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`

has more methods than in `OVect`, so that we could still access them in the result of `add`. Intuitively, this amounts to defining `t` in `OVect` as

```
type t = <length: int; get: int -> float; ..>
```

where the ellipsis "`..`" allows extra methods. But free type variables are not allowed in types definition (think of `type t = 'a`,) and the "`..`" in the above type represents an internal type variable, usually called the *row variable*, which is free here. The first solution that comes to mind is to do as we would with normal type variables, and define an abstract type corresponding to this "`..`".

```
type t_row
type t = <length: int; get: int -> float; t_row>
```

This requires the ability to name the row variable, which is anonymous in OCaml. We formalize this idea at the beginning of section 3. We also find that it is only a first step, as incremental refinement of type definitions would be clumsy, and this formalization cannot fully handle polymorphic variant types.

A better approach to this problem is to find a middle-ground between abstract types, which are completely opaque, and concrete types, which cannot be further refined.

One option to introduce such semi-abstract types would be to exploit subtyping: one might allow defining upper or lower bounds for abstract types. This is the idea behind F-bounded polymorphism [7], which has been integrated into a number of languages such as Generic Java [8], Moby [9], or Scala [10]. In particular, Moby and Scala do have a module system able to express functors, and Scala gives an elegant solution to the expression problem [11].

In a language offering complete type inference, like Objective Caml does, subtyping has to be explicit, if we are to keep types simple. This makes the F-bounded polymorphism approach impractical, because any use of a value whose type is semi-abstract would require an explicit coercion. It is more natural to stick with the fully structural approach inherent to OCaml, simply abstracting extensibility (rather than the whole type) as if it were a type variable. This means that we actually follow the idea of adding an abstract `t_row`, but that we will keep it unnamed. Here is our syntax for it.

```
type t = private <length: int; get: int -> float; ..>
```

A private row type[2] is defined by a structural type, either object or variant, where the only free type variable is the row variable. Superficially, this looks exactly like the definition we just rejected as not well-formed. But here the "private" keyword implicitly binds the row variable as an anonymous abstract type, at the same level as the type definition. Using this definition in `OVect`, the functor `OPoly` now accepts any object type having at least the methods `length` and `get` with proper types.

There have been examples in the past combining classes with functors. Such a combination has been used by the FOC project for instance [12]. But in the absence of private row types, classes were only used to provide late-binding at the value level, and classes or object types did not appear in parameters of functors. We will also see that

[2] The "private" part of the naming will get clearer in section 2.2. The qualifiers "row" and "structural" are more or less interchangeable in this paper. The author somehow prefers structural, but some people seem to find the concept of row easier to grasp.

private row types, in combination with recursive modules, are even more interesting for polymorphic variants, as they provide a powerful way to structure programs using them.

The body of this paper is composed of two sections. The next one presents various examples using private row types, for functors, privacy, and extensible recursion. Section 3 formalizes the definitions, combining structural polymorphism with applicative functors.

2 Using Private Row Types

In this section we give examples of various uses of private row types, in combination with other features. All examples were type-checked using Objective Caml 3.09. The only new syntax compared to previous versions of the language is the "`private`" keyword, which indicates a private row type. While some function definitions contain type annotations, they are only there for demonstrative purposes, and the definitions would still be typable without them, leading to a more general type —*i.e.* type inference is still principal.

2.1 Simple Functors

Private row types are essential in combining functors with structural polymorphism. A natural application is our introduction example. For definitions prefixed with `#`, we show in italic the types inferred, as in an interactive session.

```
module type OVect = sig
  type t = private <length: int; get: int -> float; ..>
  val init : int -> (int -> float) -> t
end
# module OPoly (V : OVect) = struct ... end ;;
module OPoly : functor (V: OVect) -> sig val add : V.t -> V.t -> V.t end
```

We can develop it more, by adding a `map` method and using it in a function `mul` for external product.

```
module type OVect2 = sig
  type t = private
      <length: int; get: int -> float; map: (float -> float) -> t; ..>
  val init : int -> (int -> float) -> t
end
# module OPoly2 (V : OVect2) = struct
    include OPoly(V)
    let mul x (p : V.t) = p#map (fun y -> x *. y)
  end ;;
module OPoly2 : functor (V : OVect2) ->
  sig
    val add : V.t -> V.t -> V.t
    val mul : float -> V.t -> V.t
  end
```

Since we wish to extend OPoly, we include an instance of it. Note how we pass an argument of type OVect2 to OPoly which expects an OVect. This is accepted as OVect2.t is an instance of OVect.t.

Another typical case where we need to use functors with objects, is when the functionality we need is already provided as a functor.

```
module OMap(X : sig type t = private <compare : t -> int; ..> end)
  = Map.Make(struct type t = X.t let compare (x:t) y = x#compare y end)
class vector (n : int) (f : int -> float) = object (s : 's)
  val v = Array.init n f
  method length = n
  method get i = v.(i)
  method map f = < v = Array.map f v >
  method compare (vec : 's) = compare v (Array.init vec#length vec#get)
end
module VMap = OMap(struct type t = vector end)
module VPoly = OPoly2(struct type t = vector let init = new vector end)
```

Here the functor Map.Make from the standard library expects a type t and a function compare : t -> t -> int. Since t is not allowed any polymorphism, we have to wrap it in a new functor expecting only one type, which provides this time a method compare. We define a class vector —which implicitly also defines a type vector for its objects—, with all the methods required by OMap and OPoly2, so we can pass its type as parameter to both. Here the type annotations on f and vec are required, as class definitions may not contain free type variables.

Examples involving polymorphic variants also arise naturally. Consider for instance a simple property base, such that we may add new types of properties.

```
type basic = [‘Bool of bool | ‘String of string]
module Props(X : sig type t = private [> basic] end) =
  struct
    let base : (string,X.t) Hashtbl.t = Hashtbl.create 17
    let put_bool k b = Hashtbl.add k (‘Bool b)
    let put_str k s = Hashtbl.add k (‘String s)
    let to_string (v : X.t) = match v with
        ‘Bool b   -> if b then "true" else "false"
      | ‘String s -> s
      | _ -> "other" (* required by typing *)
  end
```

The notation [> basic] is an abbreviation for [> ‘Bool of bool | ‘String ofstring]. It means that the actual variant type X.t will have to contain at least the constructors of basic, and eventually more. The “>” implies the presence of a row variable. This notation is not new to this proposal, but the “private” keyword is needed to bind the implicit row variable in a type definition. An interesting consequence of extensibility is that any pattern-matching on X.t needs to contain a default case, as it may actually contain more cases than basic. This is similar to Zenger&Odersky’s approach to extensible datatypes, which also requires defaults [13].

In order to extend this basic property type, we only need to define a new type and apply the functor.

```
# type extended = [basic | `Int of int] ;;
type extended = [ `Bool of bool | `Int of int | `String of string ]
# module MyProps = Props(struct type t = extended end) ;;
module MyProps :
  sig
    val base : (string, extended) Hashtbl.t
    val put_bool : string -> bool -> unit
    val put_str : string -> string -> unit
    val to_string : extended -> string
  end
```

Note that here, `extended` is a "final" type, not extensible, thus we may write complete pattern-matchings for it. We may want to use this property to refine the `to_string` function. The notation `#basic` is an abbreviation for the or-pattern collecting all cases from `basic`, *i.e.* (`` `Bool _ | `String _ ``).

```
# let to_string (v : extended) = match v with
    `Int n -> string_of_int n
  | #basic -> MyProps.to_string v ;;
val to_string : extended -> string
```

The functorial approach is also useful when combining polymorphic variants and mutable values. It allows to extend the type of a polymorphic variant in a different compilation unit, which was not possible before. Here is an example which causes a compile time error.

```
(* base.ml *)
type basic = [`Bool of bool | `String of string]
let base : (string, [>basic]) Hashtbl.t = Hashtbl.create 17
$ ocamlc -c base.ml
File "base.ml", line 2, characters 41-58:
The type of this expression, (string, _[> basic ]) Hashtbl.t,
contains type variables that cannot be generalized
```

Since `base` is not a value, its type cannot be made polymorphic. A final type for it should be determined in the same compilation unit. Since no such type is given here, this results in an error. Using the above functor avoids the problem, by delaying the creation of the hash table to the application of the functor. Note that using a functor means that any code accessing the property base must be functorized too. This is a classical downside of doing linking through functor application. As a counter part, this enhances modularity, allowing to use several property bases in the same program for instance.

2.2 Relation to Private Types

Since version 3.07, released in 2003, Objective Caml has *private types*, introduced by Pierre Weis [14]. Like private row types, private types are intended to appear in signatures, abstracting some behavior of the implementation. To do that, they simply restrict (non-polymorphic) variants and records, prohibiting the creation of values outside of

the module where they were defined, while still allowing pattern-matching or field access. Contrary to private row types, they do not allow refinement of type definitions. Their main intent is to allow to enforce invariant properties on concrete types, like it is possible with abstract datatypes, while avoiding any overhead.

```
module Relative : sig
  type t = private Zero | Pos of int | Neg of int
  val inj : int -> t
end = struct
  type t = Zero | Pos of int | Neg of int
  let inj n = if n=0 then Zero else if n>0 then Pos n else Neg (-n)
end
# open Relative ;;
# let string_of_rel = function
    Zero -> "0"
  | Pos n -> string_of_int n
  | Neg n -> "-" ^ string_of_int n;;
val string_of_rel : rel -> string
# Zero;;
Cannot create values of the private type Relative.t
```

Interestingly, we can simulate private types with private row types. The kind of variant refinement used here is opposite to the previous section: we model restrictions on construction by assuming that some constructors may actually not be there. This gives us more flexibility than with the original private types, as some constructors may be declared as present, to make them public.

```
module Relative : sig
  type t = private [< `Zero | `Pos of int | `Neg of int > `Zero]
  val inj : int -> t
end = struct
  type t = [`Zero | `Pos of int | `Neg of int]
  let inj n = if n=0 then `Zero else if n>0 then `Pos n else `Neg (-n)
end
# let zero : Relative.t = `Zero;;
val zero : Relative.t = `Zero
# let one : Relative.t = `Pos (-1);;
This expression has type [> `Pos of int ] but is here used with type
  Relative.t
```

The private definition of `t` has one public constructor, `` `Zero ``, as implied by the "`> `Zero`" bit of the definition, which says that it must be present in the implementation, but `` `Pos `` and `` `Neg `` are allowed to be absent, so they are private. As a result, `` `Zero `` can be given type `Relative.t`, but `` `Pos(-1) `` cannot, which protects abstraction.

Private record types can be modeled by object types, this time in the usual way. As an extra feature we naturally gain the possibility of hiding some fields. This allows to define module-private (or friend) methods, like in Java, while OCaml only has object-private methods.

```
module Vector : sig
  type 'a c = private
      < length: int; get: int -> 'a; compare: 'a c -> int; .. >
  val init : int -> (int -> 'a) -> 'a c
  val map : ('a -> 'b) -> 'a c -> 'b c
end = struct
  class ['a] c v = object (s : 's)
    method v = v
    method length = Array.length v
    method get i : 'a = v.(i)
    method compare (vec : 's) = compare v vec#v
  end
  let init n f = new c (Array.init n f)
  let map f v = new c (Array.map f v#v)
end
```

Here we have used a private object type to hide the method `v`, while enforcing its presence in the actual object. This allows accessing the contents of the object in a more efficient way. If `v` were visible outside of `Vector`, encapsulation would be broken, as one could use it to mutate these contents.

One might think that it would be enough to use an abstract type for the array returned by `v`, without hiding `v` itself. However, object typing in OCaml is purely structural: one can freely create an object by hand, and give it the same type as an existing class, even though its methods might cunningly call methods from different objects, breaking the coherence of the definitions. Only private object types can protect against this, while still allowing the programmer to call methods in a natural way. As with private types, this allows to enforce invariants, for instance saying that for a value `v` of type `Vector.c`, calling `v#get i` always succeeds when $0 \leq$ `i` $<$ `v#length`.

Note that private object types do not interact directly with classes, and as such they are not as expressive as abstract views for instance [15]. In particular one cannot inherit from a private type.

2.3 Recursion and the Expression Problem

Examples in previous sections have kept to a simple structure. In particular, the variant types involved were not recursive. As we indicated in introduction, polymorphic variants are known to provide a very simple solution to the expression problem, allowing one to extend a recursive type with new constructors, with full type safety, and without any recompilation. However, the original solution has a small drawback: one has to close the recursion individually for each operation defined on the datatype. Moreover it relies quite heavily on type inference to produce polymorphic types.

With the introduction of recursive modules, a natural way to make things more explicit is to close the recursion at the module level. However, this also requires private row types, to allow extension without introducing mind-boggling coercions (see `mixmod.ml` at [4] for an example with coercions.)

We present here a variation on the expression problem, where we insist only on the addition of new constructors, since adding new operations is trivial in this setting. If you find it difficult to follow our approach, reading [4] first should help a lot. We first define a module type describing the operations involved.

```
module type Ops = sig
  type expr
  val eval : expr -> expr
  val show : expr -> string
end
```

We then define a first language, with only integer constants and addition. To keep it extensible, we leave the recursion open in the variant type, and have operations recurse through the parameter of a functor.

```
module Plus = struct
  type 'a expr0 = [`Num of int | `Plus of 'a * 'a]
  module F(X : Ops with type expr = private [> 'a expr0] as 'a) =
    struct
      type expr = X.expr expr0
      let eval : expr -> X.expr = function
          `Num _ as e  -> e
        | `Plus(e1,e2) -> match X.eval e1, X.eval e2 with
                            `Num m, `Num n -> `Num(m+n)
                          | e12            -> `Plus e12
      let show : expr -> string = function
          `Num n -> string_of_int n
        | `Plus(e1,e2) -> "("^X.show e1^"+"^X.show e2^")"
    end
  module rec L : (Ops with type expr = L.expr expr0) = F(L)
end
```

Observe how closing the recursion is now easy: we just have to take a fix-point of the functor.

The next step is to define a second language, adding multiplication. Inside the functor, we instantiate the original addition language, and use it to delegate known cases in operations, using variant dispatch.

```
module Mult = struct
  type 'a expr0 = ['a Plus.expr0 | `Mult of 'a * 'a]
  module F(X : Ops with type expr = private [> 'a expr0] as 'a) =
    struct
      type expr = X.expr expr0
      module L = Plus.F(X)
      let eval : expr -> X.expr = function
          #L.expr as e -> L.eval e
        | `Mult(e1,e2) -> match X.eval e1, X.eval e2 with
                            `Num m, `Num n -> `Num(m*n)
                          | e12            -> `Mult e12
      let show : expr -> string = function
          #L.expr as e -> L.show e
        | `Mult(e1,e2) -> "("^X.show e1^"*"^X.show e2^")"
    end
  module rec L : (Ops with type expr = L.expr expr0) = F(L)
end
```

That's it. Here is a simple example using the final language.

```
# Mult.L.show(‘Plus(‘Num 2,‘Mult(‘Num 3,‘Num 5)));;
- : string = "(2+(3*5))"
```

This whole approach may seem verbose at first, but a large part of it appears to be boilerplate. Half of the lines of `Plus` have to be repeated in `Mult`, and would actually be in any similar code. From a more theoretical point of view, this example makes clearer the relation between solutions to the expression problem that use type abstraction, such as [11], and our original solution which used only polymorphism.

Combining object types with recursive modules also has applications, but they are less immediate, as classes already provide a form of open recursion.

3 Formalization

Before giving a complete formalization, we first describe a much simpler one, which is limited to private object types. The idea is to formalize objects as rows, in the style of Rémy [16]. Here are our core types.

$$\begin{array}{ll} \nu ::= \alpha \mid t(\vec{\tau}\vec{\rho}) & \text{abstractions} \\ \tau ::= \nu \mid \tau \to \tau \mid \langle \rho \rangle & \text{types} \\ \rho ::= \nu \mid \emptyset \mid l : \tau; \rho & \text{rows} \\ k ::= \star \mid \diamond & \text{kinds} \\ \sigma ::= \tau \mid \forall \alpha{:}k.\sigma & \text{polytypes} \end{array}$$

Types are composed of abstractions, function types, and object types. An object type is described by a row, which is a list of pairs label-type, terminated either by the empty list or an abstraction. Abstractions are either type variables or abstract types (which may have parameters, types or rows.) In order to indicate the contexts where an abstraction may be used, we introduce two kinds: $\star$ for types and $\diamond$ for rows. We allow fields to commute in rows, that is

$$l_1 : \tau_1; l_2 : \tau_2; \rho = l_2 : \tau_2; l_1 : \tau_1; \rho \quad \text{if } l_1 \neq l_2$$

The same label may occur twice in a row (as for labeled arguments [17].) This simplifies kinds —they don't need to track which labels are used—, but this has no practical impact, as there is no way to create such an object.

If we start with this core type system, moving to the module level is trivial: we just need to add kinds to abstract types. This creates no difficulty, as Leroy's modular module system already handles simple kinds [18]. In such a system, the signature `OVect` would be:

```
module type OVect = sig
  type t_row : ◇
  type t : ⋆ = <length: int; get: int → float; t_row>
  val init : int → (int → float) → t
end
```

Then defining a particular instance just requires providing a concrete definition for `t_row`.

Unfortunately, type refinement in this system proves to be very clumsy. The trouble is that the natural encoding of `OVect2` would not be an instance of `OVect`. We need extra type definitions to make it possible.

```
module type OVect2 = sig
  type t_row' : ◇
  type t_row : ◇ = map : (float → float) → t; t_row'
  type t : ⋆ = <length: int; get: int → float; t_row>
  val init : int → (int → float) → t
end
```

The fact one has to change the name of the abstract row is particularly confusing.

This clumsiness leads to our implicit syntax for private row types: rather than make abstract rows explicit, and have them pollute signatures, we prefer to leave them implicit, just indicating their presence. Implementations do not need to give a concrete definition for abstract rows, as the type system can recover them by comparing a private type definition and its implementation. Technically this amounts to an extension of the subtyping relation for modules. And as we keep rows implicit, we can omit kinds from the surface language.

We might have gone even further, and allowed any free variable to be automatically converted into an anonymous abstract type. We refrained from this for two reasons. This contradicts the principle of minimality in language changes, and this doesn't fit well the intuition of "private" type. Yet this might be an interesting choice when designing a more implicit type system for modules.

While this sketch of a formalization gives a good intuition of what private row types are, sufficient for practical uses, we will use a different formalization for our core language. The main reason is that this system does not extend nicely to private variant types. As can be seen in Rémy's paper, allowing variant tags to disappear from a type require additional *presence* variables. If we were to apply this scheme, we would need an abstract presence type for each constructor we want to keep private, adding a lot of complexity[3].

We provide in the rest of this section a condensed description of the formal system underlying private row types. It is based on our formalism for structural polymorphism [19] for the core language part, combined with Leroy's description of an applicative functor calculus [20]. A combination of these two systems already provides a complete description of Objective Caml's type system (without polymorphic methods, labeled parameters, and extensions.)

We will not give full details of these two systems, as both of them are rather complex, yet very few changes are needed. One is the ability to specify inside structural types that they have an identity (a name), and are only compatible with types having the same identity. The other is to allow refining private row types through module subtyping, and check that all such refinements are legal.

While we will still internally use an abstract type to represent a "virtual" row variable, the formalism we describe here does not have explicit row variables. It is rather

[3] The internal representation of polymorphic variant types in the Objective Caml compiler does use such presence variables, but they are not shown to the programmer, and they are not abstracted individually.

$$
\begin{array}{rll}
\tau ::= & \alpha & \text{type variable} \\
 & \mid t(\vec{\tau}) & \text{abstract type} \\
 & \mid \tau \to \tau & \text{function type} \\
\mathrm{K} ::= & \emptyset \mid \mathrm{K}, \alpha :: (C,R) & \text{kinding environment} \\
\theta ::= & \tau \mid \mathrm{K} \triangleright \tau & \text{kinded type} \\
\sigma ::= & \theta \mid \forall \vec{\alpha}.\theta & \text{polytype}
\end{array}
$$

Fig. 1. Types and kindings

$$
\begin{array}{l}
<l_1 : \tau_1; \ldots; l_n : \tau_n; ..> \stackrel{\text{def}}{=} \alpha :: (\mathsf{o}, \{l_1, ..., l_n\}, \mathcal{L}, 0, \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\}) \triangleright \alpha \\
<l_1 : \tau_1; \ldots; l_n : \tau_n> \stackrel{\text{def}}{=} \alpha :: (\mathsf{o}, \{l_1, ..., l_n\}, \{l_1, ..., l_n\}, 0, \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\}) \triangleright \alpha \\
[> l_1 \text{ of } \tau_1 \mid \ldots \mid l_n \text{ of } \tau_n] \stackrel{\text{def}}{=} \alpha :: (\mathsf{v}, \{l_1, ..., l_n\}, \mathcal{L}, 0, \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n)\}) \triangleright \alpha \\
[< l_1 \text{ of } \tau_1 \mid \ldots \mid l_n \text{ of } \tau_n > l_1 \ldots l_k] \stackrel{\text{def}}{=} \\
\qquad \alpha :: (\mathsf{v}, \{l_1, ..., l_k\}, \{l_1, ..., l_n\}, 0, \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\}) \triangleright \alpha
\end{array}
$$

Fig. 2. Kindings corresponding to surface syntax

based on an expressive kinding relation [6], which describes constraints on types rather than simply categories.

3.1 Core Type System

We will directly use the formalism from [19], as it is already general enough. We only have to add parameterized abstract types. This section may seem obscure without a good understanding of the formalism used, yet understanding figure 2 and the entailment relation should be sufficient to go on to the module level. An important point is that the definitions here ensure automatically subject reduction (leading to type soundness) and principal type inference, without need of extra proofs.

The syntax for types and kindings is given in figure 1. Simple types τ are defined as usual. They include type variables, function types, and named abstract types with type parameters. Polytypes σ are extended with a kinding environment K that restricts possible instances for constrained variables. K is a set of bindings $\alpha :: (C,R)$, C a constraint and R a set of relations from labels to types, describing together the possible values admitted for the type α. There is no specific syntax in types for object and variants, as they are denoted by type variables constrained in a kinding environment. The kindings corresponding to the syntax used in previous sections, using the constraint domain defined lower, are given in figure 2, respectively for open or closed, object and variant types. The only relation we use in kindings, $\mapsto$, is not a function: a label may be related to several types. Recursive types can be defined using a mutually recursive kinding environment, *i.e.* where kinds are related to each other. It should be clear by now that the notion of kind in this type system bears no resemblance to the simple kinds we considered first. Note that we only introduce abstract types here; type abbreviations can be seen as always expanded.

In order to have a proper type system, we only need to define a constraint domain. Our constraint domain includes both object and variant types, and support for identifying a type by its name. We assume a set $\mathcal{L}$ of labels, denoting methods or variant

constructors. $\mathcal{L}$ includes a special label *row* used to encode our virtual row. The C in a kind is an element of the following set.

$$(k, L, U, p) \in \{\mathsf{o}, \mathsf{v}\} \times P_{fin}(\mathcal{L}) \times (P_{fin}(\mathcal{L}) \cup \{\mathcal{L}\}) \times \{0, 1\}$$

k distinguishes objects and variants. L represents a lower bound on available methods or constructors (*required* or *present* ones), and should be a finite subset of $\mathcal{L}$. U represents an upper bound, and should be either a finite subset of $\mathcal{L}$, or $\mathcal{L}$ itself. p is 0 for normal types, 1 for private types, and will be used at the module level. For both of objects and variants, we obtain a "final" (non-refinable) type by choosing $L = U$.

We define an entailment relation on constraints, noted "$C \models C'$", which is reflexive and transitive. We first distinguish inconsistent constraints.

$$(\mathsf{o}, L, U, p) \models \bot \text{ if } U \neq L \text{ and } U \neq \mathcal{L}$$
$$(\mathsf{v}, L, U, p) \models \bot \text{ if } L \not\subset U$$

An object type can only be extensible or final: its upper bound is either L or all labels. On the other hand, a variant type with a finite upper bound may still be refined by removing tags, so that the only restriction is that the lower bound should be included in the upper bound.

Entailment can refine a constraint as long as it is not private. Note that refinement goes backward: a variable with the kind on the right of the entailment relation can be instantiated to one with the kind on the left.

$$(k, L', U', p) \models (k, L, U, 0) \text{ if } L \subset L' \text{ and } U \supset U'$$

Next we use our constraints to selectively propagate type equalities. For a constraint $C = (k, L, U, p)$ and a label l:

$$C \vdash \mathit{uniq}(l) \stackrel{\text{def}}{=} k = \mathsf{o} \vee l \in L \vee (p = 1 \wedge l \in U) \vee l = \mathit{row}$$
$$l \mapsto \alpha_1 \wedge l \mapsto \alpha_2 \wedge \mathit{uniq}(l) \Rightarrow \alpha_1 = \alpha_2.$$

The first line defines a predicate *uniq*, denoting when only one type can be associated to a label. The second line is a propagation rule. It means that, for a kind (C, R), when a label satisfies the property *uniq*, then types associated to this label in R should be unified. In the original system without private rows, the definition of *uniq* was $k = \mathsf{o} \vee l \in L$, meaning that unification is triggered either if we consider an object type, or a required label in a variant type. Now it is also triggered for possible labels in private variant types. That is, all possible labels in private types must have unique types. Combined with that fact their constraint cannot be further refined, this ensures that no typing information will be added to them. The special label *row* is always unique, and will be associated to an abstract type denoting the identity of a private row type.

It is easy to see that these definitions satisfy the conditions for a valid constraint domain, as stated in [19].

Note that this extension of the core type system is also required in order to handle first-class polymorphism, available through polymorphic methods and record fields. In that case, *row* is only associated with a universal type variable.

3.2 Module Type System

The second part is at the module level: we must introduce private type definitions, and allow refinement through module subtyping. In order to formalize this, we will switch to Leroy's module calculus [20], which has 4 kinds of judgements: well-formedness ($E \vdash \sigma$ type), module typing ($E \vdash s : S$), type equivalence ($E \vdash \theta \approx \theta'$), and module subtyping ($E \vdash S <: S'$.) We will proceed by adding and modifying rules in this calculus, without reproducing all rules for the sake of space.

Leroy leaves the base language unspecified. We have to be more specific, in particular allowing parameterized type definitions. We will see manifest type definitions as kinded types: $\texttt{type}\ t_i(\vec{\alpha}) = \mathrm{K} \triangleright \tau$. Note that while variables of refinable kinds must all appear in $\vec{\alpha}$, as there is no way to quantify a variable explicitly outside of the type definition, variables whose kind is no longer refinable, *i.e.* either $L = U$ or $p = 1$, are seen as implicitly quantified, and may appear in K but not in $\vec{\alpha}$. "$E \vdash \sigma\ \texttt{type}$" checks that σ is a valid polytype under environment E, and that no refinable type variable is free.

The basic typing rule for type definitions is unchanged, up to our addition of type parameters.

$$\frac{E \vdash \forall\vec{\alpha}.\theta\ \texttt{type} \quad t_i \notin BV(E) \quad E;\texttt{type}\ t_i(\vec{\alpha}) = \theta \vdash s : S}{E \vdash (\texttt{type}\ t_i(\vec{\alpha}) = \theta; s) : (\texttt{type}\ t_i(\vec{\alpha}) = \theta; S)}$$

As it does not handle directly private row types, we first need to translate private definitions into normal ones, both inside modules and signatures. As we have explained before, we do it by defining an abstract type t_{row} along with the manifest type t, using it as row.

$$\texttt{type}\ t_i(\vec{\alpha}) = \texttt{private}\ \theta_0 \stackrel{\triangle}{=} \texttt{type}\ t_{rowi}(\vec{\alpha}); \texttt{type}\ t_i(\vec{\alpha}) = \theta$$
$$\text{where} \begin{array}{l} \theta_0 = K, \beta :: (k, L, U, 0, R) \triangleright \beta \quad L \neq U \\ \theta\ = K, \beta :: (k, L, U, 1, R \cup \{row \mapsto t_{rowi}(\vec{\alpha}))\}) \triangleright \beta \end{array}$$

θ_0 is a row type, with a single non-quantified refinable type variable β. In θ, we make its kind private, and mark it with the abstract type t_{rowi}, which is defined along t_i.

Once we have introduced private row types, we should allow refinement through subtyping. However, the standard approach of having t_{rowi} manifest on one side, and abstract on the other, will not work here, as we want to allow the enclosing kinds to be different. Here is the original rule for subtyping.

$$\frac{E \vdash \theta \approx \theta'}{E \vdash (\texttt{type}\ t_i(\vec{\alpha}) = \theta) <: (\texttt{type}\ t_i(\vec{\alpha}) = \theta')}$$

As you can see, the trouble here is that this rule is limited to equivalent type representations. In order to accommodate refinement, we add a new rule, using entailment.

$$\frac{\begin{array}{c}(k, L, U, 0) \models (k, L', U', 0) \quad E \vdash K \approx K' \quad row \mapsto t_{rowi}(\vec{\alpha}) \in R' \\ (\forall l)\ \ l \mapsto \tau \in R \wedge l \mapsto \tau' \in R' \Rightarrow E \vdash \tau \approx \tau'\end{array}}{\begin{array}{l}E \vdash (\texttt{type}\ t_i(\vec{\alpha}) = K, \beta :: (k, L, U, p, R) \triangleright \beta) \\ \qquad\qquad <: (\texttt{type}\ t_i(\vec{\alpha}) = K', \beta :: (k, L', U', 1, R') \triangleright \beta)\end{array}}$$

This rule says that, a row type definition (either private or not) subsumes a private row type definition when: (1) the original definition entails the private one (both assumed

public), (2) kinding environments K and K' are identical, up to the equivalence of the types they contain, (3) all labels common to both definitions are associated to equivalent types, which also implies that if $row \mapsto \tau \in R$, then $E \vdash \tau \approx t_{rowi}(\vec{\alpha})$. The requirement $row \mapsto t_{rowi}(\vec{\alpha}) \in R'$ additionally ensures that the abstract row is declared inside the same signature.

Another slight modification we need is to allow the introduction of hidden types in subtyping. This accounts for two situations. The first one is when the original type definition is public, and we make it private through subtyping. We need to introduce a new abstract t_{rowi} in the subtype, matching the implicit one in the supertype.

$$\frac{t_{rowi} \notin BV(D_i) \quad (1 \leq i \leq n)}{E \vdash \texttt{sig}\ D_1;...;D_n\ \texttt{end} <: \texttt{sig}\ D_1;...;D_k;\texttt{type}\ t_{rowi}(\vec{\alpha});D_{k+1};...;D_n\ \texttt{end}}$$

The second one occurs when we define a type alias for a private type, and then export it as being itself a private type. Here is an example.

```
module M : sig type t = private [> `A] end = struct
  module M1 = struct type t = private [> `A | `B] end
  type t = M1.t
end
```

We need to add $\texttt{type}\ t_{rowi} = M_1.t_{rowi}$ in the signature of our implementation, in order to use the subtyping rule for private row types:

$$\frac{t_{rowi} \notin BV(D_i) \quad D_k = (\texttt{type}\ t_i(\vec{\alpha}) = K, \beta :: (k,L,U,1,R) \triangleright \beta) \quad row \mapsto \tau \in R}{E \vdash \texttt{sig}\ D_1;...;D_k;S\ \texttt{end} <: \texttt{sig}\ D_1;...;D_{k-1};\texttt{type}\ t_{rowi}(\vec{\alpha}) = \tau;D_k;S\ \texttt{end}}$$

These rules together provide a complete formalization of private row types.

3.3 Extra Features

Independently of these questions of formalism, another issue appears with the introduction of the `with` construct for signatures. This construct is not present in [20], but it is needed in practice for any implementation, to avoid expanding all signatures by hand. We are using it in our own example of section 2.3. The technical difficulty with `with` comes from the fact it only substitutes one definition at a time, and the environment of the signature to be modified is not available in the new definition. It had to be extended to allow private row types, particularly recursive ones. This is not yet enough for mutually recursive types, and it seems that there are approaches more promising than `with` to manipulate signatures [21].

A last design decision is related to the handling of variance. In order to allow more subtyping, in OCaml both abstract types and algebraic datatypes have variances associated to their type parameters. For instance the type $\texttt{list}(\alpha)$ is covariant, which can be written $\texttt{type}\ \texttt{list}(+\alpha)$ in its type definition. For abstract types variance annotations are explicit, but for algebraic datatypes they are inferred from the definition of the type. As private row types have a structural definition, one might think of inferring their variance. However, the presence of an associated abstract type clearly indicates that variance should be explicit. This also means that this variance must be respected:

i.e. an implementation should have a stronger variance than the private row type it replaces, and variance can only be weakened through subtyping. This reasoning can be used to explain why private types, while they do not allow refinement, use also explicit variances.

4 Conclusion

We have introduced a new form of type definition, which is both manifest and abstract at the same time. We branded it as private, as it behaves in a way very similar to both private types in OCaml, and private methods as they are understood in Java. Nonetheless, the power of this new feature is not limited to privacy, but goes a long way towards abstraction allowing incremental extension. As this feature relies heavily on the expressive power of modules, it is most interesting when combined with recent extensions of module systems, such as recursive modules [22,23,24] or, in an hopefully close future, combinable signatures [21].

Another desirable addition is support for unions of private variant types. One can already define unions of concrete polymorphic variant types, and use them through dispatch. The private case is more complex, as one must ensure that the combined types are compatible. We are currently working on this question.

Acknowledgements

Comments from Didier Rémy, Keiko Nakata, Romain Bardou, and anonymous referees were a great help in improving this paper. I thank them all.

References

1. Rémy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML. Theory and Practice of Object Systems **4** (1998) 27–50
2. Garrigue, J.: Programming with polymorphic variants. In: ML Workshop, Baltimore (1998)
3. Wadler, P.: The expression problem. Java Genericity mailing list (1998) `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`.
4. Garrigue, J.: Code reuse through polymorphic variants. In: Workshop on Foundations of Software Engineering, Sasaguri, Japan (2000) `http://www.math.nagoya-u.ac.jp/~garrigue/papers/fose2000.html`.
5. Rémy, D., Garrigue, J.: On the expression problem. `http://pauillac.inria.fr/~remy /work/expr/`(2004)
6. Ohori, A.: A polymorphic record calculus and its compilation. ACM Transactions on Programming Languages and Systems **17** (1995) 844–895
7. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proc. ACM Symposium on Functional Programming and Computer Architectures. (1989) 273–280
8. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: Proc. ACM Symposium on Object Oriented Programming, Systems, Languages and Applications. (1998)

9. Fisher, K., Reppy, J.: The design of a class mechanism for Moby. In: Proc. ACM Conference on Programming Language Design and Implementation. (1999)
10. Odersky, M., Crémet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Proc. European Conference on Object-Oriented Programming. (2003)
11. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: Workshop on Foundations of Object-Oriented Languages. (2005)
12. Boulmé, S., Hardin, T., Rioboo, R.: Polymorphic data types, objects, modules and functors: is it too much? RR 014, LIP6, Université Paris 6 (2000)
13. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: Proc. ACM International Conference on Functional Programming. (2001) 241–252
14. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.09, Documentation and user's manual. Projet Cristal, INRIA. (2005)
15. Vouillon, J.: Combining subsumption and binary methods: an object calculus with views. In: Proc. ACM Symposium on Principles of Programming Languages. (2001) 290–303
16. Rémy, D.: Type inference for records in a natural extension of ML. In Gunter, C.A., Mitchell, J.C., eds.: Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press (1993)
17. Garrigue, J., Aït-Kaci, H.: The typed polymorphic label-selective λ-calculus. In: Proc. ACM Symposium on Principles of Programming Languages. (1994) 35–47
18. Leroy, X.: A modular module system. Journal of Functional Programming **10** (2000) 269–303
19. Garrigue, J.: Simple type inference for structural polymorphism. In: Workshop on Foundations of Object-Oriented Languages, Portland, Oregon (2002)
20. Leroy, X.: Applicative functors and fully transparent higher-order modules. In: Proc. ACM Symposium on Principles of Programming Languages. (1995) 142–153
21. Ramsey, N., Fisher, K., Govereau, P.: An expressive language of signatures. In: Proc. ACM International Conference on Functional Programming. (2005)
22. Crary, K., Harper, R., Puri, S.: What is a recursive module? In: Proc. ACM Conference on Programming Language Design and Implementation. (1999) 50–63
23. Russo, C.V.: Recursive structures for Standard ML. In: Proc. ACM International Conference on Functional Programming. (2001) 50–61
24. Nakata, K., Garrigue, J.: Recursive modules for programming. In: Proc. ACM International Conference on Functional Programming, Portland, Oregon (2006)