Private row types: abstracting the unnamed

Jacques Garrigue

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

March 15, 2006

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 to 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 [15]. Similarly, a list of polymorphic variant values can be used in different contexts expecting different numbers of constructors, as long as the types of constructor arguments agree, and all constructors present in the list are allowed [6].

These new types are particularly interesting in programming situations where one gradually extends a type witha 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 also true 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 simultaneoulsy enriches a small expression language with new constructs and new operations [17]. 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 [7] or classes [14].

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 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.

```
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. This works nicely, but what happens if we want to represent vectors as objects, calling methods inside the functor?

```
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) =
    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
```

This looks like it works again, but there is a hidden limitation in the above code: only methods `length` and `get` would be allowed. That is, we cannot use an object with any extra method. What we would like is actually the above signature to contain

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

allowing extra methods. Yet the "`..`" in the above type represents an internal type variable, usually called the row-variable, and free type variables are not allowed in type definitions. With an usual type variable, we could define an abstract type and use it in place of the type variable. This would amount to something like

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

However row-variables cannot be "named" in this way, so we cannot abstract them.

What we need to overcome this problem is 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 [3], which has been integrated into a number of languages such as Generic Java [2], Moby [5], or Scala [11]. In particular, Moby and Scala do have a module system able to express functors, and Scala gives an elegant solution to the expression problem [19].

We avoided F-bounded polymorhism, because subtyping in Objective Caml is fully explicit: any use of a value whose type is semi-abstract would have required a coercion. We rather chose to stick with the fully structural approach inherent to OCaml, simply abstracting extensibility as if it were a type variable. This means that we follow the idea of adding an abstract `t_row`, but that we will keep it unnamed. Here is an example, using our syntax for private row types[1].

```
module type OVect = sig
  type t = private <length: int; get: int -> float; ..>
  val init : int -> (int -> float) -> t
end
```

With this definition, the functor `OPoly` now accepts any object type having at least the methods

[1]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 most people seem to find the concept of row easier to grasp.

`length` and `get` with proper types. As seen here, a private row type is defined by a structural type, either object or variant, where the only free type variable is the row variable.

As the solution looks so simple, one might wonder why it was not introduced long ago. One answer is that the problem doesn't occur if one writes everything using classes rather than functors (but then why have both in the same language?) Another is that the `t_row` above just describes a correct intuition, while formalization is a bit trickier.

There has been examples in the past combining classes with functors. Such a combination has been used by the FOC project for instance [1]. 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 private row types, in combination with recursive modules, are even more interesting for polymorphic variants, as variants have no class syntax to express late-binding.

Private row types take their name from their relation to private types. As a side-effect of abstraction, private row types can restrict the ability to create values of a given type. This is already nice in itself, as it provides private abbreviations for free. This also offers a way of understanding the meaning of private types, as types whose structure is partially abstract.

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.

2.1 Simple functors

Private row types are essential in combining functors with structural polymorphism. A natural application is our introduction example:

```
module type OVect = sig
  type t = private <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) =
    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
```

We can develop it more, by extending with a new 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
```

We added an extra method `map` to `t`, and used it inside `Poly2`. We also passed an argument of type `OVect2` to `OPoly` which expected an `OVect`. This is accepted as `OVect2.t` is an instance of `OVect.t`.

An other 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` with all the methods required by `OMap` and `OPoly2`, so we can pass it as parameter to both.

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 = Hashtbl.add base
    let put_bool k b = put k (`Bool b)
    let put_str k s = put k (`String s)
    let get = Hashtbl.find base
    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 of string]`. 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 without the "`private`" keyword one could not use it in a type declaration. 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 [18].

In order to extend this basic property type, we only need to define a new type and apply the functor. Here we show interactive definitions at the toplevel, including the types inferred by the compiler (in italic).

```
# 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 : string -> extended -> unit
    val put_bool : string -> bool -> unit
    val put_str : string -> string -> unit
    val get : string -> extended
    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
```

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

```
(* base.ml *)
type basic = [‘Bool of bool | ‘Int of int | ‘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 [10]. 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 ADTs, 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
```

You can see that `` `Zero ``, being public, 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 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, yet without abstraction it would result in unsoundness, as one could use it to mutate these contents.

Another approach would be to use only an abstract type for the array returned by `v`. However, one has to keep in mind that 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, eventhough 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`. There is unfortunately no support for inheritance.

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 had a small drawback: one had to close the recursion individually for each operation defined on the datatype. Moreover it relied 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 [7] 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. 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 [19], and our original solution which used only polymorphism.

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

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

Figure 1: Types and kinds

3 Formalization

We provide here a short description of the formal system underlying private row types. It is based on our formalism for structural polymorphism [8] for the core language part, combined with Leroy's description of an applicative functor calculus [9].

We will not give full details of these two systems, as both of them are rather complex, yet very few changes are actually needed. In order to introduce private row types, we only need two things. 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.

Actually, following the intuition that row types are just structural types with a row variable, the first aspect may seem trivial: why not simply use an abstract type for this row variable. This would mean that private row types should actually be two types: a structural type, with a manifest definition, and an associated abstract type, used as row variable for the manifest one. This is exactly what we will do, up to the point that there is actually no such thing as a single row variable, capturing all possible refinements of a type. More precisely, we could use one for object types, as one can only refine them by adding new methods, but this is impossible for variant types, as existing constructors may also disappear. For this reason, the formalism we use here is based on kinds [12] rather than row variables, but we will still use an abstract type representing a "virtual" row variable.

3.1 Core type system

We will directly use the formalism from [8], as it is already general enough. We only have to add parameterized abstract types. We repeat the basic definitions in the appendix.

The syntax for types and kinds 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. The only relation we use, $\mapsto$, 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. 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 will include both object and variant types, and support for type identity. We assume a set $\mathcal{L}$ of labels, denoting methods or variant constructors.

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

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 $\mathcal{L}$ (for objects, only $U = L$ is allowed), or $\mathcal{L}$ itself. p is 1 for private types, 0 for normal types. The kinds corresponding to the syntax used in previous sections are given in figure 2, respectively for object types, open variants, and closed variants. For both of objects and variants, we obtain a "final" (non-refinable) type by choosing $L = U$.

$$
\begin{array}{ll}
< m_1 : \tau_1; \ldots; m_n : \tau_n; .. > & \stackrel{\text{def}}{=} \quad \alpha :: (\mathsf{o}, \{m_1, \ldots, m_n\}, \mathcal{L}, 0, \{m_1 \mapsto \tau_1, \ldots, m_n \mapsto \tau_n\}) \triangleright \alpha \\
[> l_1 \text{ of } \tau_1 \mid \ldots \mid l_n \text{ of } \tau_n] & \stackrel{\text{def}}{=} \quad \alpha :: (\mathsf{v}, \{l_1, \ldots, l_n\}, \mathcal{L}, 0, \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n)\}) \triangleright \alpha \\
[< l_1 \text{ of } \tau_1 \mid \ldots \mid l_n \text{ of } \tau_n > l_1 \ldots l_k] & \stackrel{\text{def}}{=} \\
& \qquad \alpha :: (\mathsf{v}, \{l_1, \ldots, l_k\}, \{l_1, \ldots, l_n\}, 0, \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\}) \triangleright \alpha
\end{array}
$$

Figure 2: Kinds corresponding to surface syntax

We define an entailment relation on constraints, that is reflexive and transitive. We first distinguish inconsistent constraints.

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

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.

Then entailment can refine a constraint as long as it is not private. Note that refinement goes backward.

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

Next we must define predicates on our constraints, and use them in propagation rules. Our only predicate is *uniq*, denoting when only one type can be associated to a label. For a constraint $C = (k, L, U, p)$ and a label $l \in \mathcal{L} \cup \{row\}$:

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

In the original system without private rows, the definition was $uniq(l) \stackrel{\text{def}}{=} k = \mathsf{o} \vee l \in L$, 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, not only private types cannot have their constraint further refined, but all their possible labels must have unique types —this ensures that no typing information will be added to them. The special label *row* is used to encode our virtual row; it is always unique, and will be associated to an abstract type.

It is easy to see that these definitions satisfy the conditions for a valid constraint domain, given in appendix. This means that we have subject reduction (leading to type soundness) and principal type inference for a type system using them.

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, we will switch to Leroy's module calculus [9]. 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 K may contain variables outside of $\vec{\alpha}$, as long as their kinds are no longer refinable, *i.e.* either $L = U$ or $p = 1$. It would be clearly unsound to allow variables of refinable kinds to be free. "$E \vdash \sigma\ \texttt{type}$" checks that σ is a valid polytype under environment E, and that no refinable type variable is free.

We have two rules for manifest type definitions. The public case is identical to [9], up to our type definitions.

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

As we have explained before, we understand the declaration of a private row type t as defining an abstract type t_{row}, and using it inside a manifest type t. Both definitions have the same type parameters.

$$\frac{\begin{array}{c}E;\texttt{type}\ t_{rowi}(\vec{\alpha}) \vdash \forall\vec{\alpha}.\theta\ \texttt{type} \qquad t_i \notin BV(E) \\ E;\texttt{type}\ t_{rowi}(\vec{\alpha});\texttt{type}\ t_i(\vec{\alpha}) = \theta \vdash s : S\end{array}}{E \vdash (\texttt{type}\ t_i = \texttt{private}\ \theta_0; s) : (\texttt{type}\ t_{rowi}(\vec{\alpha});\texttt{type}\ t_i(\vec{\alpha}) = \theta; S)}$$

where
$$\theta = K, \beta :: (k, L, U, 1, R \cup \{row \mapsto t_{rowi}(\vec{\alpha}))\}) \triangleright \beta$$
$$\theta_0 = K, \beta :: (k, L, U, 0, R) \triangleright \beta \qquad L \neq U$$

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_{row} 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.

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

This rule says that, a row type definition (either private or not) is subsumed by 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, (4) if $row \mapsto \tau \in R$, then we know that the original definition is also private, and both row variables are equivalent under E (*i.e.* they are the same $t_{row}(\vec{\alpha})$.)

Another slight modification we need is to allow the introduction of abstract types in the supertype. This accounts for the case where the original type definition is public, and we make it private through subtyping, introducing a new t_{row}. While it seems sound to allow it for any abstract type, in the following rule we limit ourselves here to hidden abstract types, that the user cannot explicitly refer, to avoid changing the set of accessible identifiers bound by a module.

$$\frac{E \vdash M : \texttt{sig}\ D_1; \ldots; D_n\ \texttt{end} \qquad D_i \neq (\texttt{type}\ t_{row})\ (1 \leq i \leq n)}{E \vdash M : \texttt{sig}\ D_1; \ldots; D_k; \texttt{type}\ t_{row}; D_{k+1}; \ldots; D_n\ \texttt{end}}$$

Our last concern is about substitution. Some typing rules substitute manifest definitions for some paths. Our whole encoding is based on the assumption that a manifest type is defined along its abstract row variable, in the same module or signature. If we substitute another abstract row type for the original one, we are left with an incoherent signature. A way to tackle the problem at this level is to force the simultaneous substitution of enclosing kinds. That is, if we substitute t_{row} with t'_{row} inside a kind $(k, L, U, 1, R \cup \{row \mapsto t_{row}\})$ (this is the only place it may appear), then we have to substitute the whole kind with the one defined by the t' corresponding to t'_{row} (this t' exists by our invariant.) A more natural way to see it, is that there is no need to substitute t_{row} itself (it never appears alone), but when substituting t we should look for occurrences of t_{row} inside a kind. If the type system keeps abbreviations, like OCaml does, rather than just replacing them by their manifest type, there is actually nothing to do: no occurrence of t_{row} will be visible anyway.

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 [9], 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 [13].

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 $\texttt{type list}(+\alpha)$ 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, 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 [4, 16] or, in an hopefully close future, combinable signatures [13].

While we have already considered a large number of examples using private polymorphic variants types, the interaction of private row types with objects is still more speculative. A natural extension would be to have private class types. However they would be of limited use, as one cannot inherit from a class whose type is private. There seems to be much work to do yet in that direction.

References

[1] Boulmé, S., T. Hardin and R. Rioboo, *Polymorphic data types, objects, modules and functors: is it too much?*, RR 014, LIP6, Université Paris 6 (2000).

[2] Bracha, G., M. Odersky, D. Stoutamire and P. Wadler, *Making the future safe for the past: Adding genericity to the Java programming language*, in: *Proc. AMC Symposium on Object Oriented Programming, Systems, Languages and Applications*, 1998.

[3] Canning, P., W. Cook, W. Hill, W. Olthoff and J. C. Mitchell, *F-bounded polymorphism for object-oriented programming*, in: *Proc. ACM Symposium on Functional Programming and Computer Architectures*, 1989, pp. 273–280.

[4] Crary, K., R. Harper and S. Puri, *What is a recursive module?*, in: *Proc. ACM Conference on Programming Language Design and Implementation*, 1999, pp. 50–63.

[5] Fisher, K. and J. Reppy, *The design of a class mechanism for Moby*, in: *Proc. ACM Conference on Programming Language Design and Implementation*, 1999.

[6] Garrigue, J., *Programming with polymorphic variants*, in: *ML Workshop*, Baltimore, 1998.

[7] Garrigue, J., *Code reuse through polymorphic variants*, in: *Workshop on Foundations of Software Engineering*, number 25 in Lecture Notes in Software Science (2000), pp. 93–100, `http://wwwfun.kurims.kyoto-u.ac.jp/ ~garrigue/papers/fose2000.html`.

[8] Garrigue, J., *Simple type inference for structural polymorphism*, in: *The Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, 2002.

[9] Leroy, X., *Applicative functors and fully transparent higher-order modules*, in: *Proc. ACM Symposium on Principles of Programming Languages*, 1995, pp. 142–153.

[10] Leroy, X., D. Doligez, J. Garrigue, D. Rémy and J. Vouillon, "The Objective Caml system release 3.08, Documentation and user's manual," Projet Cristal, INRIA (2004).

[11] Odersky, M., V. Crémet, C. Röckl and M. Zenger, *A nominal theory of objects with dependent types*, in: *Proc. European Conference on Object-Oriented Programming*, 2003.

[12] Ohori, A., *A polymorphic record calculus and its compilation*, ACM Transactions on Programming Languages and Systems **17** (1995), pp. 844–895.

[13] Ramsey, N., K. Fisher and P. Govereau, *An expressive language of signatures*, in: *Proc. International Conference on Functional Programming*, 2005.

[14] Rémy, D. and J. Garrigue, *On the expression problem* (2004), `http://pauillac.inria.fr/~remy/work/expr/`.

[15] Rémy, D. and J. Vouillon, *Objective ML: An effective object-oriented extension to ML*, Theory and Practice of Object Systems **4** (1998), pp. 27–50.

[16] Russo, C. V., *Recursive structures for Standard ML*, in: *Proc. International Conference on Functional Programming*, 2001, pp. 50–61.

[17] Wadler, P., *The expression problem*, Java Genericity mailing list (1998), `http://www.cse.ohio-state.edu/~gb/cis888.07g/java-genericity/20`.

[18] Zenger, M. and M. Odersky, *Extensible algebraic datatypes with defaults*, in: *Proc. International Conference on Functional Programming*, 2001, pp. 241–252.

[19] Zenger, M. and M. Odersky, *Independently extensible solutions to the expression problem*, in: *Proc. Workshop on Foundations of Object-Oriented Languages*, 2005.

A Core type system

This appendix includes the main definitions from [8].

A.1 Constraint domain

A constraint domain describes a class of constraints, and how they interact with the type system.

Definition 1 *A constraint domain $\mathcal{C}$ is composed of the following items.*

1. *A theory $\mathcal{T}_{\mathcal{C}}$ with an entailment relation $\models$ satisfying the following properties*
   (a) *There is a constraint $\bot$, such that for any C we have $\bot \models C$.*
   (b) *A constraint C such that $C \models \bot$ is* invalid. *Validity is decidable.*
   (c) *Entailment is reflexive and transitive: $C \models C$; if $C \models C'$ and $C' \models C''$ then $C \models C''$.*
   (d) *For any two constraints C and C', there is a constraint $C \wedge C'$ such that $C \wedge C' \models C$, $C \wedge C' \models C'$, and for all C'' such that $C'' \models C$ and $C'' \models C'$, we have $C'' \models C \wedge C'$.*

$$
\begin{aligned}
\mathrm{FV}_{\mathrm{K}}(\forall\alpha_1 \ldots \alpha_n.\mathrm{K}' \triangleright \tau) &= \mathrm{FV}_{\mathrm{K},\mathrm{K}'}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\} \\
\mathrm{FV}_{\mathrm{K},\alpha::(C,R)}(\alpha) &= \{\alpha\} \cup \mathrm{FV}_{\mathrm{K}}(R) \\
\mathrm{FV}_{\mathrm{K}}(u) &= \emptyset \\
\mathrm{FV}_{\mathrm{K}}(\tau_1 \to \tau_2) &= \mathrm{FV}_{\mathrm{K}}(\tau_1) \cup \mathrm{FV}_{\mathrm{K}}(\tau_2)
\end{aligned}
$$

Figure 3: Free variables under K

2. *An observation relation* $\vdash$ *checking some atomic properties of a constraint:* $C \vdash p(a)$ *where* p *and* a *are respectively a predicate and a label for the domain. Observation should be compatible with entailment:*

   If $C \models C'$ *and* $C' \vdash p(a)$ *then* $C \vdash p(a)$.

3. *A set of relating predicates of the form* $a \mapsto_r \tau$*, which relate labels and types.*

4. *A set of propagation rules* $\mathcal{E}_{\mathcal{C}}$*, of the form*

$$\forall x.(x \mapsto_r \alpha_1 \wedge A \mapsto_r \alpha_2 \wedge p(x) \Rightarrow \alpha_1 = \alpha_2)$$

   where A *is either the same variable* x *or a label.*

A.2 Types and kinds

Types and kinds were defined in section 3.

Definition 2 *A kind* $k = (C, R)$ *is well formed if*

1. *the constraint* C *is satisfiable.*

2. *for each* $x \mapsto_r \alpha_1 \wedge A \mapsto_r \alpha_2 \wedge p(x) \Rightarrow \alpha_1 = \alpha_2$ *in* $\mathcal{E}$ *and each* $a \mapsto_r \tau_1$ *and* $[a/x]A \mapsto_r \tau_2$ *in* R *such that* $C \vdash p(a)$*, we have* $\tau_1 = \tau_2$.

We define kinding environments as containing only well formed kinds. Notice that all type variables do not necessarily have a kind, only those that represent constrained types do.

Free variables $\mathrm{FV}_{\mathrm{K}}(\sigma)$ of a polytype σ under a kinding environment K are defined as the minimum set satisfying the equations of figure 3.

Definition 3 *A type substitution* ς*, extended as usual on monotypes and polytypes, is* admissible *between the kinding environments* K *and* K′*, written* $\mathrm{K} \vdash \varsigma : \mathrm{K}'$*, if for all* $\alpha :: (C, R)$ *in* K, $\varsigma(\alpha)$ *is a type variable* α' *and it satisfies the following properties.*

1. $\alpha' :: (C', R') \in \mathrm{K}'$

2. $C' \models C$

3. $\varsigma(R) \subseteq R'$

Condition 1 ensures that all kinded variables are mapped to kinded variables. Condition 2 ensures that constraints are instantiated correctly (according to entailment). Condition 3 ensures that all type constraints are kept.

We will write $\mathrm{K}|_D$ for the restriction of the kinding environments K to variables in D, and $\mathrm{K}|_{\overline{D}}$ for its restriction to variables outside of D.

VARIABLE
$$\frac{\mathrm{K}, \mathrm{K}_0 \vdash \varsigma : \mathrm{K} \quad \mathrm{Dom}(\varsigma) \subset B}{\mathrm{K}; \Gamma, x : \forall B.K_0 \triangleright \tau \vdash x : \varsigma(\tau)}$$

ABSTRACTION
$$\frac{\mathrm{K}; \Gamma, x : \tau \vdash e : \tau'}{\mathrm{K}; \Gamma \vdash \mathsf{fun}\ x \to e : \tau \to \tau'}$$

APPLICATION
$$\frac{\mathrm{K}; \Gamma \vdash e_1 : \tau \to \tau' \quad \mathrm{K}; \Gamma \vdash e_2 : \tau}{\mathrm{K}; \Gamma \vdash e_1\ e_2 : \tau'}$$

GENERALIZE
$$\frac{\mathrm{K}; \Gamma \vdash e : \tau \quad B = \mathrm{FV}_{\mathrm{K}}(\tau) \setminus \mathrm{FV}_{\mathrm{K}}(\Gamma)}{\mathrm{K}|_{\overline{B}}; \Gamma \vdash e : \forall B.\mathrm{K}|_B \triangleright \tau}$$

LET
$$\frac{\mathrm{K}; \Gamma \vdash e_1 : \sigma \quad \mathrm{K}; \Gamma, x : \sigma \vdash e_2 : \tau}{\mathrm{K}; \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

CONSTANT
$$\frac{\mathrm{K}_0 \vdash \varsigma : \mathrm{K} \quad \mathrm{Tconst}(c) = \mathrm{K}_0 \triangleright \tau}{\mathrm{K}; \Gamma \vdash c : \varsigma(\tau)}$$

Figure 4: Typing rules

Objects

$$\begin{array}{lcl}
\mathsf{object}_{l_1 \ldots l_n} & : & \alpha :: (\mathsf{o}, \{l_1, ..., l_n\}, \{l_1, ..., l_n\}, 0, \{l_1 \mapsto \alpha_1, ..., l_n \mapsto \alpha_n\}) \\
& & \triangleright\ (\alpha \to \alpha_1) \to \ldots \to (\alpha \to \alpha_n) \to \alpha \\
\mathsf{call}_l & : & \alpha :: (\mathsf{o}, \{l\}, \mathcal{L}, 0, \{l \mapsto \beta\}) \triangleright \alpha \to \beta
\end{array}$$

Variants

$$\begin{array}{lcl}
\mathsf{tag}_l & : & \alpha :: (\mathsf{v}, \{l\}, \mathcal{L}, 0, \{l \mapsto \beta\}) \triangleright \beta \to \alpha \\
\mathsf{match}_{l_1 \ldots l_n} & : & \alpha :: (\mathsf{v}, \emptyset, \{l_1, ..., l_n\}, 0, \{l_1 \mapsto \alpha_1, ..., l_n \mapsto \alpha_n\}) \\
& & \triangleright\ (\alpha_1 \to \beta) \to \ldots \to (\alpha_n \to \beta) \to \alpha \to \beta
\end{array}$$

Figure 5: Constants for objects and variants

A.3 Terms and typing

Expressions are the standard ones,

$$\begin{array}{rcll}
e & ::= & x \mid \mathsf{fun}\ x \to e \mid e\ e & \text{core lambda} \\
& \mid & c \mid \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{constants and let}
\end{array}$$

Type judgments are extended with a kinding environment,

$$\mathrm{K}; \Gamma \vdash e : \tau$$

where K is a well-formed kinding environment and Γ is a set of bindings $x : \sigma$ from term variables to polytypes.

Typing rules appear in figure 4. They are in the syntax-directed style, instantiating and generalizing in one step, which simplifies the handling of eventual mutual recursion in the kinding environment. For information, we also include in figure 5 the types of the constants associated with our constraint domain.

Theorem 1 (Subject reduction) *If* $\mathrm{K}; \Gamma \vdash E[e] : \sigma$ *and* $e \to e'$ *by* BETA : $((\mathsf{fun}\ x \to e_1)\ e_2) \to [e_2/x]e_1$ *or* LET : $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \to [e_2/x]e_1$, *then* $\mathrm{K}; \Gamma \vdash E[e'] : \sigma$

A solution to a typing problem $\mathrm{K}; \Gamma \triangleright e : \tau$ is a substitution $\mathrm{K} \vdash \varsigma : \mathrm{K}'$ such that $\mathrm{K}'; \varsigma(\Gamma) \vdash e : \varsigma(\tau)$ is derivable. We have a type reconstruction algorithm satisfying the following theorem.

Theorem 2 (Principality) *If* $\mathrm{K}; \Gamma \triangleright e : \tau$ *can be reduced to* $\mathrm{K} \vdash \varsigma : \mathrm{K}'$ *by the type reconstruction algorithm,* $\mathrm{K}'; \varsigma(\Gamma) \vdash e : \varsigma(\tau)$ *is derivable, and* ς *is the most general solution; otherwise it reduces to* $\bot$ *and there is no solution.*