A proposal for recursive modules in Objective Caml

Xavier Leroy
INRIA Rocquencourt

Version 1.1, May 13, 2003

This notes describes a design and prototype implementation of an extension of the Objective Caml language with mutually-recursive module definitions.

1 Syntax

The syntax for recursive module definitions is as follows:

```
definition ::=
      ...
    | module rec ident_1 : module-type_1 = module-expr_1
      and ...
      and  ident_n : module-type_n = module-expr_n
```

A `module rec` definition can appear anywhere a regular module definition `module` *ident* `=` *module-expr* can appear: at the top-level of a compilation unit, within a `struct`...`end` structure definition, or as a phrase for the top-level interactive loop.

However, a recursive module definition must be contained whole within a compilation unit: the proposal does not support recursion between compilation units. The latter can however be encoded using separately-compiled functors, whose fix-point is taken later using the `module rec` construct.[1]

The scope of the identifiers $ident_1, \ldots, ident_n$ encompasses not only the module expressions $module\text{-}expr_i$, but also the module types $module\text{-}type_i$. Thus, not only the modules are recursive, but also their types.

The typing annotations $module\text{-}type_i$ are syntactically required. There is no way to type-check recursive module definitions if the expected module types are not provided by the user.

To allow declaring the types of recursively-defined modules in signatures and compilation unit interfaces, syntax is provided for recursive module declarations:

```
specification ::=
      ...
```

[1] Recursion between value components of implementations of compilation units (the `.ml` files) can be supported without language extension, by modifying the linker only. This is what Fabrice Le Fessant implemented a while ago in one of his patches, and this implementation is entirely orthogonal to what is described in this note. However, recursion between the interfaces of the compilation units (the `.mli` files), as required to split type definitions between several units, is significantly harder to achieve, and requires at the very least that all `.mli` files that refer to each other are compiled simultaneously.

```
| module rec ident_1 : module-type_1
  and ...
  and  ident_n : module-type_n
```

Example: we will use the following, often wished-for recursive module definition as our running example.

```
module A : sig
             type t = Leaf of string | Node of ASet.t
             val compare: t -> t -> int
           end
         = struct
             type t = Leaf of string | Node of ASet.t
             let compare t1 t2 =
               match (t1, t2) with
                 (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
               | (Leaf _, Node _) -> 1
               | (Node _, Leaf _) -> -1
               | (Node n1, Node n2) -> ASet.compare n1 n2
           end
and ASet : Set.S with type elt = A.t
         = Set.Make(A)
```

2 Type-checking

2.1 Checking recursive module types

For recursively-defined module types, checking their type-correctness and elaborating them to internal form is complicated by the very fact that they refer to each other in a recursive fashion. Consider the recursive module declaration

$$\texttt{module rec } X_1 : \textit{module-type}_1 \texttt{ and } \ldots \texttt{ and } X_n : \textit{module-type}_n$$

Checking is done in three steps:

1. Each module type expression $\textit{module-type}_i$ is syntactically approximated by a module type A_i that records only the names of type, module, and module type components, and declares all types as abstract, retaining only their arities. For example,

   ```
   module
     val x : int
     type 'a t = A | B of 'a
     class c : ...
     module M : sig type ('a,'b) t = 'a * 'b  val y : bool end
   end
   ```

   is syntactically approximated by

```
module
  type 'a t
  type c
  module M : sig type ('a, 'b) t end
end
```

2. The module type expressions *module-type*$_i$ are checked and translated to module types M_i under the typing assumptions $X_1 : A_1, \ldots, X_n : A_n$. The approximations, while imprecise, capture the valid type paths rooted at the X_j along with their arities, allowing the translation to module types to go through if the declaration is type-correct. Errors such as references to unbound type paths or type arity mismatches are caught at this point.

3. The module type expressions *module-type*$_i$ are checked again under the typing assumptions $X_1 : M_1, \ldots, X_n : M_n$. This re-checking is necessary e.g. to ensure that constraints on parameters of type constructors are satisfied.

The approximation phase (step 1) fails if it encounters one of the following situations:

- `include` *module-type* in a module signature, in case *module-type* refers to one of the recursively-defined module identifiers X_i;
- `inherit` *class-type* in a class body type, in case *class-type* refers to a class component of one of the recursively-defined module identifiers X_i.

Both situations correspond to cases where the existence of a fixpoint for the recursive definition of the module types is highly unclear.

Another failure case for the checking of recursive module types is ill-funded type definitions, as in `module A : sig type t = A.t end`. This is caught by a generalization of the cyclicity test in type definitions (the mechanism that prevents definitions such as `type t = t`).

2.2 Typing recursive modules

Consider now the recursive module definition

`module rec` $\ldots X_i$: *module-type*$_i$ $=$ *module-expr*$_i$ `and` $\ldots$

Typing this definition proceeds in three steps:

1. Check the module type expressions *module-type*$_i$ and translate them to module types M_i as described in section 2.1.

2. Type the module expressions *module-expr*$_i$ and infer their types N_i under the typing assumptions $X_1 : M_1, \ldots, X_n : M_n$.

3. Check that $N_i <: M_i$ under the typing assumptions $X_1 : M_1, \ldots, X_n : M_n$.

Datatype generativity complicates steps 2 and 3 to some extent. This is best explained on examples. For step 3, consider:

```
module rec A : sig type t = C  val x: A.t end
             = struct type t = C  let x = C end
```

The module type inferred for the right-hand side is $N =$ `sig type t = C  val x: t end`, and this is not a subtype of $M =$ `sig type t = C  val x: A.t end`, since the `t` type component of the first signature is treated as a new, freshly-generated type. To allow this definition to be accepted, it is not the inferred type N that is checked to be a subtype of the declared type M, but the result of strengthening N by the identifier `A`, i.e. of adding type equalities to reflect the fact that N is the type of a module bound to A. The result of the strengthening is $N' =$ `sig type t = A.t = C  val x: t end`, and N' is indeed a subtype of M.

A similar problem occurs during step 2 when the right-hand side is a structure defining datatypes. Consider:

```
module rec A : sig type t ... end
             = struct
                  type t = C
                  ... B.f C ...
               end
       and B : sig val f: A.t -> int end
             = ...
```

When `B.f C` is type-checked, `C` is known to be of type `t = C`, which is a freshly-generated type, distinct from `A.t`. Thus, the argument to `B.f` is not of the expected type `A.t`.

To address this issue, when the right-hand side of a recursive module definition is a structure, a strengthening of the datatype definitions is performed on the fly during the typing of the structure. In the example above, when the structure component `type t = C` is processed, the remainder of the structure is typed under the assumption `type t = A.t = C` rather than just `type t = C`. This allows the remainder of the structure to "know" that the types `t` and `A.t` are indeed the same. This incremental strengthening also applies to sub-modules, as shown by the following, more complex example.

```
module rec A
  : sig type t  module M : sig type u end end
  = struct
      type t = C
      (* here we enter type t = A.t = C in the environment *)
      type 'a u = D
      (* here we enter type 'a u = D since there is no A.u component
         in the declared signature *)
      module M = struct
        type u = E
        (* here we enter type u = A.M.u = E in the environment *)
        ...
      end
      (* here we enter module M : sig type u = A.M.u = E end *)
      ...
    end
```

3 Compilation and evaluation

The run-time representation of modules is composed of records (for structures) and functions (for functors). Hence, the evaluation of a recursive module definition entails taking fixpoints involving both functions and records.

The "in-place update" scheme The core Caml language already supports `let rec` definitions involving records and functions in the right-hand sides of the `let rec`, such as

```
let rec x = { a = function y -> x.b (y + 1);
              b = function y -> y * 2 }
```

These `let rec` definitions are evaluated using the "in-place update" trick: first, `x` is bound to a block of the same size as the result of the defining expression (here, 2), initialized with dummy values; then, the defining expression is evaluated; finally, the contents of the block resulting from the evaluation of the defining expression are copied in place to the dummy block, thus building the required cycles in the data representation.

However, this scheme is statically restricted to ensure that the evaluation of the defining expression does not destructure the recursively-defined identifier, but only uses the pointer representing it. Otherwise, the defining expression could go wrong by accessing and using the dummy values contained in the dummy block.

The syntactic restrictions that prevent this behavior are unfortunately too strong for recursive modules. In particular, they would disallow any functor application to one of the recursively-defined modules, which is something that we really want to do. In other terms, if we were to apply the `let rec` scheme to recursive modules, the only recursive module definitions that would be supported are those where the defining module expressions are literal `struct...end` structures where (to a first approximation) the only uses of the recursively-defined module identifiers are under syntactic functions. Again, this would rule out many interesting examples.

The "lazy evaluation" scheme At the other end of the expressivity spectrum, lazy evaluation (or similar schemes involving additional indirections and run-time tests) could be used to support arbitrary right-hand sides, with run-time failures in case of ill-founded definitions. For instance, the definition

```
module rec A : ... = struct ... A.f 3 ... end
```

would be compiled down to

```
let rec A = lazy { ... (Lazy.force A.f) 3 ... }
let A = Lazy.force A
```

One drawback of this approach is that the additional indirection and test entailed by lazy evaluation has a run-time cost. Another drawback is that functor applications must be manually eta-expanded:

```
module rec A : sig val f: t->t end = F(A)
```

would be compiled down to

```
let rec A = lazy(F(Lazy.force A))
let A = Lazy.force A
```

systematically causing a run-time error. The user would have to write

```
module rec A : sig val f: t->t end = F(struct let f x = A.f x end)
```

to avoid forcing `A` too early.

The "relaxed in-place update" scheme The evaluation scheme that I propose for recursive modules is a variant of the "in-place update" scheme where we trade the possibility of run-time failures (in case of ill-founded definitions) for greater expressiveness. Namely, we allow the defining module expressions to use recursively-defined module identifiers arbitrarily provided that these identifiers can be bound to a dummy block containing safe values for the expected types, e.g.

```
function x -> raise Undefined_recursive_module
```

for a field of functional type. Ill-founded recursion thus leads to this safe value being used; evaluation cannot go wrong, but may raise the `Undefined_recursive_module` exception.

Unlike the lazy evaluation scheme, this variant of the "in-place update" scheme entails no additional indirections and no additional run-time tests. Thus, run-time efficiency is preserved.

More precisely, we say that a module is *safe* if it is a structure, and all the value components it contains have a function type or a `Lazy.t` type, and all the module components it contains are themselves safe.[2] All other modules are unsafe. Note that the safe/unsafe determination is done by looking only at the type of the module, not at its actual definition. In the `A`/`ASet` example of section 1, the module `A` is safe because its only value component, `A.compare`, is of functional type; `ASet` is unsafe because it contains e.g. `ASet.empty`, which is of an abstract type `ASet.t`.

Consider the recursive module definition

$$\texttt{module rec } X_1 : M_1 = m_1 \texttt{ and } \ldots \texttt{ and } X_n : M_n = m_n$$

After classifying $X_1, \ldots, X_n$ as safe or unsafe based on their types $M_1, \ldots, M_n$, the compiler reorders the bindings so that the following criterion holds:

> For each i, there does not exist $j \geq i$ such that X_j is unsafe and occurs free in m_i.

If no such reordering exists, the definition is statically rejected. This happens when we have dependency cycles that do not cross at least one safe definition, for instance: X_1 is free in m_2, X_2 is free in m_1, and both X_1 and X_2 are unsafe.

In general, several valid reorderings exist. The following heuristics are applied to choose one:

1. Retain the original ordering of definitions as much as possible;
2. Evaluate a right-hand side m_i after all the recursively-defined identifiers X_j that are free in m_i have been evaluated.

[2] Notice that a functor is always unsafe. The reason for this is that the size of the closure representing the functor at run-time cannot be determined from its type, thus preventing the construction of a suitable initial value in phase 1 of the compilation scheme described later.

If a suitable ordering was found, we emit the following code for the recursive module definition.

1. All safe X_i are bound to an initial value derived from the type M_i. Value fields of functional type are initialized to `function x -> raise(Undefined_recursive_module` *loc*`)`, where *loc* is the source location of the recursive module. Value fields of lazy type are initialized to `lazy (raise Undefined_recursive_module(`*loc*`)`. Exception fields are initialized to a suitable exception identifier. Class fields are initialized with an empty method table and class initializer and object construction functions that just raise the `Undefined_recursive_module` exception. Sub-modules are recursively set to correct initial values for their types.

2. Then, the definitions $X_i = m_i$ are processed in sequence from $i = 1$ to $i = n$. For each definition, the defining module expression m_i is computed normally. If X_i is unsafe, the value of m_i is `let`-bound to X_i. If X_i is safe, the contents of the initial value X_i are overwritten (in place) by the contents of the value of m_i.

Owing to the reordering criterion described above, the evaluation of m_i cannot refer to a not-yet-bound recursively-defined identifier X_j: either $j < i$ and X_j was already evaluated to its final value, or $j \geq i$ and X_j is safe, thus bound to an initial value of the correct type.

Example Continuing the `A`/`ASet` example of section 1, we have that `A` is safe (because its only value component, `compare`, has a function type) and `ASet` is unsafe (because it contains non-functional value components). Thus, the reordering phase decides to evaluate `ASet` first, then `A`. The generated code is

```
(* phase 1 *)
let A = { compare = fun x -> raise(Undefined_recursive_module loc) } in
(* phase 2 *)
let ASet = Set.Make(A) in
update(A, { compare = fun x y -> match (x,y) with ... });
...
```

The `Undefined_recursive_module` exception is not raised provided the `Set.Make` functor does not use immediately the field `A.compare` of its argument, which is the case.

4 Extended examples

Putting it all together, here are some more examples illustrating the various aspects of the design.

The expression/binding example This example comes from the paper "What is a recursive module?" by Crary, Harper and Puri. It type-checks and evaluates as expected. Type-checking requires the incremental type-strengthening trick described in section 2.2.

```
module rec Expr
  : sig
      type t =
        Var of string
```

```
      | Const of int
      | Add of t * t
      | Binding of Binding.t * t
      val make_let: string -> t -> t -> t
      val simpl: t -> t
    end
  = struct
      type t =
        Var of string
      | Const of int
      | Add of t * t
      | Binding of Binding.t * t
      let make_let id e1 e2 = Binding([id, e1], e2)
      let rec simpl = function
        Var s -> Var s
      | Const n -> Const n
      | Add(Const i, Const j) -> Const (i+j)
      | Add(Const 0, t) -> simpl t
      | Add(t, Const 0) -> simpl t
      | Add(t1,t2) -> Add(simpl t1, simpl t2)
      | Binding(b, t) -> Binding(Binding.simpl b, simpl t)
    end

and Binding
  : sig
      type t = (string * Expr.t) list
      val simpl: t -> t
    end
  = struct
      type t = (string * Expr.t) list
      let simpl b =
        List.map (fun (id,e) -> (id, Expr.simpl e)) b
    end
```

Okasaki's bootstrapped heaps The following example comes from Okasaki's book *Purely functional data structures*. It shows a higher-order functor that takes a fixpoint of its functorial parameter. In addition, it seems genuinely useful.

```
module type ORDERED =
  sig
    type t
    val eq: t -> t -> bool
    val lt: t -> t -> bool
    val leq: t -> t -> bool
  end
```

```
module type HEAP =
  sig
    module Elem: ORDERED
    type heap
    val empty: heap
    val isEmpty: heap -> bool
    val insert: Elem.t -> heap -> heap
    val merge: heap -> heap -> heap
    val findMin: heap -> Elem.t
    val deleteMin: heap -> heap
  end

module Bootstrap (MakeH: functor (Element:ORDERED) ->
                                    HEAP with module Elem = Element)
                 (Element: ORDERED) : HEAP with module Elem = Element =
  struct
    module Elem = Element
    module rec BE
      : sig type t = E | H of Elem.t * PrimH.heap
            val eq: t -> t -> bool
            val lt: t -> t -> bool
            val leq: t -> t -> bool
        end
      = struct
          type t = E | H of Elem.t * PrimH.heap
          let leq (H(x, _)) (H(y, _)) = Elem.leq x y
          let eq (H(x, _)) (H(y, _)) = Elem.eq x y
          let lt (H(x, _)) (H(y, _)) = Elem.lt x y
        end
    and PrimH
      : HEAP with type Elem.t = BE.t
      = MakeH(BE)
    type heap = BE.t
    let empty = BE.E
    let isEmpty = function BE.E -> true | _ -> false
    let rec merge x y =
      match (x,y) with
        (BE.E, _) -> y
      | (_, BE.E) -> x
      | (BE.H(e1,p1) as h1), (BE.H(e2,p2) as h2) ->
          if Elem.leq e1 e2
          then BE.H(e1, PrimH.insert h2 p1)
          else BE.H(e2, PrimH.insert h1 p2)
    let insert x h =
```

```
      merge (BE.H(x, PrimH.empty)) h
    let findMin = function
        BE.E -> raise Not_found
      | BE.H(x, _) -> x
    let deleteMin = function
        BE.E -> raise Not_found
      | BE.H(x, p) ->
          if PrimH.isEmpty p then BE.E else begin
            let (BE.H(y, p1)) = PrimH.findMin p in
            let p2 = PrimH.deleteMin p in
            BE.H(y, PrimH.merge p1 p2)
          end
  end
```

Polymorphic recursion Recursive modules support polymorphic recursion: polymorphic functions that call themselves at a different type than the one they were invoked with.

```
module rec PolyRec
  : sig
      type 'a t = Leaf of 'a | Node of 'a list t * 'a list t
      val depth: 'a t -> int
    end
  = struct
      type 'a t = Leaf of 'a | Node of 'a list t * 'a list t
      let x = (PolyRec.Leaf 1 : int t)
      let depth = function
        Leaf x -> 0
      | Node(l,r) -> 1 + max (PolyRec.depth l) (PolyRec.depth r)
    end
```

Litmus tests Here are some "litmus tests" for the static or dynamic detection of ill-founded recursion. The following example is rejected statically:

```
module rec A : sig val x : int end = struct let x = B.x + 1 end
       and B : sig val x : int end = struct let x = A.x * 2 end
```

Indeed, both modules `A` and `B` are unsafe, yet depend on each other.

The following variant is accepted because it contains no such cyclic unsafe dependency:

```
module rec A : sig val x : int end = struct let x = B.x + 1 end
       and B : sig val x : int end = struct let x = 2 end
```

and causes `B` to be evaluated first, followed by `A`.

The following example is accepted statically, but raises a run-time exception at run-time (more precisely, at module initialization time):

```
module rec Bad : sig val f : int -> int end
               = struct let f = let y = Bad.f 5 in fun x -> x+y end
```