Expressive Synchronization Types for Inheritance in the Join Calculus

Qin Ma and Luc Maranget

Inria Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
{Qin.Ma, Luc.Maranget}@inria.fr

Abstract. In prior work, Fournet *et al.* proposed an extension of the join calculus with class-based inheritance, aiming to provide a precise semantics for concurrent objects. However, as we show here, their system suffers from several limitations, which make it inadequate to form the basis of a practical implementation.
In this paper, we redesign the static semantics for inheritance in the join calculus, equipping class types with more precise information. Compared to previous work, the new type system is more powerful, more expressive and simpler. Additionally, one runtime check of the old system is suppressed in the new design. We also prove the soundness of the new system, and have implemented type inference.

1 Introduction

Object-oriented programming is an attractive framework for taming the complexity of modern concurrent systems. In the standard approach, as in Java [11], ABCL/1 [20] or Obliq [4], objects represent agents that may exchange messages over a network. These languages usually model concurrency by threads and control synchronization behavior with locks (or some variant thereof). However, combining concurrency and object-oriented programming as if they were independent hinders incremental code development by inheritance [15]. Roughly, objects follow synchronization policies to keep their internal state consistent in the presence of concurrent method calls, but traditional inheritance is not sufficient to program these policies incrementally.

In [9], Fournet *et al.* address this issue. They define the *objective-join* calculus, aiming to provide a precise semantics for concurrent objects. Doing so, they follow many authors, who either enrich well-established object calculi with concurrency primitives [10,2,16], or extend process calculi with object-oriented features [19]. The base calculus of [9] is the *join calculus* [7], a simple name-passing calculus, related to the pi-calculus but with a functional flavor. Objects are introduced by the classical technique of objects-as-records (of names). This technique is particularly natural in the join calculus, where several names and their behavior with respect to synchronization are defined simultaneously, via *join-definitions*. Then, [9] introduces a class layer to support incremental programming and code reuse.

However, there are several limitations to the theory presented in [9] which make it inadequate to form the basis of a practical implementation. At first glance, class types look close to their sequential counterparts, namely, lists of channel names with the types of the messages they accept. Although this information is sufficient for typing objects, whose basic operation is message receiving, it falls short in typing class-based inheritance of synchronization behavior. As a consequence, [9] in fact includes some extra information about synchronization behavior in class types. Unfortunately, the extra information is not expressive enough, leading to the following shortcomings: objects sometimes are not as polymorphic as they could; names flagged as abstract by typing may actually possess a definition; and, under particular conditions, a complicated runtime check on classes is necessary to guarantee program safety. The present paper solves these difficulties with a new type system for classes, of which the main contribution is the inclusion of complete synchronization behavior in class types.

The paper is organized as follows. We give a brief review of the object and class calculus in Sec. 2. Then, from Sec. 3 to Sec. 5, we illustrate the shortcomings of the former system, and meanwhile suggest possible remedies. Sec. 6 presents our solution —a new design of the static class semantics— and as well states the properties of subject reduction and type safety though omits the lengthy proofs (given in [14]). Finally, we conclude.

2 A brief review of the objective-join calculus

2.1 Objects and classes

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

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

The basic operation is still asynchronous message passing, but expressed by object-oriented dot notation. Here, the process $\mathit{buffer}.\mathit{put}(n,r)$ sends a $\mathit{put}(n,r)$ message to the object *buffer*. In the message, *put* is the *label* and (n,r) is the *content*.

As in join, the labels defined in one object are organized by several *reaction rules* to specify how messages sent on these labels will be synchronized and processed. In the above *buffer* object, four labels are defined and arranged in two reaction rules. Each reaction rule consists of a *join-pattern* and a *guarded process*, separated by $\triangleright$. When there are messages pending on all the labels in a given pattern, the object can react by consuming the messages and triggering the guarded process. Given this synchronization mechanism, *buffer* will behave as follows:

- If *buffer* is empty (*i.e.* an *Empty*() message is pending), and a put attempt is made (by sending a $put(n,r)$ message), then *buffer* will react by sending back an acknowledgement message on label *reply* of the continuation r and, concurrently, shifting itself into the $Some(n)$ state.
- Symmetrically, the value n stored in *buffer* (*i.e.* a $Some(n)$ message is pending) can be retrieved by $get(r)$, and the buffer then returns 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 part initializes *buffer* as an empty buffer. See [1] for a more implementation-oriented description of a similar example.

As we can see, the state of *buffer* is encoded as a message pending on *Empty* or on *Some*. To keep the one-place buffer consistent, there should be only one message pending on either of these two labels. This invariant holds as long as no message on these labels can be sent from outside, which can be guaranteed by the following a statically enforced privacy policy: Labels starting with an uppercase letter are *private* labels; messages can be sent on those labels only from within guarded processes (and from the init part as well). Other labels are unrestricted; they are called *public* labels and work as ports offered by the object to the external world.

Classes act as templates for sets of objects with the same behavior. For instance, the class defining the one-place buffer above looks like this:

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

The prefix self (z) explicitly binds the name z to the self reference in the class definition. And to instantiate an object from the class, we just do:

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

The calculus of [9] allows the derivation of new class definitions from existing ones. A class can be extended with new reaction rules, synchronizations among the labels of a class can be modified, and new processes can be added to the reaction rules of a class. These computations on classes finally yield classes solely made of *disjunctions* (or) of several reaction rules (as is the *c_buffer* class). Such classes are ready for being instantiated and are called *normal forms*.

2.2 Types

Types for classes and objects are inspired by the OCaml type system [18]. More precisely, typing is by structure; both class types and object types are sets of label names paired with the types of the messages they accept; object types only collect public labels, while class types include both private and public labels. Object types may end with a row variable, standing for more label-type pairs and such row variables provide a useful degree of polymorphism.

For instance, consider a simple synchronization object *join* with an internal counter:

$$\text{obj } join = sync1(r1) \ \& \ sync2(r2) \ \& \ Count(x) \triangleright$$
$$r1.reply() \ \& \ r2.reply() \ \& \ join.Count(x{+}1)$$

The type of the object *join* is $[sync1 : ([reply : (); \varrho]); sync2 : ([reply : (); \varrho'])]$. The two row variables ϱ and ϱ' indicate that except for the *reply* label, there is no other constraints on the continuation objects sent to the *sync* labels. Furthermore, label *Count* does not show up because it is private. By contrast, the type of the corresponding *c_join* class contains all the labels:

object
  label $sync1$: $([reply\text{: } (); \ \varrho])$; label $sync2$: $([reply\text{: } (); \ \varrho'])$;
  label $Count$: (int)
end

However, during the design of the type system of [9], it appeared that class types should include additional information. We introduce them in the next two sections using examples.

3 Polymorphism

3.1 Polymorphism and synchronization

There are two kinds of type variables in the type system: ordinary type variables and row variables. As in the ML type system, polymorphism is parametric polymorphism, obtained essentially by generalizing the free type variables.

As in the join calculus, synchronizations impact on polymorphism [8,5]: two different labels are said to be *correlated* when they appear in the same join-pattern, and type variables that appear in the types of several correlated labels cannot be generalized. Consider, for instance, a synchronous buffer object:

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

The types of the two correlated labels *get* and *put* are $([reply : (\theta); \varrho])$ and $(\theta, [reply : (); \varrho'])$, respectively. Type variable θ is shared by their types, so it should not be generalized. Otherwise, the two occurrence of θ could be instantiated independently as, for instance, integer and string. This then might result in a runtime type error: attempting to deliver a string when an integer is expected. By contrast, the type variables ϱ and ϱ' can be generalized. Generalized row variable ϱ intuitively means that as long as the constraint on *reply* label is preserved, different messages sent on label *get* can take different continuation objects as r, and similarly for ϱ' with respect to label *put*.

However, the limitation on polymorphism is only object-level, not class-level, because two *get* or *put* labels of two different objects are completely independent *w.r.t.* synchronization. More concretely, consider the following class version of the synchronous buffer.

$$\text{class } c\_sbuffer = \ get(r) \ \& \ put(n,s) \triangleright r.reply(n) \ \& \ s.reply()$$

Class *c_sbuffer* can be instantiated twice to two different objects, one dealing with integers, and the other with strings. As a consequence, all the free type variables in class types are generalized, but class types record the correlation amongst labels, in order to restrict polymorphism at object creation time. To collect this information, the type system of [9] abstracts the join-patterns of a class into a set of *coupled* labels W. The computation of coupled labels from reaction rules can be described as a two-phase process:

1. First, simplify join-patterns by removing labels accepting empty tuple (because correlating with this kind of labels does not cause type variable sharing) and remove join-patterns consisting of only one label.
2. Then, gather all the remaining labels into the set of coupled labels.

The set W is then kept in the class type, and referred to when an object is to be created: all the type variables shared by several labels in W are restricted to be monomorphic in the corresponding object type. As a result, the type of the synchronous buffer class is as follows:

$$\begin{array}{l} \textsf{class}\ \ \textit{c\_sbuffer}\colon \forall\theta,\varrho,\varrho'.\ \textsf{object} \\ \quad\quad \textsf{label}\ \ \textit{get}\colon ([\ \textit{reply}\colon (\ \theta);\ \varrho])\ ; \\ \quad\quad \textsf{label}\ \ \textit{put}\colon (\theta,\ [\ \textit{reply}\colon ();\ \varrho']) \\ \quad \textsf{end}\ \ W = \{\textit{get},\ \textit{put}\} \end{array}$$

Here labels *get* and *put* are coupled, and hence the type variable θ is not generalizable in object types.

3.2 More polymorphism in object types

According to the previous section, the following object definition:

$$\begin{array}{l} \textsf{obj}\ \textit{counter} = \\ \quad\quad \textit{Value}(x)\ \&\ \textit{plus}(y{,}r) \triangleright r.\textit{reply}()\ \&\ \textit{counter}.\textit{Value}(x{+}y) \\ \textsf{or}\ \ \textit{Value}(x)\ \&\ \textit{incr}(r) \triangleright \textit{counter}.\textit{Value}(x)\ \&\ \textit{counter}.\textit{plus}(1{,}r) \end{array}$$

correlates the labels *plus* and *Value* by the first join-pattern, and the labels *incr* and *Value* by the second join-pattern (but not the labels *plus* and *incr*).

The types of the labels of this object are *Value*: $(\textsf{int})$, *plus*: $(\textsf{int}, [\textit{reply} : (); \varrho])$ and *incr*: $([\textit{reply} : (); \varrho])$. The free row variable ϱ appears in the types of both *plus* and *incr*, but since the two labels are not correlated, it can be generalized.

However, lifting the object definition *counter* into a class definition will yield the class *c_counter* with type:

$$\textsf{class}\ \ \textit{c\_counter}\colon \textsf{object} \ldots \textsf{end}\ \ W = \{\textit{Value},\ \textit{plus},\ \textit{incr}\}$$

Both *plus* and *incr* appear as coupled labels, which forbids the generalization of the shared free type variable ϱ in the type of the objects.

This undue restriction on polymorphism originates in the flat structure of the set W. Such a structure may couple labels that are not actually correlated, more precisely, the coupled label relation is the transitive closure of correlation amongst labels accepting non-empty tuples. In the *counter* example, labels *plus* and *incr* get "correlated" through *Value*.

However, if we replace the second step of the computation of the set W in Sec. 3.1 by collecting labels from different patterns in separate sets, thereby organizing W as the set of these sets, we may have a better solution. The criterion for restriction changes accordingly: for now, only those type variables shared by two labels coming from the same member set of W are non-generalizable. The new type of the class *c_counter* is

$$\textsf{class}\ \ c\_counter\colon \textsf{object}\ \ \ldots\ \textsf{end}\ \ W = \{\{\mathit{Value},\ \mathit{plus}\},\ \{\mathit{Value},\ \mathit{incr}\}\}$$

We see that the labels *plus* and *incr* are separated in different member sets, so that the type variable ϱ can be polymorphic. More generally, the layered structure of the new W gets rid of the transitivity side-effect, and restores proper polymorphism. It is important to notice that the new W sets still abstract on the join-patterns: zero-arity labels and non-synchronized labels do not show up.

4 Selective refinement and abstract labels

4.1 Semantics of selective refinement

Selective refinement is the only operator on classes that can modify the reaction rules. It can be understood by analogy with ML-style pattern matching. Selective refinement rewrites a class c into a new class c' by matching the reaction rules of c against a refinement sequence. A refinement sequence is an ordered list of refinement clauses, and refinement clauses are tried in that order. A refinement clause $K_1 \Rightarrow K_2 \triangleright Q$ consists of three parts: the selective pattern K_1, the refinement pattern K_2 and the refinement process Q. A refinement clause matches a reaction rule $J \triangleright P$ when K_1 is a sub-pattern of J, that is, when all the labels in K_1 also appear in J. Then, J can be written as K_1 & K and the matched reaction rule is refined into the new reaction K & $K_2 \triangleright P$ & Q. As a consequence, a reaction rule is rewritten according to the first matching refinement clause if it exists. Otherwise, the reaction rule remains unchanged.

4.2 Abstract labels after selective refinement

Now let us consider again the class *c_counter* of Sec. 3.2 and assume that, for debugging purpose, we wish to log each *plus* attempt on the terminal. This can be achieved by a two-step modification: First, the *plus* label is renamed into *unlogged_plus*, using selective refinement; then, the *plus* label is redefined:

$$\begin{array}{l}
\textsf{class}\ \ c_intermediate = \\
\quad \textsf{match}\ c_counter\ \textsf{with} \\
\qquad plus(y,r) \Rightarrow unlogged_plus(y,r) \triangleright \textsf{0} \\
\quad \textsf{end} \\
\textsf{class}\ \ c_logged_counter = \textsf{self}(z) \\
\quad c_intermediate \\
\textsf{or}\ plus(y,r) \triangleright out.print_int(y)\ \&\ z.unlogged_plus(y,r)
\end{array}$$

The type of the class *c_intermediate* is as follows:

class $c\_intermediate$: $\forall\varrho$. object
label $Value$: (int) ;
abstract label $plus$: (int , [$reply$: (); ϱ]) ;
label $incr$: ([$reply$: (); ϱ]) ;
label $unlogged\_plus$: (int, [$reply$: (); ϱ])
end $W = \{...\}$

It is important to notice that this type is inferred at compile time, considering only the type of the class *c_counter* and the refinement clauses, without actually performing the selective refinement. We observe that the label *plus* is still present in the type of class *c_intermediate*. However, this label is now tagged as abstract. In fact, the normal form of class *c_intermediate* as computed at runtime looks like:

class $c\_intermediate = \mathsf{self}(z)$
$Value(x)$ & $unlogged\_plus(y,r) \triangleright r.reply()$ & $z.Value(x+y)$
or $Value(x)$ & $incr(r) \triangleright counter.Value(x)$ & $z.plus(1,r)$
or $\{plus\}$

The label *plus* does not appear in the patterns any more, but it still appears in the guarded process of the second reaction rule. This fact is accounted for by stating that *plus* is abstract in class *c_intermediate*, using the new construct $\{plus\}$. As in the ordinary sense of object-oriented programming, classes with abstract labels cannot be instantiated. Therefore, class *c_intermediate* is prevented from instantiation until a concrete and type compatible definition for *plus* is provided, as performed by the definition of class *c_logged_counter*. This constraint is reflected by the abstract tag in the class type.

4.3 Abstract or maybe abstract

We move on to another example: a machine creator (the class) which creates vendor machines (the objects) selling both tea and coffee.

class c_vendor =
$two\_euros()$ & $tea() \triangleright out.print\_string(\text{“}tea\text{”})$
or $two\_euros()$ & $coffee() \triangleright out.print\_string(\text{“}coffee\text{”})$

Later, the price of tea decreases to one euro, so the machine creator should be modified. Such a modification is attempted by selective refinement:

class c_promo_vendor =
match c_vendor with
$two\_euros()$ & $tea() \Rightarrow one\_euro()$ & $tea() \triangleright 0$
end

The normal form of class *c_promo_vendor* at runtime is:

class c_promo_vendor =
$one\_euro()$ & $tea() \triangleright out.print\_string(\text{“}tea\text{”})$
or $two\_euros()$ & $coffee() \triangleright out.print\_string(\text{“}coffee\text{”})$

Unfortunately, the type of class *c_promo_vendor* tags *two_euros* as abstract:

class c_promo_vendor: object

abstract label two_euros: () ;
label tea: () ; label $coffee$: () ; label one_euro: ()
end $W = \{...\}$

And this will, in practice, prevent from creating any real new vendor machine objects from this class.

The discrepancy between the compile time type and the runtime normal form comes from the way selective refinement is typed. Since types do not include all synchronization information, the effect of selective refinement can only be approximated while typing. As regards labels that no more exist in the join-patterns of the resulting class, the approximation can be described as follows:

1. For each clause in the selective refinement $K_1 \Rightarrow K_2 \triangleright Q$, compute the set of labels that appear in K_1 but not in K_2.
2. Take the union of the differences calculated in 1.

As demonstrated by the refinement from c_vendor to c_promo_vendor, this approximation is not always exact. As an untimely consequence, the abstract qualifier in types significantly departs from its usual meaning: an abstract label is in fact a label that *may* lack a definition, while common sense and tradition suggest that abstract labels should be labels with no definition. Accordingly, the information conveyed by the abstract qualifier in types is significantly reduced.

This abstract as maybe abstract semantics is expressed by a "subsumption" rule Sub in the type system of [9]. Basically, this rule lifts abstract label set inclusion into subtyping. Such a subtyping is of little practical interest.

However, if selective refinement can be effectively carried out during typing, the type of compile time will be able to tell exactly which label is abstract in the runtime normal form. Then the expedient of the Sub rule can be withdrawn. It is certainly a more practical solution compared with the former type system.

5 New labels during selective refinement

In the previous section, we saw that some labels in a class can lose their definitions during selective refinement. Symmetrically, selective refinement introduces new labels. Those new labels endanger the basic type safety property of avoiding "message not understood" errors. For instance, consider the following selective refinement.

class cc = match c with $x()$ & $y() \Rightarrow z() \triangleright 0$ end

We additionally assume that class c indeed possesses the two labels x and y, as shown by its type.

class c: object label x: () ; label y: () end $W = \{...\}$

Then, label z is new to the class c, and by the typing of [9], the type of the resulting class cc will include the new label:

class cc: object
abstract label x: () ; abstract label y: () ;
label z: () ;
end $W = \{...\}$

However, class c can be either of the following two definitions, because both of them have the type of class c given above.

$\mathsf{class}\ \ c_1 = x() \ \&\ y() \triangleright P_1$

$\mathsf{class}\ \ c_2 = \ x() \triangleright P_2 \ \mathsf{or}\ y() \triangleright Q_2$

If c is c_1, the runtime normal form of class cc will effectively possess the new label z. But, if x and y are not synchronized (*i.e.* if c is c_2), then class cc does not provide a definition for label z. Obviously, if nothing is done for the latter case, basic type safety is broken. The solution adopted in [9] enforces the actual introduction of all new labels. In practice, a runtime check of the presence of some labels (in this example, label z) has to be performed after selective refinement. When some labels are missing, the running program fails with a specific *refinement error*. From a language design perspective, this solution is moderately acceptable.

1. It can be argued that programmers would design selective refinements with the intention to define some new labels.
2. Refinement error is by far more tolerable than the "message not understood" error. The former may only show up when building classes, while the latter may occur when using objects. Hence, refinement error occurs earlier, less frequently, and conveys properer information for easier correctness by programmers.

However, guaranteeing type safety at no runtime price is certainly a better solution. This can be achieved provided types are detailed enough, so that pattern matching on reaction rules can be performed while typing. Then, only effectively introduced new labels show up in the resulting type.

6 The new type system

Based on the analysis in the previous three sections, we realize that the weak points of the system of [9] originates in the information deficiency in class types. As a consequence, this system fails to embody the semantics of synchronization behaviors and their inheritance. We now present our new type system, which elaborates on the former system of [9]. The main novelty lies in equipping each class type with the structure of all its join-patterns, but there are other differences. In our description, we mostly point out changes *w.r.t.* the former system. We do not include here the syntax of objects and classes, nor their semantics, since those are unchanged except for the suppression of one runtime check (see Sec. 5).

6.1 Type algebra

The type algebra of the new type system appears in Fig. 1. The class types are written as $\zeta(\rho)B^W$. As before, B lists the types of all the *declared* labels in the class and $[\rho]$ refers to the object type of self. By contrast, our *correlated* label

$$
\begin{array}{lll}
\tau & ::= \theta \mid [\rho] & \textbf{Object type} \\
\rho & ::= \emptyset \mid \varrho \mid m : \widetilde{\tau}; \rho & \textbf{Row type} \\
\sigma & ::= \forall X.\tau & \textbf{Type scheme} \\
\\
\alpha & ::= \theta \mid \varrho & \textbf{Type Variable} \\
\widetilde{\tau} & ::= (\tau_i{}^{i \in I}) & \textbf{Tuple type} \\
B & ::= \emptyset \mid \ell : \widetilde{\tau}; B & \textbf{Internal type} \\
\\
\tau^c & ::= \zeta(\rho)B^W & \textbf{Class type} \\
\sigma^c & ::= \forall X.\tau^c & \textbf{Class type scheme} \\
\\
\tau^s & ::= B^F & \textbf{Refinement sequence type} \\
F & ::= \emptyset \mid \pi \Rightarrow W; F & \textbf{Refinement rules}
\end{array}
$$

Fig. 1. Syntax for the type algebra

set W replaces the coupled label set of [9]. Following the two-layer structure introduced in Sec. 3.2, W is a set of sets of labels, namely, $W \subseteq 2^{\mathcal{L}}$ ($\mathcal{L}$ collects all the labels in this calculus). Member sets of W are ranged over by π ($\pi \subseteq \mathcal{L}$), each π corresponding to one join-pattern, and the whole set W represents all the patterns of the class normal form. Our class types do not include a set of abstract labels, while the types of [9] did. The set of abstract labels can now be computed easily from class types as $dom(B) \setminus \overline{W}$, where $\overline{W}$ is the flattening of W (union of all member sets) and $dom(B)$ is the domain of B. Moreover, by contrast, the abstract labels computed in this manner are abstract in the ordinary sense (see Sec. 4.3)

The type B^F of a refinement sequence conveys the latter's whole structure, including refinement clause order. The system of [9] did not attach a particular type to a refinement sequence. Although such a type is not needed to type applications of selective refinement, our solution simplifies the typing rules and reflects our attempt to make selective refinement an autonomous operator.

6.2 Typing rules

Our system's typing judgments are as follows:

$A \vdash x : \tau$	object x has type τ in environment A;
$A \vdash x.\ell :: \widetilde{\tau}$	label ℓ in object x has type $\widetilde{\tau}$ in environment A;
$A \vdash P$	process P is well-typed in environment A;
$A \vdash K :: B$	join-pattern K has type B in environment A;
$A \vdash C :: \tau^c$	class C has type τ^c in environment A;
$A \vdash S :: \tau^s$	refinement sequence S has type τ^s in environment A.

Typing judgments rely on *type environments* A that bind class and object names to type schemes:

$$A ::= \emptyset \mid c : \sigma^c; A \mid x : \sigma; A \mid x : \forall X.B; A$$

Rules for object names and labels

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

Rules for processes

$$\text{Null}\quad A \vdash 0 \qquad \text{Send}\quad \frac{A \vdash x.\ell :: (\tau_i{}^{i\in I}) \qquad (A \vdash x_i : \tau_i)^{\,i\in I}}{A \vdash x.\ell(x_i{}^{i\in I})} \qquad \text{Join Parallel}\quad \frac{A \vdash x.M_1 \qquad A \vdash x.M_2}{A \vdash x.(M_1 \,\&\, M_2)}$$

$$\text{Parallel}\quad \frac{A \vdash P \qquad A \vdash Q}{A \vdash P \,\&\, Q} \qquad \text{Class}\quad \frac{\begin{array}{l} A \vdash C :: \zeta(\rho)B^W \qquad \rho = (B \upharpoonright \mathcal{M}); \varrho \\ A + c : \forall \mathsf{Gen}\ (\rho, B, A).\zeta(\rho)B^W \vdash P \end{array}}{A \vdash \mathsf{class}\ c = C\ \mathsf{in}\ P}$$

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

Fig. 2. Typing rules for names, and processes

An object x may have two complementary bindings in A, $x : \sigma$ (external scheme) for public labels, and $x : \forall X.B$ (internal scheme) for private labels.

The typing rules appear in Fig. 2 and 3. Before we have a close look at the rules, some auxiliary definitions should be mentioned.

- $\{\gamma_\alpha/\alpha^{\,\alpha\in X}\}$ expresses the substitution of type variables by types.
- $dom(A)$ is the set of identifiers bound in A. $A + A'$ equals $(A \setminus dom(A')) \cup A'$, where $A \setminus X$ removes from A all the bindings of names in X and $A + x : \forall X.[\rho], x : \forall X.B$ means $A \setminus \{x\} \cup x : \forall X.[\rho] \cup x : \forall X.B$.
- $B \upharpoonright L$ restricts B to the set of labels L. Labels $\ell \in \mathcal{L}$ are divided into *private labels* $f \in \mathcal{F}$ and *public labels* $m \in \mathcal{M}$.
- $ftv()$ is the set of free type variables occurring in various constructs.
- $\mathsf{Gen}\ (\rho, B, A)$ is the set of the type variables that occur free in row type ρ or in internal type B, but not in type environment A.
- $ctv(B^W)$ computes the subset of the free type variables in B that are shared by at least the types of two correlated labels according to W.

$$\begin{aligned} ctv(B^W) &= \textstyle\bigcup_{\pi_i \in W} ctv(B^{\{\pi_i\}}) \\ ctv(B^{\{\pi\}}) &= \textstyle\bigcup_{l\in\pi, l'\in\pi, l\neq l'} ftv(B(l)) \cap ftv(B(l')) \end{aligned}$$

- predicate $B_1 \uparrow B_2$ expresses that B_1 and B_2 coincide on their common labels.
- $B_1 \oplus B_2$ is written for the union of B_1 and B_2, provided $B_1 \uparrow B_2$ holds.

Rules for patterns

Empty-Pattern
$$A \vdash 0 :: \emptyset$$

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

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

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

Rules for classes

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

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

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

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

Disjunction
$$\frac{A \vdash C_1 :: \zeta(\rho)B_1^{W_1} \qquad A \vdash C_2 :: \zeta(\rho)B_2^{W_2}}{A \vdash C_1 \text{ or } C_2 :: \zeta(\rho)(B_1 \oplus B_2)^{W_1 \cup W_2}}$$

Refinement
$$\frac{\begin{array}{l} A \vdash C :: \zeta(\rho)B^W \\ A \vdash S :: B'^{F} \\ \vdash W \text{ with } F :: W' \qquad B \uparrow B' \end{array}}{A \vdash \text{match } C \text{ with } S \text{ end} :: \zeta(\rho)((B' \upharpoonright \overline{W'}) \oplus B)^{W'}}$$

Rules for refinement clauses

Modifier-Clause
$$\frac{\begin{array}{ll} A' \vdash K :: B_1 & A \vdash S :: B^F \\ A' \vdash K' :: B_2 & \mathit{col}(K) = \{\pi\} \\ A + A' \vdash P & \mathit{dom}(A') = \mathit{fn}(K') \end{array}}{A \vdash K \Rightarrow K' \triangleright P \,\big|\, S :: (B_1 \oplus B_2 \oplus B)^{(\pi \Rightarrow \mathit{col}(K'))|F}}$$

Modifier-Empty
$$A \vdash \emptyset :: \emptyset^{\emptyset}$$

Rules for filters

Apply
$$\vdash \pi \uplus \pi' \text{ with } \pi \Rightarrow W \,\big|\, F :: \{\pi' \uplus \pi_i \mid \pi_i \in W\}$$

End
$$\vdash \pi \text{ with } \emptyset :: \{\pi\}$$

Next
$$\frac{\pi_1 \not\subseteq \pi \qquad \vdash \pi \text{ with } F :: W}{\vdash \pi \text{ with } \pi_1 \Rightarrow W_2 \,\big|\, F :: W}$$

Or
$$\frac{(\vdash \pi_i \text{ with } F :: W_i)^{\,\pi_i \in W}}{\vdash W \text{ with } F :: \cup_{\pi_i \in W} W_i}$$

Fig. 3. Typing rules for patterns, classes, refinement clauses, and filters

– $col(J)$ computes the correlated labels set from a join-pattern J.

$$\begin{aligned} col(\emptyset) &= \emptyset \\ col(\ell(\widetilde{u})) &= \{\{\ell\}\} \\ col(J \ \& \ J') &= \{\pi_1 \uplus \pi_2 \mid \pi_1 \in col(J), \pi_2 \in col(J')\} \\ col(J \ \mathsf{or} \ J') &= col(J) \cup col(J') \end{aligned}$$

The symbol $\uplus$ means disjoint union, since patterns are required to be linear (or in patterns is a complication, which can be ignored in a first reading).

Processes. The Class rule differs from before by the additional condition $\rho = (B \upharpoonright \mathcal{M}); \varrho$ in the premise. This condition requires an early compatibility check to verify that the types (in B) of the public labels inferred from the patterns are compatible with those (in ρ) inferred from the messages sent to self. In the old system, this check was delayed until object instantiation, which in turn delayed the detection of some errors. Such a delay is particularly annoying in a separate compilation context.

For the Object rule, checking the absence of abstract labels is now performed by the equation $dom(B) = \overline{W}$. In spite of the new condition on public labels in the Class rule, we retain the condition $\rho = (B \upharpoonright \mathcal{M})$ of [9]. It is still needed for anonymous class definitions, and also has the effect of closing the object type ρ. However, the most significant change is the replacement of the imprecise coupled labels by our exact correlated labels in the computation of generalizable type variables.

Classes. All rules are simplified thanks to the disappearance of the explicit abstract labels component in class types. The bizarre Sub rule is not needed any more, as discussed in Sec. 4.3. Our precise correlated labels set is introduced in the rule Reaction, using the external definition $col(J)$,

The rules for typing selective refinement undergo significant changes. In match C with S end, the typing of class C and the typing of refinement sequence S were combined in [9] but are now independent. Compatibility checks between labels common to C and S are now performed explicitly by condition $B \uparrow B'$; furthermore, most of the construction of the resulting type is performed by the auxiliary judgment W with $F :: W'$, which computes the patterns of the resulting class. Finally, the effectively introduced new labels (compare to Sec. 5) are added by $B' \upharpoonright \overline{W'}$. There is a minor difference *w.r.t* [9]: our system allows labels in the selective patterns of S that are not necessarily in class C. Such a requirement is no longer essential to type safety and we believe that imposing it on programmers is too restrictive. Of course, some warning (analogous to the "unused match case" of ML pattern-matching) can be issued.

Refinement clauses. Rule Modifier-Clause serves both to collect label-type pairs (in B) and to compute the refinement rules F. Processes of refinement clauses P are also typed.

Filters. The rules for typing a filter mimic the semantics of selective refinement which is in analogy with ML-style pattern matching, but the irrelevant management of abstract names is omitted. Rules Apply, Next, and End handle the matching of one π set by the refinement rule F, while rule Or collects the resulting partial W_i sets into the resulting correlated label set W'. Linearity of resulting patterns is now checked simply by the disjoint union $\pi' \uplus \pi_i$ used in the rule Apply. Thereby, we drop one cryptic premise in the Refinement rule of [9].

6.3 Subject reduction and safety

The typing is finally extended to chemical solutions in order to illustrate the properties of the type system. An additional judgment: $A \vdash \mathcal{D} \Vdash \mathcal{P}$ is used and the rules are as follows:

Chemical-Solution
$$\frac{A = \cup_{\psi x \# D \,\in\, \mathcal{D}}\ A_x \qquad (A^{\psi} \setminus \{x\} \vdash D :: A_x)^{\psi x \# D \,\in\, \mathcal{D}} \qquad (A^{\psi} \vdash P)^{\,\psi \# P \,\in\, \mathcal{P}}}{\vdash \mathcal{D} \Vdash \mathcal{P}}$$

Definition
$$\frac{A \vdash \mathsf{self}(x)\, D :: \zeta(\rho) B^{W} \qquad X = \mathsf{Gen}\ (\rho, B, A) \setminus ctv(B^{W}) \qquad \rho = B \upharpoonright \mathcal{M} \qquad dom(B) = \overline{W}}{A \vdash D :: x : \forall X.[\rho], x : \forall X.(B \upharpoonright \mathcal{F})}$$

For lack of space, and because this work focuses on type system design, we will not recall the semantics of the objective-join calculus, which can be found in [9][1]. However, we formally state the correctness of our system.

Theorem 1 (Subject reduction).

1. *Chemical reduction preserves chemical typing:* *Suppose* $\vdash \mathcal{D} \Vdash \mathcal{P}$. *If* $\mathcal{D} \Vdash \mathcal{P} \equiv \mathcal{D}' \Vdash \mathcal{P}'$ *or* $\mathcal{D} \Vdash \mathcal{P} \longrightarrow \mathcal{D}' \Vdash \mathcal{P}'$, *then* $\vdash \mathcal{D}' \Vdash \mathcal{P}'$.
2. *Class rewriting preserves typing: if* $A \vdash P$ *and* $P \longmapsto P'$ *then* $A \vdash P'$.

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

Informally, the type system guarantees: no undefined class names or object names; no attempts to instantiate from abstract classes; no violations of privacy; and no "message not understood" errors. The formal definitions of failures appear in [14], so do the proofs.

This kind of safety meets the general requirements of a standard type system. Of course, richer system can be considered to provide more specific properties such as deadlock-freedom, *etc.* [12]

[1] In the class semantics, the premise $dl(S) \subseteq dl(C')$ can be erased from rule Match

7 Conclusion and future work

Class types, also known as class interfaces or class signatures, should characterize the externally visible properties of classes. We claim that when concurrent objects are concerned, the synchronization behavior is one such critical property. Based on this idea, we redesigned the static semantics of the concurrent object calculus of [9], equipped class types with their join patterns, and came up with a simpler but more expressive type system. This new type system has more polymorphism, suppresses the runtime overhead that was essential to safety in the old system, and accepts more safe programs. Furthermore, it is easier for programmers to understand and use. We also implemented type inference adapting some of the efficient techniques from [17].

Future work. As putting whole join-patterns into class types compromises abstraction, whether and how we can regain it is an interesting question. We see that the current type system handles all class operations (instantiation, disjunction and selective refinement). If we impose some restrictions on the application of certain operators to certain classes, class types can then be more abstract. For example, when some class is tagged as *final*, the only required synchronization-related information is the set of non-generalizable type variables. So instead of keeping the join-patterns, we can record just this set of variables. More generally, different restrictions will require different amount of synchronization information, and we can eventually reach a multi-leveled framework of abstracting class types, in which a class type may be assigned different degrees of abstraction in different contexts.

Another way to introduce abstraction is by *hiding* labels defined in a class. Such labels do not show up in the class type, so are untouched by later inheritance and overriding. This will in turn cause significant changes in the semantics of the class operations, probably leading us to a more powerful calculus.

In the old type system [9], selective refinement is typed per application. It would be more interesting if we could make selective refinement an autonomous class-to-class operator. Giving each selective refinement an independent type as we have done in our current type system falls short of this goal: the description of the parameters accepted is missing, and it is impossible to infer the resulting class by looking at the type of selective refinement alone. We wish to integrate our work with current research on *mixins* [3,6] and hopefully make the selective refinement a mixin-like operator in the concurrent setting.

Our ultimate goal is to implement our object-oriented extension of join calculus as a concurrent object-oriented programming language. Since the JoCaml [13] implementation already incorporates the join calculus into OCaml, we will pursue our goal by extending JoCaml.

Acknowledgements. The authors wish to thank James Leifer and Pierre-Louis Curien for their comments on drafts of this paper.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for $C^{\#}$. In *Proceedings of ECOOP 2002*, LNCS 2374, pages 415–440, 2002.
2. P. D. Blasio and K. Fisher. A calculus for concurrent objects. In *Proceedings of CONCUR '96*, LNCS 1119, pages 655–670, 1996.
3. G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of OOP-SLA.ECOOP '90*, pages 303–311. ACM press, 1990.
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. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL '98*, pages 171–183. ACM press, 1998.
7. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM press, 1996.
8. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of CONCUR '97*, LNCS 1243, pages 196–212, 1997.
9. C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*. To appear, `http://pauillac.inria.fr/~maranget/papers/ojoin.ps.gz`.
10. A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *Proceedings of HLCL '98*, ENTCS 16(3), 1998.
11. B. J. James Gosling and G. Steele. *The Java Language Specification*. Addison Wesley Longman Inc., Reading, Mass., 1996.
12. N. KoBayashi. Type systems for concurrent programs. In *Proceedings of UNU/IIST 10th Anniversary Colloquium, Lisbon, Portugal*, 2002. To appear.
13. F. Le Fessant. The JoCaml system prototype. Software and documentation available from `http://pauillac.inria.fr/jocaml`, 1998.
14. Q. Ma and L. Maranget. Expressive synchronization types for inheritance in the join calculus. Rapport de recherche 4889, INRIA-Rocquencourt, `http://pauillac.inria.fr/~ma/papers/oojoin-tr.ps`, 2003.
15. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. The MIT Press, 1993.
16. O. Nierstrasz. Towards an object calculus. In *Proceedings of ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 1–20, 1992.
17. D. Rémy. Extension of ML type system with a sorted equation theory on types. Rapport de recherche 1766, INRIA-Rocquencourt, 1992.
18. 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.
19. V. T. Vasconcelos. Typed concurrent objects. In *Proceedings of ECOOP '94 Workshop on Object-Based Concurrent Computing*, LNCS 821, pages 100–117, 1994.
20. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of OOPSLA '86*, ACM SIGPLAN Notices 21(11), pages 258–268, 1986.