INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the (un)reality of virtual types

Didier Rémy and Jérôme Vouillon

No ????

Mars, 2000, (in preparation)

THÈME 2

Rapport de recherche

ISSN 0249-6399

INRIA
ROCQUENCOURT

On the (un)reality of virtual types

Didier Rémy and Jérôme Vouillon*

Thème 2 — Génie logiciel
et calcul symbolique
Projet Cristal

Rapport de recherche n° ???? — Mars, 2000, (in preparation) — 22 pages

Abstract: We show, mostly through detailed examples, that object-oriented programming patterns known to involve the notion of virtual types can be implemented directly and concisely using parametric polymorphism. A significant improvement we make over previous approaches is to allow related classes to be defined independently. This solution is more flexible, more general, and we believe, simpler than other type-safe solutions previously proposed. This approach can be applied to several languages with parametric polymorphism that can already type binary methods and have structural object types.

Key-words: Virtual Types, Structural Types, Types, Objects, Classes, Modularity, Ocaml

(Résumé : tsvp)

---

* INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex France.

Résumé : Nous montrons, principalement sur des exemples détaillés, que des schémas de programmation avec objets connus pour nécessiter la notion de types virtuels peuvent être écrits de façon directe et concise en utilisant le polymorphisme paramétrique. Une amélioration significative que nous apportons par rapport aux approches antérieures est de permettre à des classes liées de pouvoir être définies indépendemment. Cette solution est plus flexible, plus générale, et nous pensons, plus simple que les autres solutions correctement typées qui ont été proposées. Cette approche peut être utilisée dans plusieurs langages avec polymorphisme paramétrique qui peuvent déja typer les méthodes binaires et possèsent des types d'objets structurels.

Mots-clé : Types Virtuels, Types Structurels, Types, Objets, Classes, Modularité, Ocaml

Introduction

The programming patterns that lead to the introduction of virtual types are not new in the object-oriented community. However, they recently became famous for the typechecking challenges that they posed to some recent, modern, and statically typed object-oriented languages, including several extensions to Java [OW97a, Bru97]. The notion of virtual types, first introduced in BETA [KMMPN87], was not type-safe.

It is difficult to precisely define virtual types, since they have been presented through programming patterns and (often incomplete) examples. Some of them were written in informal extensions of existing languages. Virtual types have never been given a semantic or syntactic characterization. Intuitively however, virtual types always involve the definition of several related classes that interact with one another, such that at least the type of objects of one of these classes depends on the type of objects of another one ; moreover, the types of objects of these classes are often, but not always, mutually recursive.

A few statically type-safe versions of virtual types have been proposed [BOW98, Tor98]. In [BOW98] the authors show that examples involving virtual types can be rewritten using only parametric polymorphism. The language they used for this demonstration is Pizza —an extension of Java with parametric polymorphism [OW97a]. However, as the examples of virtual types written in Pizza are sometimes long and tedious, the authors propose a new primitive construct to allow a better treatment of virtual types. This includes a mechanism for grouping together several class definitions. Classes of a group can be inherited, but only simultaneously. The other work [Tor98] also introduces a primitive but different construct.

In this paper, we argue as in [BOW98] that virtual types can be encoded with parametric polymorphism. However, we disagree with the introduction of a primitive notion of virtual types to palliate the heaviness of their encoding in Pizza. Conversely, we claim that no specific construct is necessary. We argue by giving a more direct coding of examples involving virtual types. The approach is simpler, more flexible, and more powerful than theirs.

In fact, using Ocaml [Ler96, RV98] as the implementation language, all examples that we know as involving virtual types can actually be typed in with no effort at all. Actually, the untyped code that one naturally writes needs only to be documented with class type parameters and, sometimes, with type constraints. No other change in the code is necessary. Had we not known about this challenge, we would not have noticed any particular difficulty.

Through detailed examples and comparisons with other solutions, we are still able to identify the potential difficulty of virtual types. An obvious requirement is to have self-types in the language, *i.e.* the ability to type self with an open object type (partially known object types). However, mutual recursion between open object types is also needed in many examples of virtual types. Then, the key is to be able to leave the recursion at the level of types such that classes do not have to be defined recursively. In Ocaml, this is permitted by the use of structural object types, and of row variables [Rém94, Wan87] to keep the monomorphic, recursive structure isolated from the unconstrained, polymorphic row variables.

In fact, we are also able to transpose our solution to a few other languages languages that posses structural types and that can already handle binary methods. In the language $\mathtt{F}^{\omega}_{<:}$ [Car90, SP94], we only need to assume some standard class-based object-oriented extension, but nothing else. Languages based on match-bounded quantification [BSvG95, BFP97] or F-bounded quantification [CCH^{+}89] need to be extended with recursively defined bounds. For instance, this has been recently been done for the language LOOM in [BV99]. On the other hand, languages such as Pizza or GJ that already have recursive F-bounded quantification need to be extended with structural object types, only.

The rest of the paper is organized as follows. In section 1, we describe in detail the subject/observer pattern [GHJV95] in Ocaml ; we explain the typing constraints and show the most general class types that are automatically inferred in Ocaml. In section 2, we compare our solution to other solutions in strongly-typed languages. We discuss the drawbacks of tightly grouping definitions of related classes as do virtual types ; we show that enforcing a fixed recursive schema is too rigid (it imposes some redundancies) and are too restrictive (some inheritance patterns cannot be expressed). In section 4, we consider another example of virtual types known as the "cow/animal", and discuss the issues of expressiveness and abstraction. We also show that the lack of abstraction does not clearly reduce expressiveness. In section 5, we use the example of alternating lists to argue that while parametric types are preferable to virtual types : parametric types allow a continuous refinement of a hierarchy of classes while virtual types show up abruptly at a particular point in the hierarchy.

1 The detailed example of subject/observers

Virtual types are well illustrated by the example of the subject/observer programming pattern. We simultaneously describe this example, its code, and its typing in Ocaml. Since Ocaml has type inference, the user does not need to write types except type parameters of classes and appropriate type constraints to bind them in class bodies. All the examples of this section are written in the language Ocaml [Ler96] and have been processed automatically by the toplevel interaction loop. We only show the output in italic when needed.

The subject/observer example can be used when one or several observers need to watch the state of one or several subjects or, more generally, when objects of one class need to pass themselves to objects of another class where both classes need to remain extensible. We first define the skeleton of the pattern that implements the interaction between the subject and the observer ignoring all other operations. The subject reports actions to its observers. Thus, it must remember the list of its observers in a field variable. It should also provide some methods to add or remove observers. More importantly, the subject has a distinguished method `notify` used to forward information to all of its observers (meanwhile adding itself as an extra argument so that the observers can call the subject back).

```
class ['O, 'E] subject =
  object (self)
    val mutable observers : 'O list = []
    method add_observer obs = observers ← obs :: observers
    method notify (e : 'E) =
      List.iter (fun obs → obs#at_notification self e) observers
  end;;
class ['a, 'b] subject :
  object ('c)
    constraint 'a = < at_notification : 'c → 'b → unit; .. >
    val mutable observers : 'a list
    method add_observer : 'a → unit
    method notify : 'b → unit
  end
```

The inferred type of class `subject` is parameterized over two variables `'a` and `'b` standing for the types of observers and of watched events, and self type `'c`. The type of observers `'a` is constrained to be an object type with at least a method `at_notification` of type `'c → 'b → unit`. The dots ".." in the constraint are a so-called row variable [Wan87, Rém94] that is kept anonymous in Ocaml for simplicity of presentation. Thus, `'a` may be the type of an object with more methods, for instance in a subclass. The class also has a field `observers` of type `'a list` and two methods `add_observer` and `notify` of respective types `'a → unit` and `'b → unit`. It is also implicit from the class type that the type of self bound to `'c` is the type of an object with at least two methods `add_observers` and `notify` with their respective types.

The class `observer` is simpler. Mainly, it possesses a method `at_notification` to receive information from the subjects and determine what to do next. The default behavior of the method `at_notification` is to do nothing, and the method will likely be refined in subclasses.

```
class ['S, 'E] observer = object method at_notification (s : 'S) (e : 'E) = () end;;
```

The difficulty of this pattern usually lies in preserving the ability to inherit from this general pattern to create new, real instances. This is easy in Ocaml, as we illustrate on the example of a window manager. We first define the type of events used to communicate between a window and a manager. We make the event an object with an `action` property, so that the type of events can still be refined in subclasses[1] (In Ocaml, it would be more appropriate to use open variants to represent events, which we show in appendix ; we keep the full object-oriented solution here to ease comparison with other languages.)

```
type action = Move | Lower | Raise;;
class event x = object method action : action = x end;;
```

The window executes all sorts of graphical operations, including its own display ; whenever necessary, the window will notify the manager that some particular task has been completed. To allow further refinements of events, we abstract the class window over a function that coerces actions to events.

```
class ['O, 'E] window (event : action → 'E) =
  object (self)
    inherit ['O, 'E] subject
    val mutable position = 0
    method move d = position ← position + d; self#notify (event Move)
    method draw = Printf.printf "[Position␣=␣%d]"position;
  end;;
```

The manager watches events and defines the next task to be accomplished depending on both the event and its sender. This may include replying to the sender. For instance, when the observer is informed that the window has moved, it may tell the window to redisplay itself.

```
class ['S, 'E] manager =
  object
    inherit ['S, 'E] observer
    method at_notification s e = match e#action with Move → s#draw | _ → ()
  end;;
```

Here is an example :

```
let window = new window (new event) in
window#add_observer (new manager); window#move 1;;
[Position = 1]− : unit = ()
```

---

[1] This parameterization is suggested but not implemented in [BOW98].

Classes `window` and `manager` can be further extended by inheritance. We first refine the type of events as suggested above. Then, we refine the window class with a new behavior.

```
class big_event b x = object inherit event x method resized : bool = b end;;
class [' O] big_window() =
  object (self)
    inherit ['O, big_event] window (new big_event false) as super
    val mutable size = 1
    method resize x = size ← size + x; self#notify (new big_event true Raise)
    method draw = super#draw; Printf.printf "[Size␣=␣%d]"size;
  end;;
```

Here, we have definitely fixed the type of events for simplicity ; of course, we could have kept it parametric as we did for simple windows. The behavior of the manager can also be refined, independently.

```
class [' S] big_manager =
  object
    inherit ['S, big_event] manager as super
    method at_notification s e =
      if e#resized then s#draw else super#at_notification s e
  end;;
```

Note that the big windows are not bound to be managed by big managers. In particular, one may freely choose any of the following combinations (the missing one would fail to type, since a big manager requires its window to have a method `draw`) :

```
(new window (new big_event false))#add_observer (new manager);;
(new big_window())#add_observer (new big_manager);;
(new big_window())#add_observer (new manager);;
```

Classes `manager` and `window` are defined independently. This is important, since otherwise every combination would require a new definition.

More interestingly, we can also define another, entirely unrelated, observer. This strengthens the idea that the class `window` should not be paired with the class `manager`. For instance, it is easy to implement an observer that simply traces all events.

```
class [' S] trace_observer =
  object
    inherit ['S, big_event] observer
    method at_notification s e =
      Printf.printf "<%s>"
        (match e#action with Lower → "L"| Raise → "R"| Move → "M")
  end;;
```

Here is an example combining all features :

```
let w = new big_window() in
   w#add_observer (new big_manager); w#add_observer (new trace_observer);
   w#resize 2; w#move 1;;
<R>[Position = 0][Size = 3]<M>[Position = 1][Size = 3]− : unit = ()
```

The example could be continued with many variations and refinements. We also show another, safer implementation of the subject/observer in appendix A.

2 Comparison with other proposals

The affirmed strength of virtual types is that they allow one to define families of classes (such as the subject/observer pair in the example above) whose instances are to interact with one another, in such a way that these classes can be extended by inheritance without limitation. Primitive solutions to virtual types are based on the mutually recursive definitions of classes of the same family.

2.1 The subject/observer pattern in [BOW98]

The general subject/observer pattern is implemented as follows in [BOW98] (we filled in some of the holes) :

```
public interface SubjectObserverIfc {
  public interface ObserverIfc (O) { void at_notification (S s, E e); }
  public interface SubjectIfc (S) { void add_observer (@O o); void notify (E e); }
  public use Object (E);
}
public class SubjectObserver implements SubjectObserverIfc {
  public static class Subject (S) implements @SubjectIfc {
    protected @O observers []; public void add_observer (@O obs) { ... }
    public void notify (E e) {
      for (int i = 0; i < observers.length; i++)
        observers[i].at_notification(this, e);
    }
  }
  public static class Observer (O) implements @ObserverIfc
    { public void at_notification (S s, E e) { } }
}
```

Classes `Subject` and `Observer` are defined as inner classes of the class `SubjectObserver`. The only way to inherit from the class `SubjectObserver` is to inherit and refine all elements together. Following our example from the previous section, we define a class of events to be used by `WindowManager` :

```
public class Event {
    protected int a; public Event (int x) { this.a = x; }
    public int action () { return a; }
}
```

```
public interface WindowManagerIfc extends SubjectObserverIfc {
  public interface WindowIfc (S) extends SubjectIfc {
    public void move (int d);
    public void draw;
  }
  public interface ManagerIfc (O) extends ObserverIfc { ... }
  public use Event (E);
}
public class WindowManager extends SubjectObserver implements WindowManagerIfc {
  public Window extends Subject {
    extends Subject implements SubjectIfc
    protected int position;
```

```
    public Window () { position = 0; }
    public void move (int d) { position = d; this.notify (new Event(0)); }
    public void draw () { ... }
  }
  public Manager extends Observer {
    extends Observer implements WindowIfc
    public void at_notification (S s, E e) { if (e.action == 0) s.draw(); }
  }
}
```

The construction "`use`", introduced in the appendix of [BOW98], allows one to parameterize classes of the group by another class defined outside of the group (here the class `Event`). However, this construction, which is only sketched in [BOW98], remains unclear. In particular, the code above is certainly not correct. Indeed, the expression ***new event(0)*** has type `Event` and not `E` (since `E` is any type that extends `Event`). Hence, we should either definitely set the type `E` to be `Event`, or parameterize the class `Subject` over the creation of events as we did in section 1. The former will not allow further extension of the type of events. The later would more roughly follow our solution in Ocaml. However, further complications are induced. First, one need to provide an object instead of a function to the constructor `Window` (but this is only resulting from the absence of first class functions), using an auxiliary class `MakeEvent`. Second, and more cumbersome, the type of the class `MakeEvent` need also to be introduced with a `use` clause, since its type depend on the type of events. In other words, the construction `use` permits to import class types from the outside, but does not allow a contravariant use of the corresponding class.

2.2 The subject/observer pattern in [Tor98]

The subject/observer pattern cannot apparently be implemented with virtual types as in Torgersen's proposal [Tor98] because of lack of block structure. However, one may assume an extension of [Tor98] with block structure and recursively defined classes (as in the language Beta [KMMPN87]) ; then, one should be able to solve the subject/observer pattern more or less as follows.

```
ObserverArray = Array { ELEMENT <= Observer; }
Subject = {
  A <= ObserverArray; E <= Object; O <= Observer;
  observers : A;
  add_observer (obs : O) { ... }
  notify (E e) {
    for (int i = 0; i < observers.length; i++) observers[i].at_notification(this, e);
    }
  }
Observer = {
  S <= Subject; E <= Object;
  at_notification (S s, E e) { }
}
```

These classes are all recursively defined, even though this is not syntactically apparent. However, only class types are recursive. Moreover, classes do not need to be grouped with a rigid structure. In particular, it is possible to inherit separately from any class.

```
Event = { a : Int; action () : Int { return a; } }
ManagerArray = ObserverArray { ELEMENT <= Manager; }
```

```
Window = Subject {
  A <= ManagerArray; E = Event; O <= Manager;
  position : Int;
  move (d : Int) { position = d; this.notify (new Event); }
  draw () { ... }
}
Manager = Observer {
  S <= Window; E <= Event;
  at_notification (S s, E e) { if (e.action == 0) s.draw(); }
}
```

In class `Window`, the type `E` must be fixed to be exactly the type `Event`, as the method `notify` expects an object of type `E` and receives an object of type `Event`. This will prevent possible refinement of events in a subclass. While in section 2.1, this could be fixed by type abstraction, there does not seem to be a similar solution here.

It is not possible to directly create objects from classes `Window` or `Manager`, etc. since they contain open types (types that are not finally bound to a fixed type). Therefore, one has to subclass all of them :

```
RealManagerArray = ManagerArray { ELEMENT = RealManager; }
RealWindow = Window { A = RealManagerArray; O = RealManager }
RealManager = Manager { S = RealWindow; E = Event; }
```

Classes are more loosely coupled here than in section 2.1, since the scope of the recursive definition is less rigid than in the previous solution. However, one is forced to define a hierarchy of class generators that are used only for inheritance and to take final instances of the generators that are used only to build objects. In this sense, the solution resembles the encoding of virtual types in Pizza that is discussed in the next section and presents at least similar drawbacks.

2.3 De-coupling classes of a group

As was demonstrated by our example in section 1, classes `observer` and `subject` are independent, and do not need to be recursively defined. One advantage of loosely coupled classes is to be able to inherit separately from two distinct classes or define an independent class that will interact seamlessly with a previously defined class. On the contrary, with virtual types, classes have to be defined and inherited simultaneously in a recursive pattern that has been fixed once and for all (often called a group).

In section 1, we have shown two distinct situations where such a flexibility is useful. Indeed, one may refine the observer in several interesting ways O_1, O_2, O_3 and, independently, implement several versions of the subject S_1 and S_2.

- There is no reason for O and S to have the same number of refinements; hence there is no reason for coupling the refinements S_i's with O_j's.
- Once these classes are defined, one may choose any pairing, without having to define new classes. Indeed, the type system will check that the pairing makes sense, i.e. that each one of O and S implements the components requested by the other.
- Moreover, the different classes need not be in a subclassing hierarchy. For instance, O_3 could be derived directly from the root O, and still be treated as any other observer.

In section 5, we show another argument against virtual types. Those examples demonstrate that there is a continuous path from conses to lists, via heterogenous lists. Alternating lists are then just a particular point along this path that appears simply as a type specialization of heterogeneous lists. Why should virtual types abruptly show up at this point ? Worse, following the proposal [BOW98] it would not even be possible to specialize or inherit from the previous definitions at this particular point ; instead, all classes would have to be redefined from scratch.

The more recent proposal [BV99] relaxes the coupling of classes in a group. They allow to inherit from a group and add new classes in the inherited group. However, it remains more rigid than our solution. For instance, by lack of multiple inheritance, they cannot apparently define a subject/observer group G and refine the observer class O of G in two independent observer classes O_1 and O_2 whose objects could all be connected to a same instance of the subject class S.

2.4 A pathological benefit of coupling

On the other hand, it has been argued that when the grouping is one of the goals aimed at, then a specific construct is more concise [Wad98, BV99]. Consider n mutually recursive classes, so that each class is parameterized over the types of the objects of the $n-1$ other classes. In short, the argument is that a group construct will allow the source code to be in $O(n)$ while parametric classes take at least $O(n^2)$ space of source code. However, the reality is more complex. With parametric classes, a class need to be abstracted over the $n-1$ other classes only if there are constraints in the source code that force to do so. That is, the source code of each class should at least be in $O(n)$ itself. Hence, both virtual types and parametric polymorphism would have source code in size $O(n)$. However, this is only true for the initial declaration of the group. When inheriting, the typing constraints will be duplicated with a simple inherit clause. Thus, our solution to virtual types might sometimes require larger code.

We think that such examples are pathological and that, in most cases, the number of type parameters will remain relatively small. In any case, a conclusion could only be drawn after a serious analysis of real-size examples written using both styles and not on a few sketches of programs. We should also emphasize that this is also partly a matter of syntactic sugar : recursive class definitions could be allowed to induce type abbreviations that could be used to shorten type expressions. The inconvenience of an occasional increase in the size of type annotations is also of lower importance than the serious limitation of expressiveness implied by the use of rigid grouping.

3 The necessary ingredients for decoupling classes

As shown in section 1, classes do not need to be recursive. Object types, however, often need to be mutually recursive. For instance, the type of class `subject` in section 1 has the following polymorphic class type, which uses recursive object types :

$$\begin{array}{l} \forall\, \beta, \varrho_\alpha, \varrho_\chi. \\ \quad \texttt{letrec type}\ \alpha = \langle m : \chi \to \beta \to \texttt{unit}; \varrho_\alpha \rangle \\ \quad \texttt{and}\ \chi = \langle m_1 : \alpha \to \texttt{unit}; m_2 : \beta \to \texttt{unit}; \varrho_\chi \rangle\ \texttt{in} \\ \quad \texttt{class}\ (\chi)[u : \alpha; m_1 : \alpha \to \texttt{unit}; m_2 : \beta \to \texttt{unit}] \end{array}$$

We used `letrec` **type** .. **and** .. **in** .. rather than a standard recursive μ-binder both to emphasize the mutual dependence and to reduce the verbosity (expanding the former into the later would force some duplication).

Using row-variables The above type class can be written as such in Ocaml. This is made possible by the use of row variables, such as ϱ_χ and ϱ_α (in fact, in Ocaml, ϱ_α would be left anonymous, and ϱ_χ would be left implicit.) The key point here is that the quantified row variables are independent and unconstrained. Row variables (and the corresponding rich structure of record-types) [Rém94, Wan87] allow one to grab exactly the part of the object that is polymorphic : the row variables ϱ_α and ϱ_χ occur at the leaves of the types that are bound to α and χ; therefore they are external to the recursion.

Indeed, the language Ocaml is not the only one to allow to write such a type. However, expressing this type in $F_{<:}$ fails. This is not so surprising since virtual types include binary methods and system $F_{<:}$ is not powerful enough to type binary methods. The problem is that standard record or object types used in $F_{<:}$ are less expressive than extensible-record types used for Ocaml object types : there is no way to grab record polymorphism directly as an unconstrained polymorphic type variable. Instead, one is forced to quantify α with a bound of the form $\langle m : \chi \to \beta \to \mathtt{unit}; \varrho_\alpha \rangle$ and similarly for χ; that is, one is lead to write :

$$
\begin{array}{l}
\forall\, \beta.\, \forall\, \alpha <: \langle m : \chi \to \beta \to \mathtt{unit} \rangle.\, \forall\, \chi <: \langle m_1 : \alpha \to \mathtt{unit}; m_2 : \beta \to \mathtt{unit} \rangle. \\
\quad\quad \mathtt{class}\ (\chi)[u : \alpha; m_1 : \alpha \to \mathtt{unit}; m_2 : \beta \to \mathtt{unit}]
\end{array}
$$

The bound of α depends on χ and conversely. Thus, the above type is ill-formed. There is no way to order type variables so as to obtain a well-formed type expression.

Programming in $\mathtt{F}^{\omega}_{<:}$ The system $\mathtt{F}^{\omega}_{<:}$ [SP94] is sufficiently powerful to abstract over, and thus isolate, the interior of a record type. The recursion can then be broken by parameterizing the bound of one of these types over the other one. Assuming an Ocaml-like class-based object-oriented extension of $\mathtt{F}^{\omega}_{<:}$:

$$
\begin{array}{l}
\mathtt{All}\ (\beta <: \top) \\
\mathtt{All}\ (R^\alpha <: \mathtt{Fun}\ (\chi : \star)\ \langle m : \chi \to \beta \to \mathit{unit} \rangle) \\
\mathtt{All}\ (R^\chi <: \mathtt{Fun}\ (\alpha : \star)\ \langle m_1 : \alpha \to \mathtt{unit}; m_2 : \beta \to \mathtt{unit} \rangle) \\
\quad\quad \mathtt{letrec\ type}\ \alpha = R^\alpha \chi\ \mathtt{and}\ \chi = R^\chi \alpha\ \mathtt{in} \\
\quad\quad \mathtt{class}\ (\chi)[u : \alpha; m_1 : \alpha \to \mathtt{unit}; m_2 : \beta \to \mathtt{unit}]
\end{array}
$$

The code of class subject would be :

```
class subject =
  fun (E <: ⊤)
  fun (RO <: Fun (S <: ⊤) <at_notification : S → E → unit>)
  fun (RS <: Fun (O <: ⊤) <add_observer : O → unit; notify : E → unit>)
  letrec type O = RO S and S = RS O in
    object (self : S)
      val observers : O = []
      method add_observers (x : O) = observers ← x :: observers;
      method notify (e : E) =
        List.iter (fun (obs: O) obs.at_notification self e) observers
    end
```

This solution is unsurprising, since a similar pattern has been used to type binary methods in $\mathtt{F}^{\omega}_{<:}$ [BCC$^+$96][2]. We assumed an extension of $\mathtt{F}^{\omega}_{<:}$ with a class-based language, which includes recursive types. This is to keep closer to the Ocaml object-oriented layer. We should be able to

---

[2] In [BCC$^+$96], object are directly represented as records of functions.

write the subject/observer example in plain $\mathtt{F}^{\omega}_{<:}$ encoding objects into records and existential types, even though the lack of primitive classes will likely make the example more verbose.

F-bounded quantification with recursively defined bounds

F-bounded quantification has been used to type binary methods involving recursion between a quantified type variable and its bound [CCH+89]. However, F-bounded quantification as presented in [CCH+89] only allows this particular case and does not help when the recursion extends to several bounds. Assuming a simple extension of F-bounded quantification with recursively defined bounds would allow one to replace $\forall\,(\alpha <: \tau_\alpha).\,\forall\,(\chi <: \tau_\chi).\,T$ by $\forall\,(\alpha <: \tau_\alpha \wedge \chi <: \tau_\chi).\,T$. Variables α and χ can now appear in both of their bounds.

Note that, as opposed to system $F_{<:}$ or even F-bounded quantification, ML with subtyping constraints inherently comes with recursive F-bounded quantification (with multiple bi-directional bounds) [EST95a]. Thus, an object-oriented language based on type constraints such as [EST95b] should also easily handle virtual types.

The extension of F-bounded quantification to recursively defined bounds has actually been used in the language Pizza [OW97b]. Thus, in principle, Pizza could allow the same style of programming as ours. Indeed, it was already shown in [BOW98] that examples involving virtual types can be implemented with and only with parametric polymorphism. However, the solution proposed in [BOW98] still enforces grouping : Two related classes A and B are replaced by two groups of recursive definitions. The first group defines two generators A' and B' abstracted over classes X and Y that should be instances of the generators, respectively :

public class `A'`<`X` **extends** `A'`<`X,Y`>, `Y` **extends** `B'`<`X,Y`>> { ... }
public class `B'`<`X` **extends** `A'`<`X,Y`>, `Y` **extends** `B'`<`X,Y`>> { ... }

Classes A and B are then defined as fix points of the generators :

public class `A` **extends** `A'`<`A,B`> { **public** `A` (`args`) { **super** (`args`); } }
public class `B` **extends** `B'`<`A,B`> { **public** `B` (`args`) { **super** (`args`); } }

Inheritance must be applied to generators and must preserve the group structure. The authors find this solution unsatisfactory. Consequently, they present an extension to Java/Pizza to cope with virtual types more directly. Hence, the authors chose to augment the coupling of related classes, while we prefer to loose such coupling as much as possible.

One may still wonder whether decoupling related classes is possible in Pizza, since its type system seems to have sufficient expressiveness. Below is an (unsuccessful) attempt to write the above class type in Pizza :

$$\begin{array}{l}\mathtt{class}\ (\beta \leq \langle\rangle, \alpha \leq \langle m : \chi \rightarrow \beta \rightarrow \mathtt{unit}\rangle, \chi \leq \langle m_1 : \alpha \rightarrow \mathtt{unit}; m_2 : \beta \rightarrow \mathtt{unit}\rangle)\\ \qquad [u : \alpha; m_1 : \alpha \rightarrow \mathtt{unit}; m_2 : \beta \rightarrow \mathtt{unit}]\end{array}$$

Alas, Pizza object types cannot be built from scratch, but only as the types of elements of classes. Hence, the above expression is ill-formed. An expression with equivalent meaning can only be built using recursively defined classes.

Adding structural types to Pizza would allow one to define a class with the above type. For convenience, one could also add an anonymous fix point construct so as to avoid the systematic need for the second group definition. (Once grouping has been eliminated, this second step becomes easier and should usually be a simple type application.)

Match-bounded quantification with recursively defined bounds

Match-bounded quantification [Bru95] can solve binary methods as easily as row variables do. However, it suffers from the same problem as bounded-quantification in its inability to grab the interior of a record type. In the subject/observer example, bounds would also need to be recursively defined. This difficulty has motivated an extension of LOOM [BFP97, Bru97] with recursively defined bounds, which actually lead to the proposal presented in [BOW98]. Again, we claim that match-bounded quantification with multiple recursive bounds does not have to enforce grouping. More recently, Bruce and Vanderwaart also proposed a notion of *type-group* and *class-group* that apparently releases some of the coupling of classes involving virtual types [Bru98, BV99]. In particular, an inherited group may have more components than the parent group.

In fact, since LOOM already has structural types, it would suffice to allow the definition of multiple match bounds, where all bounded type variables, as well as `MyType`, may occur in the bounds. For instance, one should be able to write the following class type in LOOM :

$$\begin{array}{l}\texttt{class}\ (\beta <\# \langle\rangle, \alpha <\# \langle m : \texttt{MyType} \to \beta \to \texttt{unit}\rangle) \\ \qquad [u : \alpha ; m_1 : \alpha \to \texttt{unit} ; m_2 : \beta \to \texttt{unit}]\end{array}$$

Moreover, using recursive classes would allow one to use the type name of one class to bound the type parameter of the other. Here is a possible implementation of the classes `subject` and `observer` in LOOM extended with recursively defined bounds and recursive classes :

```
class subject (E, O <# observer (E, MyType))
  var observers : list(O) = []
  methods
    procedure add_observer (obs : O) { observers := obs :: observers }
    procedure notify (e : E)
       { List.iter (function (obs:O) { obs.at_notification (self, e) }, observers) }
and observer (E, S <# subject (E, MyType))
  methods procedure at_notification (s : S, e : E) { }
end class
```

A significant difference with [BOW98] is that grouping is not enforced here ; on the opposite, it remains optional, and may be used only when convenient. Moreover, grouping here is just a standard recursive definition and each definition can still be understood (and used) independently from the other.

In fact, the proposal for virtual types that is presented in [BOW98] is based on matching, as done in LOOM, and is thus very close to the above schema, with the exception that it lacks structural types. This exemplifies that the choice between structural and declared object types is a rather significant design choice.

Extending type destructors to object types

Yet another possibility is to use type destructors, which have recently been proposed by Hofmann and Pierce [HP98]. Type destructors retain some of the expressiveness of type operators provided by $\texttt{F}^{\omega}_{<:}$ while keeping within second order types. In fact, row variables can be seen as a special form of type destructors for records where only the expansion is retained (the contraction for record types has been formally studied in [Rém90, Sag95], but it revealed to be useless in practice, and therefore it has not been included in the language Ocaml). Hence, type destructors might solve virtual types as well as row variables do in Ocaml. However, the meta-theory of type destructors remains quite complicated as a result of their generality, and record-type destructors are not considered in [HP98].

4 Expressiveness and abstraction

Examples of virtual types were first motivated by the need for expressiveness so that safe practical examples would not be rejected. The first solutions to virtual types were unsafe. Safe solutions have been proposed in [BOW98, Tor98]. In both works, the priority is given to expressiveness, *i.e.* to the ability to write known useful examples in a sound typed language. Recently, virtual types have also been studied for their abstraction capabilities [IP99].

The cow/animal example, introduced by David Shang [Sha96], actually mixes both issues; it also demonstrates the use of virtual types for specializing the type of a class by inheritance. First, we present our solution in Ocaml; then, we discuss the issue of type abstraction.

In its simplest and naive form, this example does not require mutual recursion between open object types; it could *a priori* be typed in languages based on $F_{<:}$ with parametric classes. The example straightforwardly translates into Ocaml, as virtual types are here only used to provide bounded type parameterization of classes.

We first define a hierarchy of food classes.

```
class food = object method food = () end ;;
class grass = object inherit food method grass = () end;;
class meat = object inherit food method meat = () end;;
```

We then introduce a generic class for starving animals.

```
class virtual ['a] starving_animal f =
  object
    constraint 'a = #food
    method diet : 'a = f
    method virtual private eat : 'a → unit
  end ;;
```

The method `eat` is flagged **virtual** because it is not yet implemented. The class must also be flagged **virtual**, so that no instance of the class can be taken. The method `eat` is flagged `private` so that it does not appear in the type of starving animals, which is `<diet : 'a>` where `'a` is the type of the food it eats. One can also define the class type of real animals (with a method eat) :

```
class type ['a] animal =
  object constraint 'a = #food method diet : 'a method eat : 'a → unit end;;
```

Of course, an animal is not an omnivore. The former eats *some* not yet specified food, while the latter eats *any* kind of food.

```
class omnivore =
  object inherit [food] starving_animal (new food) method eat f = f#food end;;
```

Note that the type `omnivore` is equal to `food animal`. We can define a particular race of animals that eat a specific type of food.

```
class camel = object
  inherit [grass] starving_animal (new grass) method eat (f:grass) = f#grass
end;;
```

The important property is to statically ensure that a camel can eat grass, but not meat (the phrase in italic is ill-typed).

```
new camel#eat (new grass);;
new camel#eat (new meat);;
```

Using an explicit coercion would not help, since `meat` is not a subtype of `grass`. A camel cannot either eat some unspecified kind of food, that is *new camel#eat (new food)* is also ill-typed.

It is important to be able to forget the exact diet of camels. Of course, a camel is not an omnivore since it cannot eat any type of food. To hide information about the type of food by subtyping, one must also hide method `eat`, which is contravariant in the type food, so as to be sound. Indeed, an animal cannot be fed if we do not know its diet. A camel can still be seen as a `grass starving_animal` since the food a camel eats is `grass`. A camel is also a `food starving_animal` : the food a camel eats is `food`, since `grass` is (a subtype of) `food`. Conversely, a camel is not a `meat animal`, since `grass` is not (a subtype of) `meat`.

In [IP99], an animal is given the following type :

$$\texttt{animal} \overset{\triangle}{=} \langle t = \exists\, \alpha <: \texttt{food}.\, \alpha; \texttt{diet} : t; \texttt{eat} : t \to \texttt{unit} \rangle$$

This says that an animal is an object with two methods `diet` and `eat` of respective types t and $t \to \texttt{unit}$ for some type t that is a subtype of `food`. A camel and an omnivore have similar types, but with t known exactly :

$$\texttt{camel} : \langle t = \texttt{grass}; \texttt{diet} : t; \texttt{eat} : t \to \texttt{unit} \rangle$$
$$\texttt{omnivore} : \langle t = \texttt{food}; \texttt{diet} : t; \texttt{eat} : t \to \texttt{unit} \rangle$$

By subtyping, a camel and an omnivore can both be given the type `animal`, forgetting their exact type, but retaining the fact that they eat some kind of food. Both of them can also be given the type of starving animals $\langle t = \exists\, \alpha <: \texttt{food}.\, \alpha; \texttt{diet} : t \rangle$, now ignoring that they can eat.

Partially abstract object types are obviously more precise than our object types. In particular, we are not able to express the type `animal` and we can only give camels the weaker type of starving animals. Practically, however, the difference is not very significant : an object of type `animal` cannot eat : since its diet is unknown, whatever food it is offered will be rejected. Thus, we have not really lost expressiveness here, but solely some form of abstraction. Conversely, common examples of the form of subject/observer pattern that involve mutual recursion of object types cannot be typed in [IP99].

In [BV99] Bruce and Vanderwaart add hash-types to provide a form of partially abstract type that is quite similar to [IP99] in terms of expressiveness.

Partially abstract types are an orthogonal feature that can also be added to mostly any language (even if this may not always be immediate and may sometimes raise other design issues). This confirms our early claim that virtual types are not an atomic construction, since it simultaneously contains a form of universal and existential (bounded) quantification.

5 At the limit of virtual types

Alternating lists have been used in [BOW98] as the basic example to illustrate the strength of virtual types. In this section, we reuse alternating lists, but as a counter-example to virtual types.

In fact, alternating lists have been chosen as a simple example of recursively define classes, and in particular, as a simplification of recursively defined parse trees [PS94][3]. We argue that alternating lists are much better implemented by parametric types than by virtual types. Using parametric polymorphism, we show that there is a continuous sequence of refinements starting with very simple classes and leading to alternating lists. Conversely, virtual types can only address a particular point in this sequence.

In Lisp the most important data structure is the memory cell called a `cons`. In an object oriented setting, it can be defined as follows :

```
class [' a, ' b] cons h t = object
  val head : 'a = h val tail : 'b = t
  method null = false method car = head method cdr = tail
end;;
```

A cons can be used as a pair, that is, a cell filled with two values of different types. In Lisp, one uses a special pointer, usually called `nil`, to distinguish from conses. Actually, it is convenient to give the class `nil` the same interface as the class `cons` since `nil` is typically used to end a chain of conses in a binary tree or a list.

```
exception Null;;
class [' a, ' b] nil = object
  method null = true method car : 'a = raise Null method cdr : 'b = raise Null
end;;
```

With parametric classes, one can define heterogeneous lists, whose elements can be of different types. In many situations one need only consider homogeneous lists ; hence we can define the more precise type :

```
type 'a homo_list = ('a, 'a home_list) cons;;
```

Alternating lists are just another particular instance of conses that could be defined as follows :

```
type ('a, 'b) alt_list = ('a, ('b, ('a, ' b) alt_list) cons) cons;;
```

Indeed, alternating lists have nothing to do with virtual types. Here, we have only specialized the types of classes `cons` and `nil` leaving the code unchanged.

The type constraint can be enforced in subclasses of `cons` and `nil` rather than in objects of those classes by restricting the types of the class parameters (we only show the subclass for cons) :

```
class [' a, ' alt_self] alt_cons h t =
  object (_ : 'self)
    constraint 'alt_self = ('b, 'self) #cons
    inherit ['a, ' alt_self] cons h t
  end;;
```

Alternating lists are not final classes, and can be specialized using inheritance. For instance, it is straightforward to add a method returning the length of the list. Of course, many more variations —including side effects and binary methods— can be devised, as for any class, just playing with parameterization and inheritance.

In summary, alternating lists appear as one particular point in a sequence of successive refinements of parametric classes. All examples can be programmed naturally and uniformly with parametric classes. On the opposite, virtual types would show up abruptly in the middle of the sequence. We do not see any problem in extending our treatment of alternating lists

---

[3] Recursively defined parse trees were chosen as a concise illustration rather than as a practical example in [PS94].

to other more complicated but similar patterns, such as parse trees, even though we are not convinced that parse trees are a good use of objects.

Conclusion

We have shown that several practical examples presented as a justification for the special treatment of virtual types can in fact be directly programmed in Ocaml. We achieved this result by decoupling related classes that had to be recursively defined, previously. Decoupling is made possible in Ocaml by the use of row variables. A few other languages that have both binary methods and structural types can also solve virtual types with parametric polymorphism, as easily.

Grouping can still be allowed, but as a notational convenience rather than as a fundamental and new construction. Given the simplicity and the additional flexibility that we have obtained by decoupling classes, we claim that parametric polymorphism and structural types are preferable to virtual types. Indeed, parametric polymorphism is well-established, easy-to-manipulate and expressive. Its strength is hereby reaffirmed. Structural types are also commonly used in type theory and almost exclusively used in higher-order type systems. However, declared types are still common among programming languages with weaker, usually non parametric, type systems. Our experience with virtual types certainly reinforces the importance of structural types. This may be an issue to consider seriously in the design of type systems for new programming languages.

Références

[BCC$^+$96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3) :221–242, 1996.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good “match” for object-oriented languages. In *ECOOP*, number 1241 in LNCS, pages 104–127. Springer-Verlag, 1997.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Informal proceedings of the FOOL'5 workshop*, 1998. Available electronically at `http://pauillac.inria.fr/~remy/fool/`.

[Bru95] Kim B. Bruce. Typing in object-oriented languages : Achieving expressibility and safety. Revised version to appear in Computing Surveys, November 1995.

[Bru97] Kim B. Bruce. Increasing java's expressiveness with ThisType and match bounded polymorphism. Technical report, Williams College, 1997.

[Bru98] Kim B. Bruce. Do parametric types beat virtual types ? Private Email discussion, October 1998.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil : A type-safe polymorphic object-oriented language. In *ECOOP*, number 952 in LNCS, pages 27–51. Springer Verlag, 1995.

[BV99] Kim B. Bruce and Joseph C. Vanderwaart. Sematics-driven language design : Type-safe virtual types in object-oriented languages. Available electronically at `http://www.cs.williams.edu/~kim/`, February 1999.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished notes, October 1990.

[CCH$^+$89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[EST95a] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA*, 1995.

[EST95b] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[HP98] Martin Hofmann and Benjamin C. Pierce. Type destructors. In *Informal proceedings of the FOOL'5 workshop*, 1998. Available electronically at `http://pauillac.inria.fr/~remy/fool/`.

[IP99] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *Informal proceedings of the FOOL'6 workshop*, 1999. Available electronically at `http://www.cs.williams.edu/~kim/FOOL/FOOL6.html`.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. The MIT Press, Cambridge, Mass, 1987.

[Ler96] Xavier Leroy. The Objective Caml system (version 2.00). Software and documentation available on the Web, `http://caml.inria.fr/`, 1996.

[OW97a] Martin Odersky and Philip Wadler. Pizza into java : Translating theory into practice. In *Proc. 24th symposium Principles of Programming Languages*, pages 146–159. ACM Press, January 1997.

[OW97b] Martin Odersky and Philip Wadler. Pizza into java : Translating theory into practice. In *24'th ACM Symposium on Principles of Programming Languages*, Paris, January 1997.

[PS94] Jens Palsberg and Michael I. Schwartzback. *Object Oriented Type Systems.* Wiley, 1994.

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat, Université de Paris 7, 1990.

[Rém94] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1994.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML : An effective object-oriented extension to ML. *Theoretical And Practice of Objects Systems*, To appear, 1998.

A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[Sag95] Camille le Moniès de Sagazan. Un système de types étiquetés polymorphes pour typer les calculs de processus à liaisons noms-canaux dynamiques. Thèse de doctorat, Université Paul Sabatier, Toulouse, November 1995. (Also CNRS/LAAS report 95077).

[Sha96] David Shang. Is a cow an animal ? Unpublished note, January 1996.

[SP94] Martin Steffen and Benjamin C. Pierce. Higher-order subtyping. Technical report ECS-LFCS-94-280, University of Edinburgh, February 1994.

[Tor98] Mads Torgersen. Virtual types are statically safe. In *Informal proceedings of the FOOL'5 workshop*, 1998. Available electronically at `http://pauillac.inria.fr/~remy/fool/`.

[Wad98] Philip Wadler. Do parametric types beat virtual types ? Private Email discussion, October 1998.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In D. Gries, editor, *Second Symposium on Logic In Computer Science*, pages 207–276, Ithaca, New York, June 1987. IEEE Computer Society Press.

A A more natural and flexible solution to the subject/observer pattern

The general idea behind the subject/observer pattern is to realize a complex operation by the tight collaboration of several objects of different classes. Here, the operations are implemented in the subject while all the control remains in the observer. In this regards, the use of type `event` to communicate between the subject and the observer is surprising and not so modular : typically, any refinement of the communication pattern will imply a simultaneous refinement of the `event` class used to communicate between the subject and the observer. Actually, the `event` class is used to pack in an extensible sum type messages that are then pattern-matched and treated by the observer class. This weakens the security, since the system does not check for exhaustive pattern-matching. A first solution in Ocaml is to represent events as a variant instead of as an object. This is described in appendix 1. Yet, a new, direct, and safer implementation of the subject/observer pattern would let the subject notify the observer by calling the appropriate message of the observer ; this is described in section A.2.

A.1 Representing events in a variant type

In is part we show a slight improvement of the example given in section 1 that represent actions as variants. The root classes `subject` and `observer` are left unchanged. The class `window` is then using a variant `` `Move `` to signal a move.

```
class ['O, 'E] window =
  object (self)
    inherit ['O, 'E] subject
    val mutable position = 0
    method move d = position ← position + d; self#notify `Move
    method draw = Printf.printf "{Position␣=␣%d}"position;
```

```
  end;;
class [' a, ' b] window :
  object ('c)
    constraint 'a = < at_notification : 'c → 'b → unit; .. >
    constraint 'b = [> ‘Move]
    val mutable observers : 'a list
    val mutable position : int
    method add_observer : 'a → unit
    method draw : unit
    method move : int → unit
    method notify : 'b → unit
  end
```

Note that, since variants are extendible, the role of the function `event` coercing actions to the type of events is played by the primitive operations on variants.

Correspondingly, the class `manager` is changed to :

```
class [' S, ' E] manager =
  object
    inherit ['S, ' E] observer
    method at_notification s e = match e with ‘Move → s#draw | _ → ()
  end;;
class [' a, ' b] manager :
  object
    constraint 'a = < draw : unit; .. >
    constraint 'b = [> ‘Move]
    method at_notification : 'a → 'b → unit
  end
```

Note the `_` that allows `at_notification` to be called with different tags, in which case the default behavior is to ignore the message. This is reminded in the type constraint `'b = [> ‘Move]`, which says that the variant type `'b` can have tag `‘Move` with a value of type unit, or any other tag with a value of any type.

A refined version of `windows` simply use other tags as needed :

```
class [' O, ' E] big_window =
  object (self)
    inherit ['O, ' E] window
    val mutable size = 1
    method resize x = size ← size + x; self#notify ‘Move
    val mutable top = false
    method raise = top ← true; self#notify (‘Resize false)
    method draw = Printf.printf "{Position␣=␣%d;␣Size␣=␣%d}"position size;
  end;;
```

Similarly, the refined version of the `manager` may respond to the new tags :

```
class [' S, ' E] big_manager =
  object
    inherit ['S, ' E] manager as super
    method at_notification s e =
      match e with ‘Resize b → s#raise | _ → super#at_notification s e
  end;;
```

Here a test showing that classes can correctly be combined : (We assume an obvious redefinition of class `trace_observer`).

```
let w = new big_window in w#add_observer (new big_manager); w#resize 2; w#move 1;;
{Position = 0; Size = 3}{Position = 1; Size = 3}− : unit = ()
```

This new version of the subject/observer pattern is simpler that the previous one, by avoiding the useless encoding of variants into objects. However, it does not provide much more security. In particular, the `at_notification` method will accept all tags in prevision of further extension. That is, a forgotten specialization of the method `at_notification` will not produce an typechecking error, but will simply ignore the notification. Below is a solution that fixes this problem and that is thus more secure. It is also simpler.

A.2 Using different methods for each kind of event

.

The method `notify` now takes a message to be send to all registered observers.

```
class [' O] subject =
  object (self : 'mytype)
    val mutable observers : 'O list = []
    method add_observer obs = observers ← obs :: observers
    method notify (message : 'O → 'mytype → unit) =
      List.iter (fun obs → message obs self) observers
  end;;
class [' a] subject :
  object ('b)
    val mutable observers : 'a list
    method add_observer : 'a → unit
    method notify : ('a → 'b → unit) → unit
  end
```

The class `observer` is initially empty.

```
class [' S] observer = object end;;
```

A window/manager is obtained by inheriting form the above pattern. For instance, the class window may have a method `move` whose code will in turn notify all observers with a messaged `moved`. Correspondingly, the window-manager is extended to accept messages `moved` : it then simply calls back the method `draw` of the window that signaled the move (called the `moved` message of the manager) :

```
class [' O] window =
  object (self : 'mytype)
    inherit ['O] subject
    val mutable position = 0
    method move d = position ← position + d; self#notify (fun x → x#moved)
    method draw = Printf.printf "[Position␣=␣%d]"position;
  end;;

class [' S] manager =
  object
    inherit ['S] observer
    method moved (s : 'S) : unit = s#draw
  end;;
```

An instance of this pattern is well-typed because the manager correctly treats all messages sent by the window.

```
let w = new window in w#add_observer (new manager); w#move 1;;
[Position = 1]− : unit = ()
```

This would not be the case if, for instance, we forgot to implement the `moved` message in class `manager`.

The pattern can further be extended. Instead of extending the `event` class, as in section 1 one can more appropriately notify the observers on another method ("resized" in the example below) :

```
class [' O] big_window =
  object (self)
    inherit ['O] window as super
    val mutable size = 1
    method resize x = size ← size + x; self#notify (fun x → x#resized true)
    method draw = super#draw; Printf.printf "[Size␣=␣%d]"size;
  end;;

class [' S] big_manager =
  object
    inherit ['S] manager as super
    method resized b (s:'S) = if b then s#draw
  end;;
```

To check the flexibility, we also add a `trace_observer` that is a refinement of the class `observer` :

```
class [' S] trace_observer =
  object
    inherit ['S] observer
    method resized (b:bool) (s:'S) = print_string "<R>"
    method moved (s:'S) = print_string "<M>"
  end;;

let w = new big_window in
   w#add_observer (new big_manager); w#add_observer (new trace_observer);
   w#resize 2; w#move 1;;
<R>[Position = 0][Size = 3]<M>[Position = 1][Size = 3]− : unit = ()
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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