Information Hiding in the Join Calculus

Qin Ma[1] and Luc Maranget[2]

[1] OFFIS, Escherweg 2, 26121 Oldenburg, Germany
`Qin.Ma@offis.de`
[2] INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
`Luc.Maranget@inria.fr`

Abstract. We aim to provide information hiding support in concurrent object-oriented programming languages. We study the issue both 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 π-calculus. In this extended abstract we focus on the class level and design a new hiding operation on classes, aimed at preventing part of parent classes from being visible in client (inheriting) classes. We define the formal semantics of our new operation in terms of α-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.
Our motivation stems from language design, aiming at producing a practical programming language.

1 Introduction

Object-oriented concepts are often claimed to handle concurrent systems better. On the one hand, objects, exchanging messages while managing their internal states in a private fashion, model a practical view of concurrent systems. On the other hand, classes, supporting modular and incremental development, provide an effective way of controlling concurrent system complexity. Numerous fundamental studies [17, 2, 12, 23, 19, 18, 25, 22], propose calculi that combine objects and concurrency. By contrast, combining classes and concurrency faces the well-known obstacle of *inheritance anomalies* [16], *i.e.*, traditional overriding mechanism from sequential settings falls short in handling synchronization behavior reuse during inheritance. Fournet *et al.* have made a significant progress in this direction with their work [11]. They supplement the Join calculus [7, 8] with objects and classes, of which the main novelty is a class operation for both behavioral and synchronization inheritance, called *selective refinement*.

However, Fournet *et al.*'s model suffers from several limitations. Briefly, their type system is counter-intuitive and significantly restricts the power of selective refinement. In prior work [14], we improved Fournet *et al.*'s model by designing a new type system, where complete synchronization behavior is included in class types. As a consequence, our class calculus is easier for programmers to understand, and our typing system accepts more programs.

Our enriched class types carry more information. Thus, they make even more visible the lack of abstraction in class types, which was already present in [11]. More specifically, it is unlikely that two different classes possess the same type. This situation hinders *information hiding*, a key issue while programming in the large. Generally, information hiding allows the separation between a restricted interface (which we assimilate to types) and implementation. This principle brings advantages, such as removing irrelevant details from interfaces and protecting critical details of the implementation. As regards objects, one can easily hide some components by declaring them to be *private*, as Fournet *et al.* and many others do. These private components do not appear in object interfaces. By contrast, information hiding in classes is more involved, especially in the presence of synchronization inheritance. We are aware of no work on this issue. Specifically, if 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, the simple privacy policy applies solely to object users while always leaving full access to inheritance users.

In this paper, we address the issue of information hiding towards inheritance users. We do so by introducing a new explicit hiding operation in the class language. This amounts to significant changes in both the semantics and the typing of class operations. Basically, we perform the hiding of names (class components) by alpha-converting them to fresh names, while, by contrast, hidden names disappear from class types. It is important to notice that, for the sake of simplicity in semantics, only names already private to object users can be hidden to class users. We believe that our proposal achieves a reasonable balance of semantical simplicity and expressiveness, and that it yields a practical level of abstraction in class types, while preserving safety. Moreover, our suprisingly simple idea of hiding by alpha-conversion should apply equally well to other class-based systems, provided they rely on structural typing as we do.

The rest of this paper is organized as follows. First, we informally present our calculus in Sect. 2. We sketch the privacy policy for object users and introduce the hiding mechanism for inheritance users. Formal syntax and class semantics appear next in Sect. 3. Finally, Sect. 4 provides a ML-style type system. We state the standard soundness property in Sect. 5, as well as some other specific properties concerning hiding. To meet the page limitation, we focus ourselves mainly on the new hiding operation, while making available a complementary technical report [15] for complete details.

2 Classes, objects, and hiding

Basic class definition consists of a *join definition* and an (optional) **init** process, called *initializer* (analog to *constructors* or *makers* in other languages). As an example, we define the following class for one-place buffers:

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

```
   init this.Empty()
```
and instantiate an object from it:

```
obj buffer = c_buffer
```

Similar to Join, four *channels* are collectively defined in this example and arranged in two *reaction rules* disjunctively connected by **or**. We use the two channels `put` and `get` for the two possible operations, and the two channels `Empty` or `Some` for the two possible states of a one-place buffer, namely, being empty or full. We here follow Fournet *et al.*'s convention to express privacy: channels with capitalized names (*aka* labels) are *private*; they can be accessed only through self references; and the privacy policy is enforced statically.

Each reaction rule consists of a *join pattern* and a *guarded process*, separated by ▷. When there are messages pending on all the channels in a given pattern, the object can react by consuming the messages and triggering the guarded process. As a result, this one-place buffer behaves as expected: the (optional) **init** process initializes the buffer as empty; we then can put a value when it is empty, or alternatively retrieve the stored value when it is full.

By contrast with Join— whose values are the channels, the objects now become the values of the calculus. The basic operation of the calculus remains asynchronous message sending, but expressed in object-oriented dot notation, such as in process `buffer.put(n,r)`. Also note that we use the keyword **this** for recursive "self" references (*aka* self-inflicted references in Obliq [4]), while other references are handled through object names. Compared with the design in [11], this modification significantly simplifies the privacy control in object semantics.

2.1 Inheritance and hiding

In addition to basic classes, two operations are provided in [11] to support inheritance: *disjunction* to combine class definitions, and *selective refinement* to perform term rewriting on existing reaction rules.

At the moment, all labels defined in a class are visible during inheritance. However, this complete knowledge of class behavior may not be necessary for building a new class by inheritance. Moreover, exposing full details during inheritance sometimes puts program safety at risk, and designers of parent classes may legitimately wish to restrict the view of inheritance users.

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_hidden_buffer
  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 (inner) class `c_join` serves the purpose of consuming two acknowledgments from the previous one-place buffer and of 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.

3 Formalism: the OJoin$_{\mathcal{H}}$ calculus

We formalize our improved object-oriented extension of Join as the OJoin$_{\mathcal{H}}$ calculus, given in Fig. 1. We assume three disjoint sets of identifiers: for class names $c \in \mathcal{C}$, for object names $x, y, z, o \in \mathcal{O}$, and for labels $l \in \mathcal{L}$. In general, we write u for either a name from $\mathcal{O}$ or the keyword **this**. Tuples are written $\tilde{u}$. For privacy purpose, the set of labels $\mathcal{L}$ is further partitioned into disjoint subsets, with $f \in \mathcal{F}$ for private labels and $m \in \mathcal{M}$ for public labels. Additionnaly, $\mathcal{F}$ includes the set of hidden labels, written $h \in \mathcal{H}$

Objects are created from classes, where classes are defined from a full variety of constructs: reaction rules, disjunction, selective refinement, hiding, *etc.* However, class operations are only for the purpose of incremental class definitions. They cannot be used directly for object instantiations. As a consequence, we must resolve those operations into basic class definition, *i.e.* join definitions (denoted by D) together with an optional initializer, by means of class evaluation. In general, the semantics of the class language evaluates class definitions into *class normal forms*: (D **or** L) **init** P_v. We write P_v explicitly for processes that only involve plain object definitions. The additional part $L \subseteq \mathcal{L}$ denotes a set of *abstract* labels that are declared but not defined in D. Classes with a nonempty L part are called abstract and should not be instantiated.

In the following, we devote ourselves only to the hiding related semantical issues. Complete description of the operational semantics can be found in [15].

3.1 The semantics of hiding

How to hide labels? In addition to erasing hidden labels from class types, hiding also involves semantical operations on classes, operations whose design is

$P, Q ::=$		**Processes**
	0	null process
	$x.M$	message sending
	$\mathbf{this}.M$	recursive message sending
	$P_1 \;\&\; P_2$	parallel composition
	$\mathbf{obj}\ o = C\ \mathbf{in}\ P$	object definition
	$\mathbf{class}\ c = C\ \mathbf{hide}\ F\ \mathbf{in}\ P$	class binding with hiding
$C ::=$		**Classes**
	c	class name
	L	abstract class
	$M \triangleright P$	reaction rule
	$C_1\ \mathbf{or}\ C_2$	disjunction
	$\mathbf{match}\ C\ \mathbf{with}\ S\ \mathbf{end}$	selective refinement
	$C\ \mathbf{init}\ P$	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

Fig. 1. Syntax of the OJoin$_{\mathcal{H}}$ calculus

governed by two concerns. On one hand, hidden labels disappear. For instance, redefining a new label homonymous to a previously hidden label yields a totally new label. On the other hand, hidden labels still exist. For instance, objects created by instantiating the class `c_hidden_buffer` from Sect. 2.1 must somehow possess labels to encode the state of a one-place buffer.

The formal evaluation rule for hiding appears as follows:

$$\textsc{Eval-Hide}\quad \frac{\Gamma \vDash C \Downarrow_{\mathsf{C}} C_v \qquad (f_i \text{ defined in } C_v, h_i \text{ fresh})^{\,i\in I} \qquad \Gamma + (c \mapsto C_v\{h_i/f_i{}^{\,i\in I}\}_{\mathcal{H}}) \vDash P \Downarrow_{\mathsf{P}} P_v}{\Gamma \vDash \mathbf{class}\ c = C\ \mathbf{hide}\ \{f_i{}^{\,i\in I}\}\ \mathbf{in}\ P \Downarrow_{\mathsf{P}} P_v}$$

Here judgment $\Gamma \vDash C \Downarrow_{\mathsf{C}} C_v$ evaluate classes, and $\Gamma \vDash P \Downarrow_{\mathsf{P}} P_v$ evaluate processes, under the environments Γ that bind class names to class normal forms.

Hiding applies only to class normal forms, and only at class binding time. The hiding procedure $\{h_i/f_i{}^{\,i\in I}\}_{\mathcal{H}}$ is implemented by α-converting the hidden channels $\{f_i{}^{\,i\in I}\}$ to fresh labels $\{h_i{}^{\,i\in I}\}$. Fig. 2 defines α-conversion formally, where we simply write $\sigma_{\mathcal{H}}$ for $\{h_i/f_i{}^{\,i\in I}\}_{\mathcal{H}}$ in inductive cases. Such an α-conversion applies to both definition occurrences (in join patterns) and reference occurrences (in guarded processes and in the **init** process) of the hidden names in the normal form. It is important to notice that this simple mechanism can only handle mes-

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

Fig. 2. α-converting hidden names to fresh names in class normal forms

sages sent through self-inflicted references. As a consequence, we require hidden labels to be private, a condition which is checked by typing, in order to guarantee the renaming of *all* occurrences of hidden labels. Moreover, we do not rename under nested object definitions because they rebind **this**. To give some intuition, the normal form of class `c_hidden_buffer` from Sect. 2.1 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 that replace `Empty` and `Some` respectively.

This design meets the two concerns described at the beginning of this section: on the one hand, freshness guarantees hidden names not to be visible during inheritance; on the other hand, hidden names are still present in class normal forms but under fresh identities.

How to inherit a class with hidden labels? Another impact of hiding on semantics manifests itself during class inheritance. Classes are inherited by their names, which is rendered in semantics by substituting class normal forms for class names. However, to avoid two subclasses of the same class overlapping on the hidden labels, class normal forms should not be used directly. Instead, we require the re-freshening of the hidden names whenever a class is substituted.

Eval-CName
$$
\frac{\Gamma(c) = C_v \qquad \{h_i{}^{i\in I}\} = \mathsf{dl}[C_v] \upharpoonright \mathcal{H} \qquad (h'_i \text{ fresh})^{\,i\in I}}{\Gamma \vDash c \Downarrow_{\mathsf{C}} C_v\{h'_i/h_i{}^{i\in I}\}_{\mathcal{H}}}
$$

Where $\mathsf{dl}[C_v] \upharpoonright \mathcal{H}$ stands for the set of those labels defined by C_v that are hidden.

$\tau ::=$	$\alpha \mid [\rho]$	**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**

Fig. 3. Syntax of the type algebra

4 The typing of hiding

We define a ML-style type system to type objects and classes, focussing on the new hiding operation.

4.1 Type algebra

The grammar of type expressions appears in Fig. 3. 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 X to range over sets of type variables. 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, the set of *dangerous type variables*, soon. Finally, W reflects 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. Intuitively, by 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 Sect. 2 is:

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

As detailed in [14, 10, 5], 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 Sect. 2.1 hides labels `Some` and `Empty` and has type:

class `c_hidden_buffer`: **object**
  label `get`: `([reply: (`θ`); `ϱ`])` ;
  label `put`: `(`θ`,[reply: (); `ϱ'`])` ;
end $W = \{\{\texttt{get}\}, \{\texttt{put}\}\}$

The two hidden labels disappear from both the label list and W. Without other restriction, this type would allow 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. In this example, the complete type of class `c_hidden_buffer` looks like:

class `c_hidden_buffer`: **object**
  label `get`: `([reply: (`θ`); `ϱ`])` ;
  label `put`: `(`θ`,[reply: (); `ϱ'`])` ;
end $W = \{\{\texttt{get}\}, \{\texttt{put}\}\}\ V = \{\theta\}$

We shall now discuss formally the collection, maintenance, and usage of dangerous type variables.

4.2 Typing rules

We focus on the typing rules related to hiding, and do not elaborate on rules for simple processes [10], nor on rules for old class operations [14]. All those appeared in previous work and are available in [15].

How to type hiding? Hiding occurs while binding classes to names, the rule for hiding is thus an elaboration over the rule for class binding in [14]:

$$\textsc{Type-Hide}\quad \frac{\begin{array}{ll} A + \textbf{this} : [\rho]; \textbf{this} : (B \upharpoonright \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V} & \rho = B \upharpoonright \mathcal{M}; \varrho \\ B' = B \setminus F & W' = W \setminus F \\ A + c : \forall \mathsf{Gen}(\rho, B', A).\zeta(\rho){B'}^{W', V \cup \mathsf{ftv}[B \upharpoonright F]} \vdash P & F \subseteq \overline{W} \end{array}}{A \vdash \textbf{class}\ c = C\ \textbf{hide}\ F\ \textbf{in}\ P}$$

Here judgment $A \vdash C :: \tau^c$ types classes, and $A \vdash P$ types processes, under the typing environments A that bind class names, object names, or **this** to corresponding polymorphic types.

The class definition is first typed, under the extended environment A with two complementary bindings for the recursive self reference — **this** : $[\rho]$ (where $\rho = B \upharpoonright \mathcal{M}; \varrho$) for public labels, and **this** : $(B \upharpoonright \mathcal{F})$ for private ones. Here $B \upharpoonright \mathcal{M}$ and $B \upharpoonright \mathcal{F}$ restrict the domain of B respectively to public and private labels. Note that with the same B appearing on both sides of the judgment, the compatibility between the defined types of labels (on the right side) and the expected types of labels (on the left side) is explicitly expressed. Moreover, as a side note, the condition $\rho = B \restriction \mathcal{M}; \varrho$ also allows us to merge the ρ component into the B component in the presentation of our class type examples.

$F \in \mathcal{F}$ stands for the set of hidden names. With $\overline{W}$ listing all the labels defined by class C, the condition $F \subseteq \overline{W}$ in the premise checks whether all the hidden labels are actually defined, in particular abstract labels cannot be hidden. Hiding has no impact on the type of objects created from the class (object user interface $[\rho]$), because ρ contains only public labels. However, the interface for inheritance users, namely B^W is restricted to ${B'}^{W'}$, where $B' = B \setminus F$, and $W' = W \setminus F$. $W \setminus F$ 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 ($\mathsf{ftv}[B \restriction F]$) are considered as dangerous, and are appended into V. One might wonder why not only add those correlated free type variables that disappear from the hidden interface, namely $\mathsf{ctv}[B^W] \setminus \mathsf{ctv}[{B'}^{W'}]$, where $\mathsf{ctv}[B^W]$ computes the free type variables in B that are common to types of at least two correlated channels according to W. At first glance, it seems feasible and would allow more polymorphism. Unfortunately, we cannot because this would not be safe (See [15, page 22]).

With the notion of dangerous type variables, the polymorphism control of object types elaborates into two parts as in the following rule for typing objects:

$$\textsc{Type-Object}\quad \frac{\begin{array}{ll} A + \mathbf{this} : [\rho]; \mathbf{this} : (B \restriction \mathcal{F}) \vdash C :: \zeta(\rho)B^{W,V} & \\ X = \mathsf{Gen}(\rho, B, A) \setminus \mathsf{ctv}[B^W] \setminus V & \rho = B \restriction \mathcal{M} \\ A + x : \forall X.[\rho] \vdash P & \mathsf{dom}[B] = \overline{W} \end{array}}{A \vdash \mathbf{obj}\ x = C\ \mathbf{in}\ P}$$

In addition to the common free type variables that are shared by correlated channels (*i.e.* computed by $\mathsf{ctv}[B^W]$), the set of dangerous type variables V is also prevented from generalization.

How to type classes with hidden labels? Not surprisingly, the set of typing rules for classes are almost kept the same as before in [14], except for the transmission of the set V. Dangerous type variables only come from hiding at class binding time (*i.e.* outside class definitions). However, for the sake of safe polymorphism, they should be preserved during class operation. Moreover, we have the following rule to type class names:

$$\textsc{Type-CName}\quad \frac{c : \forall X.\zeta(\rho)B^{W,V} \in A}{A \vdash c :: \eta(\zeta(\rho)B^{W,V})}$$

When making an instants of the polymorphic class type by a substitution η for type variables, we define $\eta(\zeta(\rho)B^{W,V}) = \zeta(\eta(\rho))\eta(B)^{W,\eta(V)}$ where:

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

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

5 Properties of the Type System

5.1 The type system is sound

One of the main goals of static typing is to exclude programs that will cause errors at runtime. Our type system is sound as stated in the following theorem.

Theorem 1 (Soundness). *Let P be a process (with class definitions). If P is well-typed, then P evaluates to P_v (without class definitions), and P_v is not an error. Moreover, putting P_v in the chemical machine, the chemical reduction of the initial solution and all the successive ones never fail.*

As can be seen from the theorem formulation, our class semantics includes "error rules" to model failed class evaluation. Thus, for instance, the type system guarantees: no unbound class names; no attempts to instantiate from abstract classes; *etc*.

Although standard is the type soundness, the proof technique is non-trivial. Briefly, the use of big-step evaluation semantics demands to type evaluation environments polymorphically. Moreover, the impact of hiding on semantics and on type system differ significantly — *i.e.* α-conversion against erasing. To bridge the gap, we introduce the new notion of *revealing*, which intuitively reflects the reverse idea of hiding. Technically, revealing is expressed as an ordinary, specific, typing rule, but not as a subtyping relation. In addition, checking against the set of dangerous type variables in class types also plays an important role.

5.2 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 $\mathtt{c}_1$ = `a()` ▷ **this**.`Ch()` **or** `Ch()` ▷ P **hide** {`Ch`}
class $\mathtt{d}_1$ = $\mathtt{c}_1$ **or** `b()` ▷ **this**.`Ch()`

Although channel `Ch` is defined in class $\mathtt{c}_1$, it is hidden. Hence in the derived class $\mathtt{d}_1$, `Ch` is not accessible. The type system will report a "unbound name" error when typing class $\mathtt{d}_1$.

Moreover, once a name is hidden, it is reasonable to define an unrelated channel that happens to have the same name, such as the following code:

class $\mathtt{c}_2$ = `a(x)` ▷ **this**.`Ch(x)` **or** `Ch(i)` ▷ `out.print_int(i)` **hide** {`Ch`}
class $\mathtt{d}_2$ = $\mathtt{c}_2$ **or** `Ch(s)` ▷ `out.print_string(s)` **or** `b(x)` ▷ **this**.`Ch(x)`

Both classes $\mathtt{c}_2$ and $\mathtt{d}_2$ are well-typed despite the fact that the two definitions of label `Ch` have incompatible types and behaviors: one receives and prints an integer and the other receives and prints a string. In addition, our hiding semantics guarantees that late-binding is not applicable during the inheritance. Suppose `o` is an object of class $\mathtt{d}_2$. Message `o.a(x)` is actually consumed by calling the fresh name corresponding to the hidden `Ch`, which requires `x` be an integer, and will print out this integer. In comparison, `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 $\mathtt{c}_2$ as follows:

class $\mathtt{c}_2' = \mathtt{a(x)} \triangleright$ **this**.$\mathtt{A'(x)}$ **or** $\mathtt{A'(i)} \triangleright \mathtt{out.print\_int(i)}$ **hide** $\{\mathtt{A'}\}$

Then both $\mathtt{c}_2$ and $\mathtt{c}_2'$ are well-typed, and provide the same interfaces to their users. Replacing $\mathtt{c}_2$ by $\mathtt{c}_2'$ in the definition of class $\mathtt{d}_2$ makes no difference.

6 Related work

The idea of using join patterns for class synchronization abstraction in object-oriented programming is also followed by other language designers, such as the authors of polyphonic C$^\sharp$ [1] and Join Java [13]. However, classes in those only support limited inheritance of Join abstractions. Fournet *et al.* study this problem based on a theoretical foundation in [11]. 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 [14] 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 [11], on the other hand it allows more degree of type abstraction than in [14].

Our hiding mechanism is inspired by the design of its counterpart for sequential classes in OCaml, which is not described in its theoretical calculus [20] but is present in its real system. Briefly, in the sequential case, hiding amounts to freezing method names, while our extension additionaly performs a similar action on synchronizations policies defined by a class. From typing point of view, hiding method names in OCaml also amounts to removing the hidden names from class types. However, hiding in OCaml (and in MOBY [6]) is performed implicitely by specifying restricted class types. Given the sophisticated class types of our class language, 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 [6]. 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* [3]. As observed by Rémy and Vouillon [20, 24], and also by Fisher and Reppy [6], 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. However, notice that Vouillon proposes a comprehensive solution [24]. All those works [6, 24] use Riecke-Stone style *dictionaries* [21] to capture the dynamic semantics of hiding. It is worth noticing that dictionaries, which basically are bindings from labels to labels, appear explicitely in these cited calculi, moreover also in the language of [24], thereby adding significant complexity. Compared with their approach, our semantics of hiding by α-conversion is simpler, and sufficient for the purpose of hiding private names only.

7 Conclusion

We have 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 an additional 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 [20].

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 safe polymorphism. 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, through non-trivial proof technique. Besides the standard soundness property, we informally argued that our mechanism deserves the name "hiding".

We have achieved significant improvements over the original design of Fournet *et al.* [11]: in [14] as regards the class system expressiveness, and in 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 [9]. We have not yet performed this extension, but rather wrote a prototype system from which we draw some precise implementation guidelines [15]. More precisely, we explain how to perform class operations in a separate compilation setting and how to implement message sending to objects over ordinary message sending in Join with a small additional price.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.

2. P. D. Blasio and K. Fisher. A calculus for concurrent objects. In *Proceedings of CONCUR'96*, pp.655–670, 1996.
3. 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.
4. L. Cardelli. Obliq: A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.
5. G. Chen, M. Odersky, C. Zenger, and M. Zenger. A functional view of join. Technical Report ACRC-99-016, University of South Australia, 1999.
6. K. Fisher and J. Reppy. The design of a class mechanism for MOBY. In *Proceedings of PLDI'99*, pp.37–49, 1999.
7. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Nov. 1998.
8. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pp.372–385, 1996.
9. C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. The JoCaml system. Software and documentation available at `http://pauillac.inria.fr/jocaml`, 2001.
10. C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of CONCUR'97*, pp.196–212, 1997.
11. 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.
12. A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *Proceedings of HLCL'98*, 1998.
13. G. S. Itzstein and D. Kearney. Join Java: An alternative concurrency semantics for Java. Technical Report ACRC-01-001, University of South Australia, 2001.
14. Q. Ma and L. Maranget. Expressive synchronization types for inheritance in the join calculus. In *Proceedings of APLAS'03*, pp.20–36, 2003.
15. Q. Ma and L. Maranget. Information hiding, inheritance and concurrency. Inria Rocquencourt Research Report RR-5631, 2005. `http://pauillac.inria.fr/~maranget/papers/hide-tr.ps`.
16. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. *Research Directions in Concurrent Object-Oriented Programming*, pp.107–150. MIT Press, 1993.
17. O. Nierstrasz. Towards an object calculus. In *Proceedings of ECOOP'91 Satellite Workshop on Object-Based Concurrent Computing*, pp.1–20, 1991.
18. M. Odersky. Functional nets. In *Proceedings of ESOP'00*, pp.1–25, 2000.
19. B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Proceedings of TPPP 94*, pp.187–215, 1995.
20. 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.
21. J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 172(1):2–28, 2002.
22. D. Sangiorgi. An interpretation of typed objects into typed π-calculus. *Information and Computation*, 143(1):34–73, 1998.
23. V. T. Vasconcelos. Typed concurrent objects. In *Proceedings of ECOOP'94 Workshop on Object-Based Concurrent Computing*, pp.100–117, 1994.
24. J. Vouillon. Combining subsumption and binary methods: an object calculus with views. In *Proceedings of POPL'01*, pp.290–303, 2001.
25. D. J. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.