (B_{k1} \setminus \text{dl}[K_2]) \oplus B_k \\
&= (B_k \oplus B_{k2}) \oplus (B_{k1} \setminus \text{dl}[K_2]) \\
&= (B_k \uplus B_{k2}) \oplus (B_{k1} \setminus \text{dl}[K_2]) \quad \text{because } \text{dom}[B_{k2}] \cap \text{dom}[B_k] = \emptyset
\end{aligned}$$

Case (Filter-Or): We assume $D = D_1$ **or** D_2 . The typing derivation of D thus looks like:

$$\frac{\text{TYPE-DISJUNCTION} \quad (23) \ A \vdash D_1 :: \zeta(\rho)B_1^{W_1, V_1} \quad (22) \ A \vdash D_2 :: \zeta(\rho)B_2^{W_2, V_2}}{A \vdash D_1 \text{ or } D_2 :: \zeta(\rho)B^{W, V}}$$

where $B = B_1 \oplus B_2$, $W = W_1 \cup W_2$, and $V = V_1 \cup V_2$. Because $B \upharpoonright B'$, we also have (24) $B_1 \upharpoonright B'$, and (25) $B_2 \upharpoonright B'$.

We assume $W = \{\pi_i^{i \in I_1 \cup I_2}\}$, $W_1 = \{\pi_i^{i \in I_1}\}$, and $W_2 = \{\pi_i^{i \in I_2}\}$. A derivation of $\vdash W$ **with** $R :: W'$ then looks like:

$$\frac{\text{OR} \quad (\vdash \pi_i \text{ with } R :: \pi'_i)^{i \in I_1} \quad (\vdash \pi_i \text{ with } R :: \pi'_i)^{i \in I_2}}{\vdash W \text{ with } R :: W'}$$

where $W' = \{\pi'_i^{i \in I_1 \cup I_2}\}$. Hence we can build the following two derivations:

$$\frac{\text{OR} \quad (\vdash \pi_i \text{ with } R :: \pi'_i)^{i \in I_1}}{(26) \ \vdash W_1 \text{ with } R :: W'_1}$$

and

$$\frac{\text{OR} \quad (\vdash \pi_i \text{ with } R :: \pi'_i)^{i \in I_2}}{(27) \ \vdash W_2 \text{ with } R :: W'_2}$$

where $W'_1 = \{\pi'_i^{i \in I_1}\}$, and $W'_2 = \{\pi'_i^{i \in I_2}\}$, hence (28) $W' = W'_1 \cup W'_2$. Applying induction hypothesis to (23), (26), (24), and $A \vdash S :: B'^R$, we thus have:

$$D_1 \text{ with } S \Downarrow_S C_1 \quad (29)$$

where C_1 is not an error and:

$$A \vdash C_1 :: \zeta(\rho)((B' \upharpoonright \overline{W'_1}) \oplus B_1)^{W'_1, V_1} \quad (30)$$

Similarly, applying induction hypothesis to (22), (27), (25), and $A \vdash S :: B'^R$, we also have:

$$D_2 \text{ with } S \Downarrow_S C_2 \quad (31)$$

where C_2 is not an error and:

$$A \vdash C_2 :: \zeta(\rho)((B' \upharpoonright \overline{W'_2}) \oplus B_2)^{W'_2, V_2} \quad (32)$$

By evaluation rule FILTER-OR, we thus have:

$$\frac{\text{FILTER-OR} \quad (29) \quad (31)}{D_1 \text{ or } D_2 \text{ with } S \Downarrow_S C_1 \text{ or } C_2}$$

where $C_1 \text{ or } C_2$ is not an error. Moreover applying typing rule TYPE-DISJUNCTION to (30) and (32), and since B' , B_1 and B_2 are compatible with each other, we also have:

$$A \vdash C_1 \text{ or } C_2 :: \zeta(\rho)((B' \upharpoonright \overline{W'_1}) \oplus B_1 \oplus (B' \upharpoonright \overline{W'_2}) \oplus B_2)^{W', V}$$

where:

$$\begin{aligned} & (B' \upharpoonright \overline{W'_1}) \oplus B_1 \oplus (B' \upharpoonright \overline{W'_2}) \oplus B_2 \\ &= (B' \upharpoonright (\overline{W'_1} \cup \overline{W'_2})) \oplus (B_1 \oplus B_2) \\ &= (B' \upharpoonright (\overline{W'_1} \cup \overline{W'_2})) \oplus B \\ &= (B' \upharpoonright \overline{W'}) \oplus B \quad \text{by (28) and definition of } \overline{W'} \end{aligned}$$

Case (Filter-Next): We assume $D = M \triangleright P_v$, $S = K_1 \Rightarrow K_2 \triangleright Q \mid S'$, and $\text{dl}[K_1] \not\subseteq \text{dl}[M]$.

According to the typing rule TYPE-REACTION, we must have $W = \{\text{dl}[M]\}$ and $V = \emptyset$ in $A \vdash M \triangleright P_v :: \zeta(\rho)B^{W, V}$.

Meanwhile a typing derivation of S looks like:

$$\frac{\begin{array}{l} \text{TYPE-CLAUSe} \\ A' \vdash K_1 :: B_1 \\ A' \vdash K_2 :: B_2 \quad \text{dom}[A'] = \text{rv}[K_2] \\ A + A' \vdash Q \quad (33) \ A \vdash S' :: B_s^{R_s} \end{array}}{A \vdash K_1 \Rightarrow K_2 \triangleright Q \mid S' :: B'^R}$$

where $B' = B_1 \oplus B_2 \oplus B_s$, and $R = \text{dl}[K_1] \Rightarrow \text{dl}[K_2] \mid R_s$. Because $B \upharpoonright B'$, we have (34) $B \upharpoonright B_s$.

By the assumption $\text{dl}[K_1] \not\subseteq \text{dl}[M]$, the typing derivation of $\vdash W \text{ with } R :: W'$ must look like:

$$\frac{\frac{\text{dl}[K_1] \not\subseteq \text{dl}[M] \quad (35) \ \vdash \text{dl}[M] \text{ with } R_s :: \pi'}{(36) \ \vdash \text{dl}[M] \text{ with } \text{dl}[K_1] \Rightarrow \text{dl}[K_2] \mid R_s :: \pi'} \text{TYPE-NEXT}}{\vdash \{\text{dl}[M]\} \text{ with } \text{dl}[K_1] \Rightarrow \text{dl}[K_2] \mid R_s :: W'} \text{TYPE-OR}$$

where $W' = \{\pi'\}$.

Applying typing rule TYPE-OR directly to (35), we have $\vdash \{\text{dl}[M]\} \text{ with } R_s :: \{\pi'\}$, namely:

$$\vdash W \text{ with } R_s :: W' \quad (37)$$

Applying induction hypothesis to D , S' (33), (37), and (34), we thus have:

$$M \triangleright P_v \text{ with } S' \Downarrow_S C \quad (38)$$

where C is not an error and:

$$A \vdash C :: \zeta(\rho)((B_s \upharpoonright \overline{W}') \oplus B)^{W',V} \quad (39)$$

Then applying rule FILTER-NEXT to (38) and because $\text{dl}[K_1] \not\subseteq \text{dl}[M]$, we have:

$$M \triangleright P_v \text{ with } K_1 \Rightarrow K_2 \triangleright Q \mid S' \Downarrow_S C$$

Following the design of the type system, we have that for any class type $\zeta(\rho)B^{W,V}$, the channels organized in W are included in the ones bound in B . Namely, $\overline{W} \subseteq \text{dom}[B]$. Therefore, we have:

$$\text{dom}[(B_1 \oplus B_2) \upharpoonright \overline{W}'] \subseteq \underbrace{\overline{W}' \subseteq \text{dom}[(B_s \upharpoonright \overline{W}') \oplus B]}_{\text{by (39)}}$$

Hence, we have:

$$\begin{aligned} (B \upharpoonright \overline{W}') \oplus B &= ((B_1 \oplus B_2 \oplus B_s) \upharpoonright \overline{W}') \oplus B \\ &= ((B_1 \oplus B_2) \upharpoonright \overline{W}') \oplus (B_s \upharpoonright \overline{W}') \oplus B \end{aligned}$$

Because $\text{dom}[(B_1 \oplus B_2) \upharpoonright \overline{W}'] \subseteq \overline{W}'$ and $\overline{W}' \subseteq \text{dom}[(B_s \upharpoonright \overline{W}') \oplus B]$ from above, we have:

$$= (B_s \upharpoonright \overline{W}') \oplus B$$

□

In the class evaluation semantics, process evaluation $\Downarrow_P$ makes use of class evaluation $\Downarrow_C$ (see Figure 5). The following lemma states the soundness w.r.t. class evaluation $\Downarrow_C$.

Lemma 6. *Let A be a type environment and Γ be a class evaluation environment, such that $A^\mathcal{O} \vdash \Gamma :> A^C$. If $A + \text{this} : [\rho], \text{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V}$, then $\Gamma \models C \Downarrow_C C_v$, C_v is not an error, and $A^\mathcal{O} + \text{this} : [\rho], \text{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash C_v :: \zeta(\rho)B'^{W',V'}$. Moreover the type of C and the type of C_v satisfy the following conditions, for some fresh $H \subseteq \mathcal{H}$:*

1. $H \subseteq \overline{W}'$.
2. $B = B' \setminus H$.
3. $W = W' \setminus H$
4. $V = V' \cup \text{ftv}[B' \upharpoonright H]$

We recall the lemma that states the soundness w.r.t. process evaluation $\Downarrow_P$ as follows:

Lemma 1 (Strong soundness w.r.t. class evaluation with hiding). *Let A be a type environment and Γ be a class evaluation environment, such that $A^\mathcal{O} \vdash \Gamma :> A^C$. If $A \vdash P$, then $\Gamma \models P \Downarrow_P P_v$, P_v is not an error and $A^\mathcal{O} \vdash P_v$.*

We prove Lemma 1 and Lemma 6 simultaneously.

Proof. We reason by induction on the depth of process terms and class definition terms.

1.Base cases : Terms of depth 1

Case (P is 0): Trivial.

Case (P is $u.M$): First of all, according to rule EVAL-SEND, the evaluation of $u.M$ always returns $u.M$ itself. Moreover, because $u.M$ has no occurrence of class names, then by Lemma 2, we have $A \vdash u.M \implies A^\mathcal{O} \vdash u.M$.

Case (C is L): According to rule EVAL-ABSTRACT, the evaluation of L returns L itself. We assume $A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash L :: \zeta(\rho)B^{W,V}$. Because L has no occurrence of class names and by Lemma 2, we have $A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash L :: \zeta(\rho)B^{W,V}$. Taking $H = \emptyset$, we have that any class type satisfies the conditions of Lemma 6 with itself.

Case (C is c): We assume:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash c :: \zeta(\rho)B^{W,V}$$

Because $\mathbf{this}$ does not appear in c , by Lemma 2, we have $A \vdash c : \zeta(\rho)B^{W,V}$. Then according to typing rule TYPE-CNAME, $\zeta(\rho)B^{W,V}$ is an instance of $A(c)$, *i.e.* :

$$\zeta(\rho)B^{W,V} \preceq A(c) \quad (1)$$

Because $A^\mathcal{O} \vdash \Gamma :> A^c$, A and Γ have the same domain for class bindings, we have $c \in \text{dom}[\Gamma]$. Then by rule EVAL-CNAME, we have:

$$\Gamma \models c \Downarrow_C C_v$$

where $\Gamma(c) = C_v$ modulo some α -conversion to freshen the hidden names in $\Gamma(c)$.

By definition, $A^\mathcal{O} \vdash \Gamma :> A^c$ implies $A^\mathcal{O} \vdash \Gamma(c) :> A^c(c)$, *i.e.* $A^\mathcal{O} \vdash \Gamma(c) :> A(c)$. Following rule TYPE-REVEAL, we thus have for some ρ', B', W', V' , and $H' \subseteq \mathcal{H}$:

$$A^\mathcal{O} + \mathbf{this} : [\rho'], \mathbf{this} : (B' \upharpoonright \mathcal{F}) \vdash \Gamma(c) :: \zeta(\rho')B'^{W',V'} \quad (2)$$

$$B'' = B' \setminus H' \quad (3)$$

$$W'' = W' \setminus H' \quad (4)$$

$$H' \subseteq \overline{W'} \quad (5)$$

$$A(c) = \forall \text{Gen}(\rho', B'', A^\mathcal{O}). \zeta(\rho')B''^{W'',V' \cup \text{ftv}[B' \upharpoonright H']} \quad (6)$$

Because of (1) and (6), there must exist a substitution on type variables η with domain $\text{dom}[\eta] \subseteq \text{Gen}(\rho', B'', A^\mathcal{O})$, such that $\zeta(\rho)B^{W,V} = \eta(\zeta(\rho')B''^{W'',V' \cup \text{ftv}[B' \upharpoonright H']})$. Following the definition of substituting a class type, we thus have:

$$\begin{aligned} \rho &= \eta(\rho') \\ B &= \eta(B'') \end{aligned} \quad (7)$$

$$W = W'' \quad (8)$$

$$V = \eta(V' \cup \text{ftv}[B' \upharpoonright H']) \quad (9)$$

Applying the substitution η to (2), and by Lemma 3, we have:

$$A^O + \mathbf{this} : [\rho], \mathbf{this} : (\eta(B') \upharpoonright \mathcal{F}) \vdash \Gamma(c) :: \zeta(\rho)\eta(B')^{W', \eta(V')}$$

We express the α -conversion from $\Gamma(c)$ to C_v by subscripting with α . We thus get:

$$A^O + \mathbf{this} : [\rho], \mathbf{this} : (\eta(B'_\alpha) \upharpoonright \mathcal{F}) \vdash C_v :: \zeta(\rho)\eta(B'_\alpha)^{W'_\alpha, \eta(V')} \quad (10)$$

We show that $\zeta(\rho)B^{W,V}$ and $\zeta(\rho)\eta(B'_\alpha)^{W'_\alpha, \eta(V')}$ satisfy the conditions of Lemma 6 w.r.t. the fresh H'_α . By (3) – (5) and (7) – (9), we have:

$$\begin{aligned} H'_\alpha &\subseteq \overline{W'_\alpha} \subseteq \text{dom}[B'_\alpha] \\ B &= \eta(B'') = \eta(B' \setminus H') = \eta(B'_\alpha \setminus H'_\alpha) = \eta(B'_\alpha) \setminus H'_\alpha \\ W &= W'' = W' \setminus H' = W'_\alpha \setminus H'_\alpha \\ V &= \eta(V' \cup \text{ftv}[B' \upharpoonright H']) = \eta(V') \cup \text{ftv}[\eta(B') \upharpoonright H'] = \eta(V') \cup \text{ftv}[\eta(B'_\alpha) \upharpoonright H'_\alpha] \end{aligned}$$

Because H'_α is fresh, $\text{dom}[B_*] \cap H'_\alpha = \emptyset$. Moreover, because $\text{dom}[B_*] \cap \text{dom}[B] = \emptyset$ by assumption and $B = \eta(B'_\alpha) \setminus H'_\alpha$ from above, we also have $\text{dom}[B_*] \cap \text{dom}[\eta(B'_\alpha) \setminus H'_\alpha] = \emptyset$. Then as a consequence, we have $\text{dom}[B_*] \cap \text{dom}[\eta(B'_\alpha)] = \emptyset$. Adding the useless B_* into the assumption of (10), (intuitively similar to Lemma 2,) we finally get:

$$A^O + \mathbf{this} : [\rho], \mathbf{this} : ((\eta(B'_\alpha); B_*) \upharpoonright \mathcal{F}) \vdash C_v :: \zeta(\rho)\eta(B'_\alpha)^{W'_\alpha, \eta(V')}$$

2. Induction cases We suppose that for any processes or class definitions of depth $< n$, Lemma 1 and Lemma 6 hold. We prove it is also the case for terms of depth n . We distinguish the top-most structure the terms may have.

Case (P is $P \ \& \ Q$): We assume:

$$A \vdash P \ \& \ Q$$

According to the typing rule TYPE-PARALLEL, this judgment is derived by the following two hypotheses:

$$A \vdash P \qquad A \vdash Q$$

Because $\text{depth}[P]$ is $< n$, by induction hypothesis, we have:

$$\Gamma \models P \Downarrow_P P_v \quad (11)$$

$$A^O \vdash P_v \quad (12)$$

Similarly, applying induction hypothesis to Q , we have:

$$\Gamma \models Q \Downarrow_P Q_v \quad (13)$$

$$A^O \vdash Q_v \quad (14)$$

Applying rule EVAL-PARALLEL to (11) and (13), we thus have:

$$\Gamma \models P \ \& \ Q \Downarrow_P P_v \ \& \ Q_v$$

where $P_v \ \& \ Q_v$ is not an error, and by applying rule TYPE-PARALLEL to (12) and (14), we also have:

$$A^O \vdash P_v \ \& \ Q_v$$

Case (P is **obj $x = C$ in P'):** We assume:

$$A \vdash \mathbf{obj} \ x = C \text{ in } P'$$

A typing derivation of this judgment looks like:

$$\begin{array}{c} \text{TYPE-OBJECT} \\ (16) \ A + \mathbf{this} : [\rho], \mathbf{this} : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V} \\ X = \text{Gen}(\rho, B, A) \setminus \text{ctv}[B^W] \setminus V \quad (15) \ \rho = B \upharpoonright \mathcal{M} \\ (17) \ A + x : \forall X. [\rho] \vdash P' \quad \text{dom}[B] = \overline{W} \\ \hline A \vdash \mathbf{obj} \ x = C \text{ in } P' \end{array}$$

Because $\text{depth}[C] < n$, applying induction hypothesis to (16), we have:

$$\begin{array}{l} \Gamma \vdash C \Downarrow_C (D \text{ or } L) \mathbf{init} \ Q_v \\ A^{\mathcal{O}} + \mathbf{this} : [\rho], \mathbf{this} : (B' \upharpoonright \mathcal{F}) \vdash (D \text{ or } L) \mathbf{init} \ Q_v :: \zeta(\rho)B'^{W',V'} \end{array}$$

where $B = B' \setminus H$, $W = W' \setminus H$ and $V = V' \cup \text{ftv}[B' \upharpoonright H]$, for some $H \subseteq \mathcal{H}$ and $H \subseteq \overline{W'} \subseteq \text{dom}[B']$. Then because $\text{dom}[B] = \overline{W}$, *i.e.* $\text{dom}[B' \setminus H] = \overline{W'} \setminus \overline{H}$, *i.e.* $\text{dom}[B'] \setminus H = \overline{W'} \setminus H$, we have:

$$\text{dom}[B'] = \overline{W'} \tag{18}$$

Then following the typing rules for classes, we must have $L = \emptyset$, namely:

$$\Gamma \vdash C \Downarrow_C (D \text{ or } \emptyset) \mathbf{init} \ Q_v \tag{19}$$

$$A^{\mathcal{O}} + \mathbf{this} : [\rho], \mathbf{this} : (B' \upharpoonright \mathcal{F}) \vdash D \mathbf{init} \ Q_v :: \zeta(\rho)B'^{W',V'} \tag{20}$$

By (15), we have:

$$\rho = B \upharpoonright \mathcal{M} = (B' \setminus H) \upharpoonright \mathcal{M} = B' \upharpoonright \mathcal{M} \tag{21}$$

Let:

$$X' = \text{Gen}(\rho, B', A^{\mathcal{O}}) \setminus \text{ctv}[B'^{W'}] \setminus V' \tag{22}$$

Because:

$$\begin{aligned} \text{ctv}[B'^{W'}] \cup V' &\subseteq (\text{ctv}[B'^{(W' \setminus H)}] \cup \text{ftv}[B' \upharpoonright H]) \cup V' \\ &= (\text{ctv}[(B' \setminus H)^{(W' \setminus H)}] \cup \text{ftv}[B' \upharpoonright H]) \cup V' \\ &= (\text{ctv}[B^W] \cup \text{ftv}[B' \upharpoonright H]) \cup V' \\ &= \text{ctv}[B^W] \cup (\text{ftv}[B' \upharpoonright H] \cup V') \\ &= \text{ctv}[B^W] \cup V \end{aligned}$$

and

$$\text{Gen}(\rho, B, A) \subseteq \text{Gen}(\rho, B, A^{\mathcal{O}}) \subseteq \text{Gen}(\rho, B', A^{\mathcal{O}})$$

we have:

$$X \subseteq X'$$

Therefore, we have that $\forall X'.[\rho]$ is more general than $\forall X.[\rho]$. Then by applying Lemma 4 to (17), we have:

$$A + x : \forall X'.[\rho] \vdash P' \quad (23)$$

By α -conversion, we can always assume that x does not appear free in Γ . Hence applying Lemma 2 to $A^\mathcal{O} \vdash \Gamma :> A^C$, we have $A^\mathcal{O} + x : \forall X'.[\rho] \vdash \Gamma :> A^C$, i.e. $(A + x : \forall X'.[\rho])^\mathcal{O} \vdash \Gamma :> (A + x : \forall X'.[\rho])^C$. Let $A' = A + x : \forall X'.[\rho]$. Applying induction hypothesis to A' and Γ with (23), we have:

$$\Gamma \models P' \Downarrow_P P'_v \quad (24)$$

$$A^\mathcal{O} + x : \forall X'.[\rho] \vdash P'_v \quad (25)$$

Applying rule EVAL-OBJECT to (19) and (24), we have:

$$\Gamma \vdash \mathbf{obj} \ x = C \ \mathbf{in} \ P' \Downarrow_P \ \mathbf{obj} \ x = D \ \mathbf{init} \ Q_v \ \mathbf{in} \ P'_v$$

Moreover, by rule TYPE-OBJECT, we also have:

$$\text{TYPE-OBJECT} \frac{\begin{array}{c} (20) \\ (22) \quad (21) \\ (25) \quad (18) \end{array}}{A^\mathcal{O} \vdash \mathbf{obj} \ x = D \ \mathbf{init} \ Q_v \ \mathbf{in} \ P'_v}$$

Case (P is class $c = C$ hide F in P'): We assume:

$$A \vdash \mathbf{class} \ c = C \ \mathbf{hide} \ F \ \mathbf{in} \ P'$$

A typing derivation of this judgment looks like:

$$\begin{array}{c} \text{TYPE-HIDE} \\ (27) \ A + \mathbf{this} : [\rho], \mathbf{this} : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V} \quad (26) \ \rho = B \upharpoonright \mathcal{M}; \varrho \\ B' = B \setminus F \quad W' = W \setminus F \\ (28) \ A + c : \forall \text{Gen}(\rho, B', A). \zeta(\rho)B'^{W', V \cup \text{ftv}[B \upharpoonright F]} \vdash P' \quad F \subseteq \overline{W} \\ \hline A \vdash \mathbf{class} \ c = C \ \mathbf{hide} \ F \ \mathbf{in} \ P' \end{array}$$

Because $\text{depth}[C] < n$, applying induction hypothesis to (27), we have:

$$\Gamma \models C \Downarrow_C (D \ \mathbf{or} \ L) \ \mathbf{init} \ Q_v \quad (29)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : (B_1 \upharpoonright \mathcal{F}) \vdash (D \ \mathbf{or} \ L) \ \mathbf{init} \ Q_v :: \zeta(\rho)B_1^{W_1, V_1} \quad (30)$$

where $B = B_1 \setminus H_1$, $W = W_1 \setminus H_1$ and $V = V_1 \cup \text{ftv}[B_1 \upharpoonright H_1]$, for some $H_1 \subseteq \mathcal{H}$ and $H_1 \subseteq \overline{W_1} \subseteq \text{dom}[B_1]$. Following typing rules for classes, for (30) to hold, we must have:

$$\begin{aligned}
 \text{dl}[D] &= \overline{W_1} \\
 L &= \text{dom}[B_1] \setminus \overline{W_1} \\
 &= ((\text{dom}[B_1] \setminus H_1) \uplus (\text{dom}[B_1] \upharpoonright H_1)) \setminus ((\overline{W_1} \setminus H_1) \uplus (\overline{W_1} \upharpoonright H_1)) \\
 &= ((\text{dom}[B_1] \setminus H_1) \uplus H_1) \setminus ((\overline{W_1} \setminus H_1) \uplus H_1) && \text{because } H_1 \subseteq \overline{W_1} \subseteq \text{dom}[B_1] \\
 &= (\text{dom}[B_1 \setminus H_1] \uplus H_1) \setminus (\overline{W_1 \setminus H_1} \uplus H_1) \\
 &= (\text{dom}[B] \uplus H_1) \setminus (\overline{W} \uplus H_1) \\
 &= \text{dom}[B] \setminus \overline{W}
 \end{aligned}$$

Then because $F \subseteq \overline{W}$ and $\overline{W} \subseteq \overline{W_1}$, we have:

$$F \subseteq \text{dl}[D] \quad (31)$$

and $F \cap L = \emptyset$. Taking a fresh $H \subseteq \mathcal{H}$ and renaming channels F to H in (30), we get:

$$\begin{aligned}
 A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : (B_1\{H/F\} \upharpoonright \mathcal{F}) \vdash \\
 (D\{H/F\}_{\mathcal{H}} \text{ or } L) \mathbf{init} Q_v\{H/F\}_{\mathcal{H}} :: \zeta(\rho)B_1\{H/F\}^{W_1\{H/F\}, V_1} \quad (32)
 \end{aligned}$$

Note that ρ remains intact during the renaming because $\text{dom}[\rho] \cap \mathcal{F} = \emptyset$. As none of H, F, H_1 contains labels from $\mathcal{M}$, by (26), we have:

$$\begin{aligned}
 \rho &= B \upharpoonright \mathcal{M}; \varrho \\
 &= (B_1 \setminus H_1) \upharpoonright \mathcal{M}; \varrho \\
 &= B_1 \upharpoonright \mathcal{M}; \varrho \\
 &= (B_1\{H/F\}) \upharpoonright \mathcal{M}; \varrho
 \end{aligned} \quad (33)$$

Besides, we also have:

$$\begin{aligned}
 B' &= B \setminus F \\
 &= (B_1 \setminus H_1) \setminus F \\
 &= (B_1 \setminus F) \setminus H_1 \\
 &= (B_1\{H/F\} \setminus H) \setminus H_1 && \text{because } \text{dom}[B_1 \setminus F] \cap H = \emptyset \\
 &= B_1\{H/F\} \setminus \{H_1 \cup H\}
 \end{aligned} \quad (34)$$

and similarly:

$$W' = W_1\{H/F\} \setminus \{H_1 \cup H\} \quad (35)$$

Moreover, because $H_1 \cap F = H_1 \cap H = \emptyset$, we have:

$$H_1 \cup H = ((H_1 \cup H)\{F/H\})\{H/F\} = (H_1 \cup F)\{H/F\}$$

And because:

$$H_1 \cup F \subseteq H_1 \cup \overline{W} = H_1 \cup \overline{W_1 \setminus H_1} = \overline{W_1}$$

we have:

$$H_1 \cup H \subseteq \overline{W_1} \{H/F\} = \overline{W_1 \{H/F\}} \quad (36)$$

Applying rule TYPE-REVEAL to (32), (33), (34), (35), and (36), we thus have:

$$\begin{aligned} A^\mathcal{O} \vdash (D\{H/F\}_\mathcal{H} \text{ or } L) \text{ init } Q_v\{H/F\}_\mathcal{H} :> \\ \forall \text{Gen}(\rho, B', A). \zeta(\rho) B'^{W', V_1 \cup \text{ftv}[B_1 \{H/F\} \upharpoonright (H_1 \cup H)]} \end{aligned}$$

Because:

$$\begin{aligned} & V_1 \cup \text{ftv}[B_1 \{H/F\} \upharpoonright (H_1 \cup H)] \\ &= V_1 \cup \text{ftv}[B_1 \{H/F\} \upharpoonright H_1] \cup \text{ftv}[B_1 \{H/F\} \upharpoonright H] \\ &= V_1 \cup \text{ftv}[B_1 \upharpoonright H_1] \cup \text{ftv}[B_1 \{H/F\} \upharpoonright H] \quad \text{because } H_1 \cap F = H_1 \cap H = \emptyset \\ &= V_1 \cup \text{ftv}[B_1 \upharpoonright H_1] \cup \text{ftv}[(B_1 \{H/F\}) \{F/H\} \upharpoonright H \{F/H\}] \\ &= V_1 \cup \text{ftv}[B_1 \upharpoonright H_1] \cup \text{ftv}[B_1 \upharpoonright F] \quad \text{because } \text{dom}[B_1 \setminus F] \cap H = \emptyset \\ &= V_1 \cup \text{ftv}[B_1 \upharpoonright H_1] \cup \text{ftv}[(B_1 \setminus H_1) \upharpoonright F] \quad \text{because } H_1 \cap F = \emptyset \\ &= V \cup \text{ftv}[B \upharpoonright F] \end{aligned}$$

that is:

$$A^\mathcal{O} \vdash (D\{H/F\}_\mathcal{H} \text{ or } L) \text{ init } Q_v\{H/F\}_\mathcal{H} :> \forall \text{Gen}(\rho, B', A). \zeta(\rho) B'^{W', V \cup \text{ftv}[B \upharpoonright F]} \quad (37)$$

Let:

$$\begin{aligned} A' &= A + c : \forall \text{Gen}(\rho, B', A). \zeta(\rho) B'^{W', V \cup \text{ftv}[B \upharpoonright F]} \\ \Gamma' &= \Gamma + (c \mapsto (D\{H/F\}_\mathcal{H} \text{ or } L) \text{ init } Q_v\{H/F\}_\mathcal{H}) \end{aligned}$$

Because $A^\mathcal{O} \vdash \Gamma :> A^\mathcal{C}$ and by (37), we have $A'^\mathcal{O} \vdash \Gamma' :> A'^\mathcal{C}$. Applying induction hypothesis to (28), we thus have:

$$\Gamma' \models P' \Downarrow_P P'_v \quad (38)$$

$$A'^\mathcal{O} \vdash P'_v \quad (39)$$

Applying rule EVAL-HIDE to (29), (31), (38) and because H is fresh, we have:

$$\Gamma \models \text{class } c = C \text{ hide } F \text{ in } P' \Downarrow_P P'_v$$

Moreover, because $A^\mathcal{O} = A'^\mathcal{O}$, we also have $A^\mathcal{O} \vdash P'_v$ by (39).

Case (C is $M \triangleright P$): We assume:

$$A + \text{this} : [\rho], \text{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash M \triangleright P :: \zeta(\rho) B^{W, V}$$

The typing derivation of this judgment looks like:

$$\begin{array}{c} \text{TYPE-REACTION} \\ \frac{\text{dom}[A_M] = \text{rv}[M] \quad A_M \vdash M :: B \quad (40) \quad A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) + A_M \vdash P}{A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash M \triangleright P :: \zeta(\rho)B^{\{\text{dl}[M]\}, \emptyset}} \end{array}$$

where $W = \{\text{dl}[M]\}$, and $V = \emptyset$.

Let $A' = A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) + A_M$. By α -conversion, we can always assume $\text{rv}[M]$ fresh with respect to $\text{dom}[A]$, namely $\text{dom}[A_M] \cap \text{dom}[A] = \emptyset$. Besides, following rules $\text{TYPE-}\Gamma_{>}$ and TYPE-REVEAL , we also know that extra bindings of **this** do not matter because **this** is always rebound. Hence applying Lemma 2 to $A^O \vdash \Gamma :> A^C$, we have $A^O + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) + A_M \vdash \Gamma :> A^C$, i.e. $A'^O \vdash \Gamma :> A'^C$ as there is no class name binding in A_M . Applying induction hypothesis to (40), we get:

$$\Gamma \models P \Downarrow_P P_v \quad (41)$$

$$A^O + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) + A_M \vdash P_v \quad (42)$$

Applying rule EVAL-REACTION to (41), we thus have:

$$\Gamma \models M \triangleright P \Downarrow_C M \triangleright P_v$$

Moreover, replacing (40) by (42) in the typing derivation, we can also build a typing derivation of the following judgment:

$$A^O + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash M \triangleright P_v :: \zeta(\rho)B^{W,V}$$

where taking $H = \emptyset$, we have that $\zeta(\rho)B^{W,V}$ and itself satisfy the conditions of Lemma 6.

Case (C is C_1 or C_2): We assume:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_1 \text{ or } C_2 :: \zeta(\rho)B^{W,V}$$

A typing derivation of this judgment looks like:

$$\begin{array}{c} \text{TYPE-DISJUNCTION} \\ \frac{A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_1 :: \zeta(\rho)B_1^{W_1, V_1} \quad A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_2 :: \zeta(\rho)B_2^{W_2, V_2}}{A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_1 \text{ or } C_2 :: \zeta(\rho)B^{W, V}} \end{array}$$

where $B = B_1 \oplus B_2$, $W = W_1 \cup W_2$, and $V = V_1 \cup V_2$. Hence the two premises can also be written as:

$$\begin{array}{l} A + \mathbf{this} : [\rho], \mathbf{this} : ((B_1; (B_2 \setminus \text{dom}[B_1]); B_*) \upharpoonright \mathcal{F}) \vdash C_1 :: \zeta(\rho)B_1^{W_1, V_1} \\ A + \mathbf{this} : [\rho], \mathbf{this} : ((B_2; (B_1 \setminus \text{dom}[B_2]); B_*) \upharpoonright \mathcal{F}) \vdash C_2 :: \zeta(\rho)B_2^{W_2, V_2} \end{array}$$

Because both C_1 and C_2 are of depth $< n$, by induction hypothesis and typing rule **TYPE-INITIALIZER**, we have:

$$\Gamma \models C_1 \Downarrow_C (D_1 \text{ or } L_1) \text{ init } P_{v1} \quad (43)$$

$$\Gamma \models C_2 \Downarrow_C (D_2 \text{ or } L_2) \text{ init } P_{v2} \quad (44)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; (B_2 \setminus \text{dom}[B_1]); B_*) \upharpoonright \mathcal{F}) \vdash D_1 \text{ or } L_1 :: \zeta(\rho)B'_1{}^{W'_1, V'_1} \quad (45)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_2; (B_1 \setminus \text{dom}[B_2]); B_*) \upharpoonright \mathcal{F}) \vdash D_2 \text{ or } L_2 :: \zeta(\rho)B'_2{}^{W'_2, V'_2} \quad (46)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; (B_2 \setminus \text{dom}[B_1]); B_*) \upharpoonright \mathcal{F}) \vdash P_{v1} \quad (47)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_2; (B_1 \setminus \text{dom}[B_2]); B_*) \upharpoonright \mathcal{F}) \vdash P_{v2} \quad (48)$$

where $\zeta(\rho)B'_1{}^{W'_1, V'_1}$ and $\zeta(\rho)B'_2{}^{W'_2, V'_2}$ satisfy the conditions of Lemma 6 w.r.t. fresh $H_1 \subseteq \mathcal{H}$ and $H_1 \subseteq \overline{W'_1} \subseteq \text{dom}[B'_1]$; and so do $\zeta(\rho)B'_2{}^{W'_2, V'_2}$ and $\zeta(\rho)B'_1{}^{W'_1, V'_1}$ w.r.t. fresh $H_2 \subseteq \mathcal{H}$ and $H_2 \subseteq \overline{W'_2} \subseteq \text{dom}[B'_2]$.

Applying rule **EVAL-DISJUNCTION** to (43) and (44), we thus have:

$$\Gamma \models C_1 \text{ or } C_2 \Downarrow_C (D \text{ or } L) \text{ init } P_v$$

where $D = D_1 \text{ or } D_2$, $P_v = P_{v1} \& P_{v2}$, and $L = (L_1 \setminus \text{dl}[D_2]) \cup (L_2 \setminus \text{dl}[D_1])$.

We are left to check the typing of $(D \text{ or } L) \text{ init } P_v$. Because H_1 and H_2 are fresh, we have:

$$\begin{aligned} B_2 \setminus \text{dom}[B_1] &= B_2 \setminus \text{dom}[B'_1 \setminus H_1] \\ &= B_2 \setminus (\text{dom}[B'_1] \setminus H_1) \\ &= B_2 \setminus \text{dom}[B'_1] \quad \text{because } \text{dom}[B_2] \cap H_1 = \emptyset \\ &= (B'_2 \setminus H_2) \setminus \text{dom}[B'_1] \\ &= (B'_2 \setminus \text{dom}[B'_1]) \setminus H_2 \end{aligned}$$

and similarly:

$$B_1 \setminus \text{dom}[B_2] = (B'_1 \setminus \text{dom}[B'_2]) \setminus H_1$$

Because adding fresh labels into the assumptions of (45) – (48) will still keep the judgments true, (intuitively similar to Lemma 2,) we thus have:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; (B'_2 \setminus \text{dom}[B'_1]); B_*) \upharpoonright \mathcal{F}) \vdash D_1 \text{ or } L_1 :: \zeta(\rho)B'_1{}^{W'_1, V'_1}$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_2; (B'_1 \setminus \text{dom}[B'_2]); B_*) \upharpoonright \mathcal{F}) \vdash D_2 \text{ or } L_2 :: \zeta(\rho)B'_2{}^{W'_2, V'_2}$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; (B'_2 \setminus \text{dom}[B'_1]); B_*) \upharpoonright \mathcal{F}) \vdash P_{v1}$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_2; (B'_1 \setminus \text{dom}[B'_2]); B_*) \upharpoonright \mathcal{F}) \vdash P_{v2}$$

Besides, since $B_1 \oplus B_2$ makes sense and by the freshness of H_1 and H_2 , we also have that $B'_1 \oplus B'_2$ makes sense. Hence, we have:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : (((B'_1 \oplus B'_2); B_*) \upharpoonright \mathcal{F}) \vdash D_1 \text{ or } L_1 :: \zeta(\rho)B'_1{}^{W'_1, V'_1} \quad (49)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : (((B'_2 \oplus B'_1); B_*) \upharpoonright \mathcal{F}) \vdash D_2 \text{ or } L_2 :: \zeta(\rho)B'_2{}^{W'_2, V'_2} \quad (50)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : (((B'_1 \oplus B'_2); B_*) \upharpoonright \mathcal{F}) \vdash P_{v1} \quad (51)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : (((B'_2 \oplus B'_1); B_*) \upharpoonright \mathcal{F}) \vdash P_{v2} \quad (52)$$

Let $B' = B'_1 \oplus B'_2 = B'_2 \oplus B'_1$, $W' = W'_1 \cup W'_2$, and $V' = V'_1 \cup V'_2$. Applying rule TYPE-DISJUNCTION to (49) and (50) we thus have:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash D_1 \text{ or } L_1 \text{ or } D_2 \text{ or } L_2 :: \zeta(\rho)B'^{W', V'}$$

Namely:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash (D_1 \text{ or } D_2) \text{ or } (L_1 \cup L_2) :: \zeta(\rho)B'^{W', V'}$$

Because $L = (L_1 \setminus \text{dl}[D_2]) \cup (L_2 \setminus \text{dl}[D_1])$ and $D = D_1 \text{ or } D_2$, this can be rephrased as:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash D \text{ or } L :: \zeta(\rho)B'^{W', V'} \quad (53)$$

Meanwhile, applying rule TYPE-PARALLEL, to (51) and (52), we get:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash P_v \quad (54)$$

Hence, applying typing rule TYPE-INITIALIZER to (53) and (54), we get:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash (D \text{ or } L) \text{ init } P_v :: \zeta(\rho)B'^{W', V'}$$

It remains to check against the conditions in Lemma 6. Let $H = H_1 \uplus H_2$. We have:

$$\begin{aligned} H &= H_1 \uplus H_2 \\ &\subseteq (\overline{W'_1} \cup \overline{W'_2}) && \text{by condition 1 of Lemma 6} \\ &= \overline{W'_1 \cup W'_2} && \text{by definition of } \overline{W} \\ &= \overline{W'} \end{aligned}$$

$$\begin{aligned} B &= B_1 \oplus B_2 \\ &= (B'_1 \setminus H_1) \oplus (B'_2 \setminus H_2) && \text{by condition 2 of Lemma 6} \\ &= (B'_1 \setminus (H_1 \uplus H_2)) \oplus (B'_2 \setminus (H_2 \uplus H_1)) && \text{because } H_1 \cap B'_2 = H_2 \cap B'_1 = \emptyset \\ &= (B'_1 \oplus B'_2) \setminus H \\ &= B' \setminus H \end{aligned}$$

$$\begin{aligned} W &= W_1 \cup W_2 \\ &= (W'_1 \setminus H_1) \cup (W'_2 \setminus H_2) && \text{by condition 3 of Lemma 6} \\ &= (W'_1 \setminus H) \cup (W'_2 \setminus H) && \text{because } H_1 \cap \overline{W'_2} = H_2 \cap \overline{W'_1} = \emptyset \\ &= (W'_1 \oplus W'_2) \setminus H \\ &= W' \setminus H \end{aligned}$$

$$\begin{aligned} V &= V_1 \cup V_2 \\ &= (V'_1 \cup \text{ftv}[B'_1 \upharpoonright H_1]) \cup (V'_2 \cup \text{ftv}[B'_2 \upharpoonright H_2]) \\ &= (V'_1 \cup V'_2) \cup (\text{ftv}[B'_1 \upharpoonright H_1] \cup \text{ftv}[B'_2 \upharpoonright H_2]) \\ &= V' \cup \text{ftv}[(B'_1 \upharpoonright H_1) \cup (B'_2 \upharpoonright H_2)] \\ &= V' \cup \text{ftv}[(B'_1 \upharpoonright H) \oplus (B'_2 \upharpoonright H)] && \text{because } H_1 \cap B'_2 = H_2 \cap B'_1 = \emptyset \\ &= V' \cup \text{ftv}[(B'_1 \oplus B'_2) \upharpoonright H] \\ &= V' \cup \text{ftv}[B' \upharpoonright H] \end{aligned}$$

Case (C is **match C_1 with S end):** We assume:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash \mathbf{match} C_1 \mathbf{with} S \mathbf{end} :: \zeta(\rho)B^{W,V}$$

A typing derivation of this judgment looks like:

TYPE-REFINEMENT

$$\begin{array}{c} (57) \ A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_1 :: \zeta(\rho)B_1^{W_1,V} \quad (55) \ \vdash W_1 \mathbf{with} R :: W \\ (58) \ A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash S :: B_S^R \quad (56) \ B_1 \upharpoonright B_S \\ \hline A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash \mathbf{match} C_1 \mathbf{with} S \mathbf{end} :: \zeta(\rho)B^{W,V} \end{array}$$

where (59) $B = (B_S \upharpoonright \overline{W}) \oplus B_1$.

Writing $(B; B_*)$ as $(B_1; B \setminus \text{dom}[B_1]; B_*)$ and applying induction hypothesis to (57), we have:

$$\Gamma \models C_1 \Downarrow_C (D_1 \mathbf{or} L_1) \mathbf{init} P_v \quad (60)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash (D_1 \mathbf{or} L_1) \mathbf{init} P_v :: \zeta(\rho)B_1^{W'_1,V'}$$

As no free class names appear in $(D_1 \mathbf{or} L_1) \mathbf{init} P_v$, by Lemma 2, we have:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash (D_1 \mathbf{or} L_1) \mathbf{init} P_v :: \zeta(\rho)B_1^{W'_1,V'} \quad (61)$$

where for some fresh $H_1 \subseteq \mathcal{H}$, we have:

$$\begin{aligned} H_1 &\subseteq \overline{W'_1} \\ B_1 &= B'_1 \setminus H_1 \end{aligned} \quad (62)$$

$$\begin{aligned} W_1 &= W'_1 \setminus H_1 \\ V &= V' \cup \text{ftv}[B'_1 \upharpoonright H_1] \end{aligned} \quad (63)$$

By rules TYPE-INITIALIZER and TYPE-DISJUNCTION to (61), we have:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash D_1 :: \zeta(\rho)B_{D_1}^{W'_1,V'} \quad (64)$$

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash L_1 :: \zeta(\rho)B_{L_1}^{\emptyset,\emptyset} \quad (65)$$

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash P_v \quad (66)$$

where $B'_1 = B_{D_1} \oplus B_{L_1}$. Writing $(B; B_*)$ as $(B_1; B \setminus \text{dom}[B_1]; B_*)$ in (58), too, we have:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash S :: B_S^R$$

By (62), that is:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1 \setminus H_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash S :: B_S^R$$

Because H_1 is fresh and adding fresh labels in the assumption keeps the judgment above true, (intuitively similar to Lemma 2,) we thus have:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \upharpoonright \mathcal{F}) \vdash S :: B_S^R \quad (67)$$

Meanwhile, following definition of $\uparrow$ and typing rules for filters, the freshness of H_1 also gives:

$$B'_1 \uparrow B_S \quad \text{by (56)} \quad (68)$$

$$\vdash W'_1 \text{ with } R :: W' \quad \text{by (55)} \quad (69)$$

where:

$$H_1 \subseteq \overline{W'} \quad (70)$$

$$W = W' \setminus H_1 \quad (71)$$

Applying Lemma 5 to (64), (67), (69), and (68), we have:

$$D_1 \text{ with } S \Downarrow_S C_2 \quad (72)$$

$$A + \text{this} : [\rho], \text{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \uparrow \mathcal{F}) \vdash C_2 :: \zeta(\rho)(B_S \uparrow \overline{W'} \oplus B_{D_1})^{W', V'}$$

Following rule TYPE-DISJUNCTION and by (65), we have:

$$A + \text{this} : [\rho], \text{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \uparrow \mathcal{F}) \vdash C_2 \text{ or } L_1 :: \zeta(\rho)(B_S \uparrow \overline{W'} \oplus B'_1)^{W', V'}$$

Because H_1 is fresh, *i.e.* $H_1 \cap \text{dom}[B_S] = \emptyset$, and by (71), this can be rephrased as:

$$A + \text{this} : [\rho], \text{this} : ((B'_1; B \setminus \text{dom}[B_1]; B_*) \uparrow \mathcal{F}) \vdash C_2 \text{ or } L_1 :: \zeta(\rho)(B_S \uparrow \overline{W} \oplus B'_1)^{W', V'} \quad (73)$$

Let $B' = B'_1; B \setminus \text{dom}[B_1]$. We have:

$$\begin{aligned} B' &= B'_1; B \setminus \text{dom}[B_1] \\ &= B'_1; ((B_S \uparrow \overline{W}) \oplus B_1) \setminus \text{dom}[B_1] && \text{by (59)} \\ &= B'_1; (B_S \uparrow \overline{W}) \setminus \text{dom}[B_1] \\ &= B'_1; (B_S \uparrow \overline{W}) \setminus \text{dom}[B'_1 \setminus H_1] && \text{by (62)} \\ &= B'_1; (B_S \uparrow \overline{W}) \setminus \text{dom}[B'_1] && \text{because } H_1 \cap \text{dom}[B_S] = \emptyset \\ &= B_S \uparrow \overline{W} \oplus B'_1 && \text{by (68)} \end{aligned}$$

Hence, applying induction hypothesis to (73), we have:

$$\Gamma \models C_2 \text{ or } L_1 \Downarrow_C D \text{ or } L \quad (74)$$

$$A^{\mathcal{O}} + \text{this} : [\rho], \text{this} : ((B''; B_*) \uparrow \mathcal{F}) \vdash D \text{ or } L :: \zeta(\rho)B''^{W'', V''} \quad (75)$$

where for some fresh $H_2 \subseteq \mathcal{H}$:

$$H_2 \subseteq \overline{W''} \quad (76)$$

$$B' = B'' \setminus H_2 \quad (77)$$

$$W' = W'' \setminus H_2 \quad (78)$$

$$V' = V'' \cup \text{ftv}[B'' \uparrow H_2] \quad (79)$$

Applying rule EVAL-REFINEMENT to (60), (72), and (74), we thus can build a derivation of the following judgment:

$$\Gamma \vdash \mathbf{match} \ C_1 \ \mathbf{with} \ S \ \mathbf{end} \ \Downarrow_C \ (D \ \mathbf{or} \ L) \ \mathbf{init} \ P_v$$

We go on and check the typing of $(D \ \mathbf{or} \ L) \ \mathbf{init} \ P_v$. Because $B' = B'_1; B \setminus \text{dom}[B_1]$, judgment (66) is rephrased as:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash P_v$$

Because H_2 is fresh and adding fresh labels in the assumption keeps the judgment above true, (intuitively similar to Lemma 2,) we thus have:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B''; B_*) \upharpoonright \mathcal{F}) \vdash P_v \quad \text{by (77)}$$

As no free class name appear in P_v , by Lemma 2, we have:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B''; B_*) \upharpoonright \mathcal{F}) \vdash P_v \quad (80)$$

Hence applying rule TYPE-INITIALIZER to (75) and (80), we have:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B''; B_*) \upharpoonright \mathcal{F}) \vdash (D \ \mathbf{or} \ L) \ \mathbf{init} \ P_v :: \zeta(\rho)B''^{W'',V''}$$

It remains to check that the conditions of Lemma 6 hold between $\zeta(\rho)B''^{W'',V''}$ and $\zeta(\rho)B^{W,V}$. Let $H = H_1 \uplus H_2$. We have:

$$\begin{aligned} H &= H_1 \uplus H_2 \\ &\subseteq (\overline{W'} \cup \overline{W''}) && \text{by (70)(76)} \\ &= (\overline{W''} \setminus H_2 \cup \overline{W''}) && \text{by (78)} \\ &= \overline{W''} \end{aligned}$$

$$\begin{aligned} B &= B_S \upharpoonright \overline{W} \oplus B_1 && \text{by (59)} \\ &= B_S \upharpoonright \overline{W} \oplus (B'_1 \setminus H_1) && \text{by (62)} \\ &= (B_S \upharpoonright \overline{W} \oplus B'_1) \setminus H_1 && \text{because } H_1 \text{ fresh in } B_S \\ &= B' \setminus H_1 && \text{because } B' = B_S \upharpoonright \overline{W} \oplus B'_1 \\ &= B'' \setminus H_2 \setminus H_1 && \text{by (77)} \\ &= B'' \setminus H \end{aligned}$$

$$\begin{aligned} W &= W' \setminus H_1 && \text{by (71)} \\ &= W'' \setminus H_2 \setminus H_1 && \text{by (78)} \\ &= W'' \setminus H \end{aligned}$$

$$\begin{aligned}
V &= V' \cup \text{ftv}[B'_1 \upharpoonright H_1] && \text{by (63)} \\
&= V'' \cup \text{ftv}[B'' \upharpoonright H_2] \cup \text{ftv}[B'_1 \upharpoonright H_1] && \text{by (79)} \\
&= V'' \cup \text{ftv}[B'' \upharpoonright H_2] \cup \text{ftv}[(B_S \upharpoonright \overline{W} \oplus B'_1) \upharpoonright H_1] && \text{because } H_1 \text{ fresh in } B_S \\
&= V'' \cup \text{ftv}[B'' \upharpoonright H_2] \cup \text{ftv}[B' \upharpoonright H_1] && \text{because } B' = B_S \upharpoonright \overline{W} \oplus B'_1 \\
&= V'' \cup \text{ftv}[B'' \upharpoonright H_2] \cup \text{ftv}[(B'' \setminus H_2) \upharpoonright H_1] && \text{by (77)} \\
&= V'' \cup \text{ftv}[B'' \upharpoonright H_2] \cup \text{ftv}[B'' \upharpoonright H_1] && \text{because } H_2 \cap H_1 = \emptyset \\
&= V'' \cup \text{ftv}[B'' \upharpoonright H]
\end{aligned}$$

Case (C is $C_1 \text{ init } P$): We assume:

$$A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_1 \text{ init } P :: \zeta(\rho)B^{W,V}$$

A typing derivation of this judgment must be an instance of TYPE-INITIALIZER as:

$$\begin{array}{c}
\text{TYPE-INITIALIZER} \\
(81) \ A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_1 :: \zeta(\rho)B^{W,V} \\
(82) \ A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash P \\
\hline
A + \mathbf{this} : [\rho], \mathbf{this} : ((B; B_*) \upharpoonright \mathcal{F}) \vdash C_1 \text{ init } P :: \zeta(\rho)B^{W,V}
\end{array}$$

Because the depth of C_1 is $< n$, applying induction hypothesis to (81), we have:

$$\Gamma \models C_1 \Downarrow_C (D \text{ or } L) \text{ init } Q_v \quad (83)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash (D \text{ or } L) \text{ init } Q_v :: \zeta(\rho)B'^{W',V'} \quad (84)$$

where $\zeta(\rho)B'^{W',V'}$ and $\zeta(\rho)B^{W,V}$ satisfy the conditions in Lemma 6 for some fresh $H \subseteq \mathcal{H}$.

Because the depth of P is $< n$, applying induction hypothesis to (82), we have:

$$\Gamma \models P \Downarrow_P P'_v \quad (85)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash P'_v$$

Because H is fresh and $B = B' \setminus H$, we rephrase the judgment above by adding fresh labels in the assumption and we have:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash P'_v \quad (86)$$

Applying rule EVAL-INITIALIZER to (83) and (85), we thus can build a derivation for the following judgment:

$$\Gamma \models C \text{ init } P \Downarrow_C (D \text{ or } L) \text{ init } (Q_v \ \& \ P'_v)$$

We check the typing of $(D \text{ or } L) \text{ init } (Q_v \ \& \ P'_v)$. Applying rule TYPE-INITIALIZER to (84) to decompose $(D \text{ or } L) \text{ init } Q_v$ into $D \text{ or } L$ and Q_v , we have:

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash D \text{ or } L :: \zeta(\rho)B'^{W',V'} \quad (87)$$

$$A^\mathcal{O} + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash Q_v \quad (88)$$

By (86) and (88), we compose Q_v and P'_v in parallel following rule TYPE-PARALLEL, and we have:

$$A^O + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash Q_v \& P'_v \quad (89)$$

Finally, applying rule TYPE-INITIALIZER again to (87) and (89), we get:

$$A^O + \mathbf{this} : [\rho], \mathbf{this} : ((B'; B_*) \upharpoonright \mathcal{F}) \vdash (D \text{ or } L) \text{ init } Q_v \& P'_v :: \zeta(\rho)B'^{W', V'}$$

□

A.3 Subject reduction of chemical reduction

The chemical semantics is given in Figure 2. Taking place successively after class reduction, the chemical semantics has its initial solution of the form $\vdash P$, where P is the result of class evaluation and its privacy annotation is empty. As the only way to add an annotation is by rule REACT, it is not difficult to check that the following lemma holds.

Lemma 7. *For any chemical solutions $\mathcal{D} \Vdash \mathcal{P}$, if $(\mathbf{this} \mapsto x) \# P \in \mathcal{P}$, then $x.D \in \mathcal{D}$, i.e., $x \in \text{fv}[\mathcal{D}]$.*

Moreover, rule STR-OBJ requires that the current defined object name does not occur free in its internal positions, namely, $x \notin \text{fv}[\mathcal{D}] \cup \text{fv}[P]$, and this condition can always be satisfied by α -conversions. Hence, we also have the following lemma:

Lemma 8. *For any chemical solution $\mathcal{D} \Vdash \mathcal{P}$,*

1. *if $x.D \in \mathcal{D}$, then $x \notin \text{fv}[\mathcal{D}]$;*
2. *if $(\mathbf{this} \mapsto x) \# P \in \mathcal{P}$, then $x \notin \text{fv}[P]$.*

We now prove the subject reduction of chemical semantics. We recall the theorem as follows:

Theorem 2 (Subject reduction w.r.t. chemical reduction). *Let $\mathcal{D} \Vdash \mathcal{P}$ be a chemical solution. If $\vdash (\mathcal{D} \Vdash \mathcal{P})$ and $\mathcal{D} \Vdash \mathcal{P} \Rightarrow \mathcal{D}' \Vdash \mathcal{P}'$, then $\vdash (\mathcal{D}' \Vdash \mathcal{P}')$.*

Proof. According to the chemical semantics given in Fig 2, for $\mathcal{D} \Vdash \mathcal{P} \Rightarrow \mathcal{D}' \Vdash \mathcal{P}'$ to happen, either STR-OBJ is applied followed by CHEMISTRY-OBJ, or one the rules: STR-NUL, STR-PAR, STR-JOIN, THIS-COMM, OBJ-COMM, REACT, is applied followed by CHEMISTRY. We distinguish cases according to the rule applied.

Case (Str-Null): We have:

$$\mathcal{D} \Vdash \phi \# 0, \mathcal{P} \equiv \mathcal{D} \Vdash \mathcal{P}$$

And we should prove:

$$\vdash (\mathcal{D} \Vdash \phi \# 0, \mathcal{P}) \iff \vdash (\mathcal{D} \Vdash \mathcal{P})$$

According to typing rule TYPE-SOLUTION, it suffices to prove that $A \vdash 0$ always holds for any A , which follows directly from typing rule TYPE-NUL.

Case (Str-Par): We have:

$$\mathcal{D} \Vdash \phi \# (P \& Q), \mathcal{P} \equiv \mathcal{D} \Vdash \phi \# P, \phi \# Q, \mathcal{P}$$

And we should prove:

$$\vdash (\mathcal{D} \Vdash \phi \# (P \& Q), \mathcal{P}) \iff \vdash (\mathcal{D} \Vdash \phi \# P, \phi \# Q, \mathcal{P})$$

According to typing rule TYPE-SOLUTION, it suffices to prove that $A \vdash P \& Q$ is equivalent to $A \vdash P, A \vdash Q$ for any A , which follows directly from the typing rule TYPE-PARALLEL.

Case (Str-Join): We have:

$$\mathcal{D} \Vdash \phi \# u.(M_1 \& M_2), \mathcal{P} \equiv \mathcal{D} \Vdash \phi \# u.M_1, \phi \# u.M_2, \mathcal{P}$$

And we should prove:

$$\vdash (\mathcal{D} \Vdash \phi \# u.(M_1 \& M_2), \mathcal{P}) \iff \vdash (\mathcal{D} \Vdash \phi \# u.M_1, \phi \# u.M_2, \mathcal{P})$$

Similarly to the previous case, it suffices to prove $A \vdash u.(M_1 \& M_2)$ is equivalent to $A \vdash u.M_1, A \vdash u.M_2$ for any A , which follows directly from the typing rule TYPE-JOIN.

Case (This-Comm): We have:

$$\mathcal{D} \Vdash \phi \# \mathbf{this}.l(\tilde{u}), \mathcal{P} \longrightarrow \mathcal{D} \Vdash \Phi \# \phi(\mathbf{this}).l(\phi(\tilde{u})), \mathcal{P}$$

where $\phi = (\mathbf{this} \mapsto x)$ and $x.D \in \mathcal{D}$. Let $\tilde{u} = \{u_i\}_{i \in I}$. By Lemma 8, we have $(u_i \neq x)_{i \in I}$.

And we should prove:

$$\vdash (\mathcal{D} \Vdash \phi \# \mathbf{this}.l(\tilde{u}), \mathcal{P}) \implies \vdash (\mathcal{D} \Vdash \Phi \# \phi(\mathbf{this}).l(\phi(\tilde{u})), \mathcal{P})$$

According to typing rule TYPE-SOLUTION, it suffices to prove:

$$A^{\mathcal{M}} + A_x\{\mathbf{this}/x\} \vdash \mathbf{this}.l(\tilde{u}) \implies A \vdash x.l(\phi(\tilde{u}))$$

where $A = \bigcup_{z.D \in \mathcal{D}} A_z$ and we assume $A_x = x : \forall X. [\rho], x : \forall X. (B \upharpoonright \mathcal{F})$.

For the left hand side to hold, the following two premises must hold by typing rule TYPE-SEND:

$$A^{\mathcal{M}} + \mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) \vdash \mathbf{this}.l :: \tau_i^{i \in I} \quad (1)$$

$$(A^{\mathcal{M}} + \mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) \vdash u_i :: \tau_i)^{i \in I} \quad (2)$$

Applying Lemma 2 to (1), we have:

$$\mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) \vdash \mathbf{this}.l :: \tau_i^{i \in I}$$

Then by α -conversion, we have:

$$A_x \vdash x.l :: \tau_i^{i \in I}$$

Again by Lemma 2, we have:

$$A \vdash x.l :: \tau_i^{i \in I} \quad (3)$$

Similarly, by applying Lemma 2, α -conversion to (2), and because the derivations of (2) do not use any private assumption (thus adding private assumptions into the typing environment has no influence), we have:

$$(A \vdash \phi(u_i) :: \tau_i)^{i \in I} \quad (4)$$

Hence, by (3) and (4), we have:

$$A \vdash x.l(\phi(\tilde{u}))$$

Case (Obj-Comm): We have:

$$\mathcal{D} \Vdash \phi \# x.m(\tilde{u}), \mathcal{P} \longrightarrow \mathcal{D} \Vdash \Phi \# x.m(\phi(\tilde{u})), \mathcal{P}$$

And we should prove:

$$\vdash (\mathcal{D} \Vdash \phi \# x.m(\tilde{u}), \mathcal{P}) \implies \vdash (\mathcal{D} \Vdash \Phi \# x.m(\phi(\tilde{u})), \mathcal{P})$$

We distinguish the possible values of ϕ .

Case ($\phi = \Phi$): Trivial, because the chemical solution remains the same after the reduction.

Case (ϕ is empty): We should show $A^{\mathcal{M}} \vdash x.m(\tilde{u}) \implies A \vdash x.m(\tilde{u})$, and it follows directly from the fact that the typing derivation of $x.m(\tilde{u})$ does not use any private assumptions because m is public.

Case ($\phi = (\mathbf{this} \mapsto y)$): We have $y \neq x$, $(u_i \neq y)^{i \in I}$, and $y.D \in \mathcal{D}$. Then as in Case (THIS-COMM), we prove:

$$A^{\mathcal{M}} + A_y\{\mathbf{this}/y\} \vdash x.m(\tilde{u}) \implies A \vdash x.m(\phi(\tilde{u}))$$

Case (React): We have:

$$\mathcal{D}, x.D \Vdash \Phi \# x.M\sigma, \mathcal{P} \longrightarrow \mathcal{D}, x.D \Vdash (\mathbf{this} \mapsto x) \# P\sigma, \mathcal{P}$$

where $D = (\dots M \triangleright P \dots)$, and M is of the form $\&_{i \in I} l_i(\tilde{u}_i)$.

And we should prove:

$$\vdash (\mathcal{D}, x.D \Vdash \Phi \# x.M\sigma, \mathcal{P}) \implies \vdash (\mathcal{D}, x.D \Vdash (\mathbf{this} \mapsto x) \# P\sigma, \mathcal{P})$$

Assume the left hand side holds, then according to rule TYPE-SOLUTION, the following premises must hold:

$$A = (\bigcup_{z.D' \in \mathcal{D}} A_z) \cup A_x \quad (5)$$

$$A^{\mathcal{M}} \vdash x.D :: A_x$$

$$A \vdash x.M\sigma \quad (6)$$

where for (5) to hold, according to TYPE-DEFINITION, we in turn have the following premises:

$$\begin{aligned}
 & A^{\mathcal{M}} + \mathbf{this} : [\rho], \mathbf{this}(B \upharpoonright \mathcal{F}) \vdash D :: \zeta(\rho)B^{W, \emptyset} \\
 & \rho = B \upharpoonright \mathcal{M} \\
 & \text{dom}[B] = \overline{W} \\
 & X = \text{Gen}(\rho, B, A^{\mathcal{M}}) \setminus \text{ctv}[B^W]
 \end{aligned} \tag{7}$$

and

$$A_x = x : \forall X. [\rho], x : \forall X. (B \upharpoonright \mathcal{F})$$

Therefore, to demonstrate $\vdash (\mathcal{D}, x.D \Vdash (\mathbf{this} \mapsto x) \# P\sigma, \mathcal{P})$, it suffices to show that:

$$A^{\mathcal{M}} + \mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) \vdash P\sigma$$

Because $D = (\dots M \triangleright P \dots)$ and M is of the form $\&_{i \in I} l_i(\tilde{u}_i)$, following the typing rule TYPE-DISJUNCTION, a derivation of (7) must have the following sub-derivation:

$$\begin{array}{c}
 \text{TYPE-REACTION} \\
 \text{dom}[A'] = \text{rv}[M] \quad A' \vdash M :: B_1 \\
 (8) \quad A^{\mathcal{M}} + \mathbf{this} : [\rho], \mathbf{this}(B \upharpoonright \mathcal{F}) + A' \vdash P \\
 \hline
 A^{\mathcal{M}} + \mathbf{this} : [\rho], \mathbf{this}(B \upharpoonright \mathcal{F}) \vdash M \triangleright P :: \zeta(\rho)B_1^{W_1, \emptyset}
 \end{array}$$

where $B_1 \subseteq B$, $W_1 = \{\text{dl}[M]\} \subseteq W$, and $A' = (\tilde{u}_i : B_1(l_i))^{i \in I}$, namely:

$$A' = (\tilde{u}_i : B(l_i))^{i \in I}$$

Moreover, by typing rules TYPE-JOIN, TYPE-SEND, TYPE-PUBLAB, and TYPE-PRILAB, a derivation of (6) must have the following premises:

$$\begin{aligned}
 & (A \vdash x.l_i :: \tilde{\tau}_i)^{i \in I} \\
 & (A \vdash \sigma(\tilde{u}_i) : \tilde{\tau}_i)^{i \in I}
 \end{aligned} \tag{9}$$

where $\tilde{\tau}_i = \eta_i(B(l_i))$, and η_i is a substitution of domain $X \cap \text{ftv}[B(l_i)]$. Because $\{\text{dl}[M]\} \subseteq W$, we have that any two channels l_{i_1} and l_{i_2} defined in M are correlated, hence $\text{ftv}[B(l_{i_1})] \cap \text{ftv}[B(l_{i_2})] \cap X = \emptyset$, hence $\text{dom}[\eta_{i_1}] \cap \text{dom}[\eta_{i_2}] = \emptyset$. Let η be the sum of $\eta_i^{i \in I}$. Applying Lemma 3 to (8) with η , we thus have:

$$A^{\mathcal{M}} + \mathbf{this} : [\eta(\rho)], \mathbf{this} : (\eta(B) \upharpoonright \mathcal{F}) + \eta(A') \vdash P \tag{10}$$

where $A^{\mathcal{M}}$ remains the same, because $\text{dom}[\eta]$ is a subset of X , thus is disjoint from the free type variables in $A^{\mathcal{M}}$. Because $\text{ftv}[B(l_{i_1})] \cap \text{dom}[\eta] \subseteq \text{dom}[\eta_{i_1}]$, we have:

$$\eta(A') = (\tilde{u}_i : \eta(B(l_i)))^{i \in I} = (\tilde{u}_i : \eta_i(B(l_i)))^{i \in I} = (\tilde{u}_i : \tilde{\tau}_i)^{i \in I}$$

Applying Lemma 4 to (10), we can make **this** polymorphic, namely:

$$A^{\mathcal{M}} + \mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) + (\tilde{u}_i : \tilde{\tau}_i)^{i \in I} \vdash P \tag{11}$$

Moreover, because the derivations of (9) do not use any private assumption, nor any **this** assumptions, by Lemma 2, we have:

$$(A^M + \mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) \vdash \sigma(\tilde{u}_i) : \tilde{\tau}_i)^{i \in I}$$

Hence we can replace $\tilde{u}_i$ s by $\eta(\tilde{u}_i)$ s in (11) and get:

$$A^M + \mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) \vdash P\sigma$$

Case (Str-Obj): We have:

$$\mathcal{D} \Vdash \phi \# \mathbf{obj} \ x = D \ \mathbf{init} \ P \ \mathbf{in} \ Q, \mathcal{P} \equiv \mathcal{D}, x. (D \ \mathbf{or} \ \mathbf{init}() \triangleright P) \Vdash \Phi \# x.\mathbf{init}(), \phi \# Q, \mathcal{P}$$

where $x \notin \text{fv}[D] \cup \text{fv}[P]$ and $x \notin \text{fv}[\mathcal{D}] \cup \text{fv}[\mathcal{P}]$. We should prove:

$$\begin{aligned} \vdash (\mathcal{D} \Vdash \phi \# \mathbf{obj} \ x = D \ \mathbf{init} \ P \ \mathbf{in} \ Q, \mathcal{P}) &\iff \\ \vdash (\mathcal{D}, x. (D \ \mathbf{or} \ \mathbf{init}() \triangleright P) \Vdash \Phi \# x.\mathbf{init}(), \phi \# Q, \mathcal{P}) \end{aligned}$$

We distinguish the possible values of ϕ .

Case ($\phi = (\mathbf{this} \mapsto y)$): According to Lemma 7, we have $y \in \text{fv}[\mathcal{D}]$. Because $x \notin \text{fv}[\mathcal{D}]$, we have $y \neq x$. Let $z.D'$ denote the active objects in $\mathcal{D}$, that is, $z \in \text{fv}[\mathcal{D}]$. Again because $x \notin \text{fv}[\mathcal{D}]$, we have $z \neq x$. Moreover, following Lemma 7, the processes in $\mathcal{P}$ can only be one of the following three kinds: P' , $\Phi \# P'$, or $(\mathbf{this} \mapsto z) \# P'$. On one hand, for the left hand side to hold, the following typing derivation must hold:

$$\begin{array}{c} \text{TYPE-SOLUTION} \\ \frac{\begin{array}{l} A = (\bigcup_{z.D' \in \mathcal{D}} A_z) \quad (16) \quad (A^M \vdash z.D' :: A_z)^{z.D' \in \mathcal{D}} \\ (15) \quad A^M + A_y\{\mathbf{this}/y\} \vdash \mathbf{obj} \ x = D \ \mathbf{init} \ P \ \mathbf{in} \ Q \quad (14) \quad (A^M \vdash P')^{P' \in \mathcal{P}} \\ (13) \quad (A \vdash P')^{\Phi \# P' \in \mathcal{P}} \quad (12) \quad (A^M + A_z\{\mathbf{this}/z\} \vdash P')^{(\mathbf{this} \mapsto z) \# P' \in \mathcal{P}} \end{array}}{\vdash (\mathcal{D} \Vdash (\mathbf{this} \mapsto y) \# \mathbf{obj} \ x = D \ \mathbf{init} \ P \ \mathbf{in} \ Q, \mathcal{P})} \end{array}$$

For (15) to hold, by typing rules TYPE-OBJECT and TYPE-INITIALIZER, the following premises must hold:

$$A^M + \mathbf{this} : [\rho_x], \mathbf{this} : (B_x \upharpoonright \mathcal{F}) \vdash D :: \zeta(\rho_x) B_x^{W_x, \emptyset} \quad (17)$$

$$A^M + \mathbf{this} : [\rho_x], \mathbf{this} : (B_x \upharpoonright \mathcal{F}) \vdash P \quad (18)$$

$$A^M + A_y\{\mathbf{this}/y\} + x : \forall X_x. [\rho_x] \vdash Q \quad (19)$$

$$\rho_x = B_x \upharpoonright \mathcal{M} \quad (20)$$

$$\text{dom}[B_x] = \overline{W_x} \quad (21)$$

$$X_x = \text{Gen}(\rho_x, B_x, A^M) \setminus \text{ctv}[B_x^{W_x}] \quad (22)$$

On the other hand, for the right hand side to hold, the following typing derivation must hold:

$$\begin{array}{c} \text{TYPE-SOLUTION} \\ \frac{\begin{array}{l} A' = (\bigcup_{z.D' \in \mathcal{D}} A'_z) \cup A'_x \\ (29) \quad (A'^M \vdash z.D' :: A'_z)^{z.D' \in \mathcal{D}} \quad (28) \quad A'^M \vdash x.(D \ \mathbf{or} \ \mathbf{init}() \triangleright P) :: A'_x \\ (27) \quad A' \vdash x.\mathbf{init}() \quad (26) \quad A'^M + A'_y\{\mathbf{this}/y\} \vdash Q \quad (25) \quad (A'^M \vdash P')^{P' \in \mathcal{P}} \\ (24) \quad (A' \vdash P')^{\Phi \# P' \in \mathcal{P}} \quad (23) \quad (A'^M + A'_z\{\mathbf{this}/z\} \vdash P')^{(\mathbf{this} \mapsto z) \# P' \in \mathcal{P}} \end{array}}{\vdash (\mathcal{D}, x. (D \ \mathbf{or} \ \mathbf{init}() \triangleright P) \Vdash \Phi \# x.\mathbf{init}(), (\mathbf{this} \mapsto y) \# Q, \mathcal{P})} \end{array}$$

For (28) and (27) to hold, by typing rules **TYPE-DEFINITION**, **TYPE-DISJUNCTION**, **TYPE-REACTION**, **TYPE-SEND**, and **TYPE-PRILAB**, the following premises must hold:

$$A'^{\mathcal{M}} + \mathbf{this} : [\rho'_x], \mathbf{this} : ((B'_x; \mathbf{init} : ()) \upharpoonright \mathcal{F}) \vdash D :: \zeta(\rho'_x) B'_x{}^{W'_x, \emptyset} \quad (30)$$

$$A'^{\mathcal{M}} + \mathbf{this} : [\rho'_x], \mathbf{this} : ((B'_x; \mathbf{init} : ()) \upharpoonright \mathcal{F}) \vdash P \quad (31)$$

$$\rho'_x = (B'_x; \mathbf{init} : ()) \upharpoonright \mathcal{M} \quad (32)$$

$$\text{dom}[B'_x; \mathbf{init} : ()] = \overline{W'_x} \cup \{\{\mathbf{init}\}\} \quad (33)$$

$$X'_x = \text{Gen}(\rho'_x, B'_x; \mathbf{init} : (), A'^{\mathcal{M}}) \setminus \text{ctv}[B'_x; \mathbf{init} : ()^{W'_x \cup \{\{\mathbf{init}\}\}}] \quad (34)$$

where:

$$A'_x = x : \forall X'_x. [\rho'_x], x : \forall X'_x. ((B'_x; \mathbf{init} : ()) \upharpoonright \mathcal{F})$$

Because **init** is a special private channel of empty tuple type and never occurring in programs, premises from (30) – (34) thus are equivalent to the following list of premises:

$$A'^{\mathcal{M}} + \mathbf{this} : [\rho'_x], \mathbf{this} : (B'_x \upharpoonright \mathcal{F}) \vdash D :: \zeta(\rho'_x) B'_x{}^{W'_x, \emptyset} \quad (35)$$

$$A'^{\mathcal{M}} + \mathbf{this} : [\rho'_x], \mathbf{this} : (B'_x \upharpoonright \mathcal{F}) \vdash P \quad (36)$$

$$\rho'_x = B'_x \upharpoonright \mathcal{M} \quad (37)$$

$$\text{dom}[B'_x] = \overline{W'_x} \quad (38)$$

$$X'_x = \text{Gen}(\rho'_x, B'_x, A'^{\mathcal{M}}) \setminus \text{ctv}[B'_x{}^{W'_x}] \quad (39)$$

We show the equivalence of corresponding premises of both sides. We first have the following equivalence:

$$\begin{aligned} A_y &= A'_y & A_z &= A'_z \\ \rho_x &= \rho'_x & B_x &= B'_x & W_x &= W'_x \\ A' &= A \cup x : \forall X_x. [\rho_x], x : \forall X_x. ((B_x; \mathbf{init} : ()) \upharpoonright \mathcal{F}) \end{aligned}$$

Hence we have the following equivalence between premises:

$$(20), (21) \iff (37), (38)$$

Moreover, by (22) and (39), we also have:

$$\begin{aligned} X'_x &= \text{Gen}(\rho'_x, B'_x, A'^{\mathcal{M}}) \setminus \text{ctv}[B'_x{}^{W'_x}] \\ &= \text{Gen}(\rho_x, B_x, A'^{\mathcal{M}}) \setminus \text{ctv}[B_x{}^{W_x}] \\ &= \text{Gen}(\rho_x, B_x, A^{\mathcal{M}}) \setminus \text{ftv}[\forall X_x. [\rho_x]] \setminus \text{ctv}[B_x{}^{W_x}] \\ &= (\text{Gen}(\rho_x, B_x, A^{\mathcal{M}}) \setminus \text{ctv}[B_x{}^{W_x}]) \setminus \text{ftv}[\forall X_x. [\rho_x]] \\ &= X_x \setminus \text{ftv}[\forall X_x. [\rho_x]] \\ &= X_x \end{aligned}$$

Because $x \notin \{y, z\} \cup \text{fv}[D'] \cup \text{fv}[P'] \cup \text{fv}[D] \cup \text{fv}[P]$, by Lemma 2, we can also draw the following equivalence between premises:

$$\begin{aligned} (16) &\iff (29) \\ (17), (18) &\iff (35), (36) \\ (19) &\iff (26) \\ (12) - (14) &\iff (23) - (25) \end{aligned}$$

To conclude, following the equivalence we draw between premises, if the left hand side holds, letting $A' = A \cup x : \forall X_{x \cdot} [\rho_x], x : \forall X_{x \cdot} ((B_x; \mathbf{init} : ()) \vdash \mathcal{F})$, a derivation tree can also be built for the right hand side; conversely, if the right hand side holds, letting $A = A' \setminus \{x\}$, a derivation tree can also be built for the left hand side.

Case (ϕ is empty): For the left hand side to hold, we must have the following typing derivation:

$$\begin{array}{c} \text{TYPE-SOLUTION} \\ A = \bigcup_{z.D' \in \mathcal{D}} A_z \quad (44) \quad (A^{\mathcal{M}} \vdash z.D' :: A_z)^{z.D' \in \mathcal{D}} \\ (43) \quad A \vdash \mathbf{obj} \ x = D \ \mathbf{init} \ P \ \mathbf{in} \ Q \quad (42) \quad (A^{\mathcal{M}} \vdash P')^{P' \in \mathcal{P}} \\ (41) \quad (A \vdash P')^{\Phi \# P' \in \mathcal{P}} \quad (40) \quad (A^{\mathcal{M}} + A_z\{\mathbf{this}/z\} \vdash P')^{(\mathbf{this} \mapsto z) \# P' \in \mathcal{P}} \\ \hline \vdash (\mathcal{D} \Vdash \mathbf{obj} \ x = D \ \mathbf{init} \ P \ \mathbf{in} \ Q, \mathcal{P}) \end{array}$$

And for the right hand side to hold, we must have the following typing derivation:

$$\begin{array}{c} \text{TYPE-SOLUTION} \\ A' = (\bigcup_{z.D' \in \mathcal{D}} A'_z) \cup A'_x \\ (51) \quad (A'^{\mathcal{M}} \vdash z.D' :: A'_z)^{z.D' \in \mathcal{D}} \quad (50) \quad A'^{\mathcal{M}} \vdash x.(D \ \mathbf{or} \ \mathbf{init}() \triangleright P) :: A'_x \\ (49) \quad A'^{\mathcal{M}} \vdash x.\mathbf{init}() \quad (48) \quad A'^{\mathcal{M}} \vdash Q \quad (47) \quad (A'^{\mathcal{M}} \vdash P')^{P' \in \mathcal{P}} \\ (46) \quad (A' \vdash P')^{\Phi \# P' \in \mathcal{P}} \quad (45) \quad (A'^{\mathcal{M}} + A'_z\{\mathbf{this}/z\} \vdash P')^{(\mathbf{this} \mapsto z) \# P' \in \mathcal{P}} \\ \hline \vdash (\mathcal{D}, x.(D \ \mathbf{or} \ \mathbf{init}() \triangleright P) \Vdash \Phi \# x.\mathbf{init}(), Q, \mathcal{P}) \end{array}$$

Similarly to the previous case, we let $A' = A \cup A'_x$, and draw the equivalence between the premises of the two sides as:

$$\begin{aligned} (44) &\iff (51) \\ (43) &\iff (50), (49), (48) \\ (42) &\iff (47) \\ (41) &\iff (46) \\ (40) &\iff (45) \end{aligned}$$

□

A.4 Safety

We prove the safety property of the type system with respect to chemical reductions. We recall the definition of chemical reduction failure and the safety theorem as follows:

Definition 1 (Chemical reduction failure). We say a chemical solution $\mathcal{D} \Vdash \mathcal{P}$ is a chemical reduction failure, if one of the following holds:

- **Free this:** $(\mathbf{this} \mapsto \perp) \# P \in \mathcal{P}$ (briefly $P \in \mathcal{P}$), and $\mathbf{this} \in \text{fv}[P]$.
- **Undefined object name:** $\phi \# P \in \mathcal{P}$, $x \in \text{fv}[P]$ or $\phi = (\mathbf{this} \mapsto x)$, and x is not defined in $\mathcal{D}$.
- **Failed privacy:** $\phi \# x.f(\tilde{u}) \in \mathcal{P}$, and $\phi \neq \Phi$.
- **Undefined channel name:** $\Phi \# x.l(\tilde{z}) \in \mathcal{P}$, $x.D \in \mathcal{D}$, and $l \notin \text{dl}[D]$.
- **Arity mismatch:** $\Phi \# x.l(\tilde{z}) \in \mathcal{P}$, $x.D \in \mathcal{D}$, $l(\tilde{y})$ appears in a join-pattern of D , and $\tilde{y}$ and $\tilde{z}$ have different arities.

Theorem 3 (Safety w.r.t. chemical reduction). Well-typed chemical solution is never a failure as defined in Definition 1.

Proof. We assume $\vdash (\mathcal{D} \Vdash \mathcal{P})$. By typing rule TYPE-SOLUTION, we have:

$$A = \bigcup_{x.D \in \mathcal{D}} A_x \quad (1)$$

$$(A^{\mathcal{M}} \vdash x.D :: A_x)^{x.D \in \mathcal{D}} \quad (2)$$

We show that the chemical failures defined in Definition 1 are prevented.

No free this. We assume $P \in \mathcal{P}$. By typing rule TYPE-SOLUTION, we have $A^{\mathcal{M}} \vdash P$. Because $\mathbf{this}$ is not bound in $A^{\mathcal{M}}$, we have that $\mathbf{this}$ does not appear free in P .

No undefined object name. We assume $\phi \# P \in \mathcal{P}$. According to the value of ϕ , by rule TYPE-SOLUTION, we have $A \vdash P$, or $A^{\mathcal{M}} \vdash P$, or $A^{\mathcal{M}} + \mathbf{this} : \forall X. [\rho], \mathbf{this} : \forall X. (B \upharpoonright \mathcal{F}) \vdash P$. In either case, if $x \in \text{fv}[P]$, we always have that x is bound in A , hence $x.D \in \mathcal{D}$.

No privacy failure. We assume $\phi \# x.l(\tilde{u})$, and $\phi \neq \Phi$. Then according to typing rule TYPE-SOLUTION, no matter whether ϕ is empty or is $(\mathbf{this} \mapsto y)$, we always have that $x.l(\tilde{u})$ is typable under an environment where there is no assumption for private channels, hence we must have $l \in \mathcal{M}$.

No undefined channel. We assume $\Phi \# x.l(\tilde{z}) \in \mathcal{P}$ and $x.D \in \mathcal{D}$. We show $l \in \text{dl}[D]$. Assume $A_x = x : \forall X. [\rho], x : \forall X. (B \upharpoonright \mathcal{F})$. Then according to typing rule TYPE-DEFINITION, we have:

$$A + \mathbf{this} : [\rho], \mathbf{this} : (B \upharpoonright \mathcal{F}) \vdash D :: \zeta(\rho) B^{W, \emptyset} \quad (3)$$

$$\rho = B \upharpoonright \mathcal{M} \quad (4)$$

$$\text{dom}[B] = \overline{W} \quad (5)$$

Moreover, we also have $A \vdash x.l(\tilde{z})$, hence the following two premises:

$$A \vdash x.l :: \tilde{\tau}' \quad (6)$$

$$A \vdash \tilde{u} : \tilde{\tau}' \quad (7)$$

Depending on whether l is public or private, for (6) to hold, we must have that either ρ is of the form $[l : \tilde{\tau}; \dots]$, or $B \upharpoonright \mathcal{F}$ is of the form $(l : \tilde{\tau}; \dots)$, where $\tilde{\tau}'$ is an instance of $\tilde{\tau}$. Because $\rho = B \upharpoonright \mathcal{M}$, in both cases, we have $l \in \text{dom}[B]$, hence $l \in \overline{W}$ by (5), hence $l \in \text{dl}[D]$ by (3).

No arity mismatch. Following the previous case, we further assume that $l(\tilde{y})$ appears in one join-pattern of D . Hence the derivation of (3) must have a sub-derivation that looks like:

$$\frac{\text{TYPE-MESSAGE} \quad A' \vdash \tilde{y} : \tilde{\tau}}{A' \vdash l(\tilde{y}) :: (l :: \tilde{\tau})}$$

Namely, we have that the arity of $\tilde{y}$ is equal to the arity of $\tilde{\tau}$. From the previous case, we also know that the arity of $\tilde{u}$ is equal to the arity of $\tilde{\tau}'$, and $\tilde{\tau}'$ is an instance of $\tilde{\tau}$. Hence we have that $\tilde{y}$ and $\tilde{u}$ are of the same arity.

□



Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399