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 this issue both at the object level and the class level, in the context of an object-oriented extension of Join— 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. The purpose of this operation is to prevent part of parent classes from being visible in client (inheriting) classes. We define its formal semantics in terms of α-converting hidden names to fresh names, and its typing in terms of eliminating hidden names from class types.

1 Introduction

Object-oriented concepts are often claimed to handle concurrent systems better. On 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 such as [6, 11] proposed calculi that combine objects and concurrency. By contrast, combining classes and concurrency faces the well-known obstacle of *inheritance anomalies* [9, 10], *i.e.*, traditional overriding mechanism from sequential settings falls short in handling synchronization behavior reuse during inheritance. Recently, Fournet *et al.* have proposed a promising solution to this problem [5]. The main idea is to extend the Join calculus [3] with objects and classes, and more importantly to design a novel class operation for both behavioral and synchronization inheritance, called *selective refinement*.

However, Fournet *et al.*'s model still suffers from several limitations, mainly in typing. Briefly, their type system is counter-intuitive and significantly restricts the power of selective refinement. In prior work [7], we tackled this problem by designing a new type system. We mainly enriched class types with complete synchronization behavior to exploit the full expressiveness of selective refinement. However, doing so inevitably impaired the other dual role of class types, *i.e.* abstraction. More specifically, it was unlikely for two different classes to possess the same type. How we can regain abstraction becomes a subsequent interesting question. We manage to achieve this goal by enabling programmers with *information hiding* capability in this paper.

Information hiding by itself is already a key issue in large-scale programming. Generally, information hiding allows programmers to decide what to export in the interface (which we assimilate to types) of an 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. The difficulty resides in that synchronization introduces certain type dependency among names, while carelessly hiding some of them would result in unsafe typing. 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.

We address the issue of information hiding towards inheritance users in this paper. 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. Theoretically, hiding a name in a class can be expressed as quantifying it existentially. In practice, we α-convert hidden names to fresh names in the operational semantics and remove hidden names from class types in typing. 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 surprisingly simple idea of hiding by α-conversion should apply equally well to other class-based systems, provided they rely on structural typing as we do.

In this short paper, we focus on intuition, while making available a complementary technical report [8] for complete formalism.

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 are *private*; they can be accessed only through recursive self references; and the privacy policy is enforced statically.

Each reaction rule consists of a *join pattern* and a *guarded process*, separated by ▷. Join patterns specify the synchronization among channels. Namely, only if 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 channels, objects now become the values of the calculus. As an important consequence, channel names are no longer governed by the usual rules of lexical scoping (objects names are). Channel names can be seen as global, as method names are in any simple object calculi. From now on, channel names are called *labels*.

The basic operation of our calculus is asynchronous message sending, but expressed in object-oriented dot notation, such as in process `r.reply(n)`, which stands for "send message `n` to the channel `reply` of object `r`". Also note that we use the keyword **this** for recursive self references, while other references are handled through object names. Compared with the design in [5], this modification significantly simplifies the privacy control in object semantics.

2.1 Inheritance and hiding

Inheritance is basically performed by using, *i.e.* computing with, parent classes in derived classes. 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 channel `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 channels. 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 can still supplement one-place buffers with a put2 operation as follows:

```
class c_put2_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.

2.2 Hiding only private channels

We here restrict our hiding mechanism only to private channels. Such a decision originates in the problems between hiding public channels and supporting advanced features, such as *selftype* (also known as *mytype*) and *binary methods* [1]. As observed in [13, 2], these two aspects do not trivially get along without endangering type soundness. More specifically, a problem manifests itself when selftype is assumed outside the class and we hide a public channel afterwards. As an example, consider the following class definitions.

```
class c0 = f(x) ▷ x.b(1)
class c1 = a() ▷ obj x = c0 in x.f(this)
        or b(n) ▷ out.print_int(n)
```

Channel f of class c_0 expects an object with a channel b of type integer. This condition is satisfied when typing the guarded process of channel a in class c_1, because the self object **this** does have a channel b of type integer. However, later inheritance may hide the channel b (in class c_2), and then define a new channel also named b but with a different type string (in class c_3).

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

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

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

A simple solution adopted in the community is not to support both. Following OCaml, we choose to support the notion of selftype and limit hiding to private channels. By contrast, Fisher and Reppy in their work for MOBY [2] choose the reverse: not to provide selftype and instead provide complete control over class-member visibility. Nevertheless, a more comprehensive solution is still possible [13], however, more complicated as well.

3 The semantics of hiding

Class semantics is expressed as the rewriting of class-terms, while object semantics by the means of reflexive chemical machines [3]. Class reductions always terminate and produce *class normal forms*, which are basically object definitions, plus an (optional) initializer, plus a a list of abstract labels. In cases where the latter is empty (which can be statically controlled by our type systems), objects can be created from such class definitions in normal form. Hence, our evaluation mode is a stratified one: first rewrite classes to object definitions; then feed the resulting term and an initial input into a chemical abstract machine.

How to hide labels? The semantics of hiding in classes is 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:

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

The above inference rule is part of the class reduction semantics (see [8]). Judgments express the reduction of classes to class normal forms, under an environment Γ that binds class names to class normal forms (call-by-value semantics).

Hiding applies only to class normal forms (C_v), 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 $\{{h_i}^{\,i\in I}\}$, whose definition is without surprise. Such a semantics makes sense because labels are *not* scoped. The α-conversion should apply to both definition occurrences (in join patterns) and reference occurrences (in guarded processes and in the **init** process) of the hidden labels in the normal form. Thanks to the restriction to only hide private labels, the recursive self references in the normal form already include *all* the reference occurrences of hidden labels. Moreover, we do not rename under nested object definitions because they re-bind **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 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.

4 The typing of hiding

4.1 Class types and object types, catching up

Types are automatically inferred. Following our prior work [7], a class type consists of three parts, written $\zeta(\rho)B^W$, where B lists the set of channels, defined or declared in the class, paired with the types of the messages they accept, and W reflects how defined channels are synchronized, *i.e.* the structure of the join patterns in the corresponding class normal form. The row type ρ collects the public label-type pairs from B for the type of objects created from this class. To avoid repetition, in concrete syntax, ρ is usually incorporated in B that is enclosed between **object** and **end**, as in the type of class `c_buffer` from Sect. 2:

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`$\}\}$

We see that messages conveyed by channels are polyadic. The type of a channel carrying k objects of types $\tau_1, \ldots, \tau_k$ is written $(\tau_1, \ldots, \tau_k)$. Object types are always enclosed in square brackets. The type of objects of this class is:

$[$`get:`$([$`reply:`$(\theta); \varrho]); $`put:`$(\theta, [$`reply:`$(); \varrho'])]$

Channels `Some` and `Empty` do not show up because they are private. Finally, W is organized as a set of sets of labels. Two labels appear in the same member set of W if and only if they are synchronized in one join pattern.

Following ML type systems, polymorphism is parametric polymorphism, obtained essentially by generalizing free type variables. However, such generalization is controlled for object types. More specifically, any type variables that are shared by synchronized channels should not be generalized. Detailed rationale for doing so is discussed in all kinds of Join typing papers, such as [7, 4]. The basic reason is for type safety. As an example, type variable θ should not be polymorphic in the object type above, because following the class type it is shared by two synchronized channels `get` and `Some` (*i.e.* appearing in the same member set of W). Otherwise, its two occurrence in `get` and `put` could then be instantiated independently as, for instance, integer and string. This then would result in a runtime type error: attempting to retrieve a string when an integer is present. By contrast, θ is safely generalized in the class type, which allows us to create two objects from it, one dealing with integers, and the other with strings. The two trailing row variables ϱ, ϱ' are both generalizable. They can be instantiated as more label-type pairs, thus introducing a useful degree of subtyping polymorphism by structure.

4.2 How to type hiding: ideas

The most straightforward idea is to remove hidden names from class types. As a consequence, class `c_hidden_buffer` from Sect. 2.1 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 channels `Some` and `Empty` are eliminated from both the B list and W. Unfortunately, such a naïve elimination has a side-effect, which may endanger safe polymorphism in the corresponding object type. Plainly, the non-generalizable type variable θ has now falsely become generalizable, because according to this class type, the only two channels that share θ are not synchronized (*i.e.* `get` and `put` coming from two different member sets of W).

To tackle the problem, we then decide to keep track of such *dangerous type variables* caused by hiding in class types, called V. More precisely, before eliminating, we first record all the non-generalizable type variables of hidden names in V. For this example, the type of class `c_hidden_buffer` then evolves to:

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

Right before hiding, the type variables in a hidden channel are of two kinds: non-generalizable or generalizable. The modification above solves perfectly the problem of losing information about non-generalizable ones. If type variables that are generalizable before hiding would always be kept so, we here already reach a working way of typing hiding. Unfortunately, it is not the case. Some generalizable type variables of hidden channels may later become non-generalizable during inheritance, even though the channels are already hidden. Consider the following class definition in which channel `Ch′` is hidden:

class $\texttt{c}_1 = \texttt{a(x)} \triangleright 0$ **or** `b(y)` & $\texttt{Ch}'(\texttt{n}_1, \texttt{n}_2) \triangleright$ **this**.(`a(`$\texttt{n}_1$`)` & `b(`$\texttt{n}_2$`)`)

The corresponding class type is:

class $\texttt{c}_1$: **object label** `a`: (θ); **label** `b`: (θ') **end** $\texttt{W} = \{\{\texttt{a}\},\{\texttt{b}\}\}$ $\texttt{V} = \{\theta'\}$

The hidden channel `Ch′` is of type (θ, θ'). According to the definition, θ' is non-generalizable (because shared by the synchronized channel `b`) thus is put in V. By contrast, θ is generalizable. However, the following inheritance of class $\texttt{c}_1$ easily convert θ into non-generalizable:

class $\texttt{c}_2 =$ **match** $\texttt{c}_1$ **with** `b(y)` $\Rightarrow$ `b(y)` & `d(z)` $\triangleright$ **this**.`a(z)` **end**

This selective refinement operation mainly replaces "`b(y)`" by "`b(y)` & `d(z)`" in join patterns and composes the corresponding guarded processes with "**this**.`a(z)`" in parallel. As a consequence, class $\texttt{c}_2$ has the following normal form:

class $\texttt{c}_{v2} = \texttt{a(x)} \triangleright 0$
or `b(y)` & `d(z)` & $\texttt{Ch}'(\texttt{n}_1, \texttt{n}_2) \triangleright$ **this**.(`a(`$\texttt{n}_1$`)` & `b(`$\texttt{n}_2$`)` & `a(z)`)

The new channel `d` is of type (θ). It synchronizes and shares θ with `Ch′`. However it is already too late to update the non-generalizable information to reflect this,

because hidden names are already eliminated from class types thus out of control of the type system. A simple solution we adopt is to treat already all the free type variable of hidden names as dangerous, non-generalizable and generalizable, in case the non-generalizable ones increase. To sum up, the final type of class $\mathtt{c}_1$ is:

class $\mathtt{c}_1$: **object label** $\mathtt{a}$: (θ); **label** $\mathtt{b}$: (θ') **end** $\mathtt{W} = \{\{\mathtt{a}\},\{\mathtt{b}\}\}$ $\mathtt{V} = \{\theta, \theta'\}$

Formal discussion of the type system appears in the complementary technical report [8], including statement and proof of a "*soundness*" theorem.

5 Conclusion

We have achieved significant improvements over the original design of Fournet *et al.* [5]: in [7] as regards the class system expressiveness, and in this paper as regards visibility control, type abstraction, and simplification of runtime semantics. We claim that these improvements yield a calculus mature enough to act as the model of a full-scale implementation.

References

1. 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.
2. K. Fisher and J. Reppy. The design of a class mechanism for MOBY. In *Proceedings of PLDI'99*, pp.37–49, 1999.
3. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pp.372–385, 1996.
4. 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.
5. 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.
6. A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *Proceedings of HLCL'98*, pp.248–264, 1998.
7. Q. Ma and L. Maranget. Expressive synchronization types for inheritance in the join calculus. In *Proceedings of APLAS'03*, pp.20–36, 2003.
8. Q. Ma and L. Maranget. Information hiding, inheritance and concurrency. Inria Rocquencourt Research Report RR-5631, 2005.
9. 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.
10. G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proceedings of SAC'96*, pp.1267–1274, 2004.
11. M. Odersky. Functional nets. In *Proceedings of ESOP'00*, pp.1–25, 2000.
12. J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 172(1):2–28, 2002.
13. J. Vouillon. Combining subsumption and binary methods: an object calculus with views. In *Proceedings of POPL'01*, pp.290–303, 2001.