Private Row Types: Abstracting the Unnamed

Jacques Garrigue

Graduate School of Mathematical Sciences,
Nagoya University, Chikusa-ku, Nagoya 464-8602
garrigue@math.nagoya-u.ac.jp

Abstract. In addition to traditional record and variant types, Objective Caml has structurally polymorphic types, for objects and polymorphic variants. These types allow new forms of polymorphic programming, but they have a limitation when used in combination with modules: there is no way to abstract their polymorphism in a signature. Private row types remedy this situation: they are manifest types whose "row-variable" is left abstract, so that an implementation may instantiate it freely. They have useful applications even in the absence of functors. Combined with recursive modules, they provide an original solution to the expression problem.

1 Introduction

Polymorphic objects and variants, as offered by Objective Caml, allow new forms of polymorphic programming. For instance, a function may take an object as parameter, and call some of its methods, without knowing its exact type, or even the list of its methods [1]. Similarly, a list of polymorphic variant values can be used in different contexts expecting different sets of constructors, as long as the types of constructor arguments agree, and all constructors present in the list are allowed [2].

These new types are particularly interesting in programming situations where one gradually extends a type with new methods or constructors. This is typically supported by classes for objects, but this is also possible with polymorphic variants, thanks to the dispatch mechanism which was added to pattern matching. This is even possible for recursive types, but then one has to be careful about making fix-points explicit, so as to allow extension. A typical example of this style is the expression problem, where one progressively and simultaneously enriches a small expression language with new constructs and new operations [3]. This problem is notoriously difficult to solve, and Objective Caml was, to the best of our knowledge, the first language to do it in a type safe way, using either polymorphic variants [4] or classes [5].

If we think of these situations as examples of incremental modular programming, we realize that an essential ML feature does not appear in this picture: functors. This is surprising, as they are supposed to be the main mechanism providing high-level modularity in ML. There is a simple reason for this situation: it is currently[1] impossible to express structural polymorphism in functors. One may of course specify polymorphic values in interfaces, but this does not provide for the main feature of functors, namely the ability to have types in the result of a functor depend on its parameters. To understand this, let's see how functor abstraction works.

[1] As of Objective Caml 3.08.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 44–60, 2006.

```
let add (p1 : float array) (p2 : float array) =
  let l1 = Array.length p1 and l2 = Array.length p2 in
  Array.init (max l1 l2)
    (fun i -> if i < l1 then if i < l2 then p1.(i) +. p2.(i)
              else p1.(i) else p2.(i))
```

This program computes the sum of two polynomials. We might want to abstract the representation of arrays, to emphasize that this program uses them functionally (arrays in OCaml are mutable.)

```
module type Vect = sig
  type t
  val init : int -> (int -> float) -> t
  val length : t -> int
  val get : t -> int -> float
end
module Poly (V : Vect) = struct
  let add p1 p2 =
    let l1 = V.length p1 and l2 = V.length p2 in
    V.init (max l1 l2)
      (fun i -> if i < l1 then if i < l2 then V.get p1 i +. V.get p2 i
                else V.get p1 i else V.get p2 i)
end
```

We have given the name `t` to `float array`, and made it abstract as a parameter. The type inferred for `add` is `V.t -> V.t -> V.t`, which depends on what implementation of `Vect` we will pass as parameter to `Poly`.

What happens now if we want to make explicit that vectors are to be represented as objects, calling methods inside the functor? Here is a first attempt.

```
module type OVect = sig
  type t = <length: int; get: int -> float>
  val init : int -> (int -> float) -> t
end
module OPoly (V : OVect) = struct
  let add (p1 : V.t) (p2 : V.t) : V.t =
    let l1 = p1#length and l2 = p2#length in
    V.init (max l1 l2)
      (fun i -> if i < l1 then if i < l2 then p1#get i +. p2#get i
                else p1#get i else p2#get i)
end
```

Type `t` is an *object type*. It gives the list of methods in the object, and their types. Methods are called with the *obj#method* notation. Objects and their types in OCaml are fully structural, and they can be seen as polymorphic records[6], extended with explicit structural subtyping. The code above typechecks correctly, but it doesn't give us enough polymorphism. Since `t` has a concrete definition in `OVect`, any module implementing `OVect` will have to include exactly the same definition. Structural subtyping allows coercing an object with more methods to type `t`, returning it in `init` or passing it to `add`, but other methods become inaccessible. That is, the result of `add` would still have only methods `length` and `get`. What we would like is to be able to define implementations where `t`