First-class modules and composable signatures in Objective Caml 3.12

Jacques Garrigue (Nagoya University)

http://www.math.nagoya-u.ac.jp/~garrigue/

with Alain Frisch (Lexifi), OCaml developper team (INRIA)

The role of modules in Objective Caml

- Structure programs and namespace
- Program specification and reuse through interfaces
- Encapsulation using abstract and private types
- Abstraction with functors

They affect all aspects of programming through many interactions

Advantages of modules over core

- Ability to pass polymorphic values around

  Already possible with records and objects, but modules also allow abstraction over parameterized type constructors.

- Exact type annotations (example: `val id : 'a -> 'a`)

  Since 3.12 they are also available in the core language.

- All forms of definitions are possible

  In the core language some definitions are not possible inside expressions, and some forms of mutual recursion are not available (e.g. class types with normal types)

- Existential types (abstract types)

  By using first-class polymorphism, one can encode existential types in the core language, but not existential type constructors.

New features in Objective Caml 3.12

- Local open
- Local abstract types
- Explicit polymorphism annotations
- First-class modules
- Signature of a module
- Destructive substitution

First-class modules

- Modules can be passed around like other values

  Definitions only possible at the module level become available in the core language.

- Fully typed

  Type safety is preserved.
  In contrast, Alice ML packages have dynamic types.

- The theory is not new

  Already implemented by Claudio Russo in Moscow ML 10 years ago.

Example 1: Wrapping polymorphic values

```
# module type ID = sig val id : 'a -> 'a end ;;
module type ID = sig val id : 'a -> 'a end
# let f id =
    let module Id = (val id : ID) in
    (Id.id 1, Id.id true) ;;
val f : (module ID) -> int * bool = <fun>
# f (module struct let id x = print_endline "Id!"; x end : ID);;
Id!
Id!
- : int * bool = (1, true)
```

Note: already possible with objects and records, but this feels more natural.

Example 2: Runtime module choice

```
module type DEVICE = sig ... end                    (* specification *)
let devices : (string, (module DEVICE)) Hashtbl.t
            = Hashtbl.create 17

module PDF = struct ... end                          (* instance *)
let () = Hashtbl.add devices "PDF" (module PDF: DEVICE)
...

module Device =                              (* use like a normal module *)
  (val (try Hashtbl.find devices Sys.argv.(1)
        with Not_found -> prerr_endline "Unknown device"; exit 2)
       : DEVICE)
```

Example 3: Type-safe plugins

```
module type PLUGIN = sig
  type t                                  (* each plugin has its state *)
  val init : t
  val start : t -> unit
  val stop : unit -> t
end ;;
let plugins = ref ([] : (string * (module PLUGIN)) list) ;;

let new_instance name =
  let module P = (val List.assoc name !plugins : PLUGIN) in
  object                                          (* P.t is fresh here *)
    val mutable state = P.init
    method start = P.start state
    method stop  = state <- P.stop ()
  end ;;
val new_instance : string -> < start : unit; stop : unit > = <fun>
```

Example 4: Dynamic class inheritance

```
module type Compute = sig
  class compute : object method x : int end
end
module Default = struct                (* put the class in a module *)
  class compute = object method x = 0 end
end
let compute = ref (module Default : Compute)

let incr () =          (* change the class by dynamic inheritance *)
  let module M = struct
    module C = (val !compute : Compute)
    class compute = object
      inherit C.compute as super
      method x = super#x + 1
    end
  end in compute := (module M : Compute)
```

Example 5: Parametric algorithms

```
module type Number = sig                              (* Number "class" *)
  type t  val int : int -> t  val (+) : t -> t -> t ...
end
module Numfloat = struct ... end                        (* instance *)

let average (type t) number arr =          (* t is locally abstract *)
  let module N = (val number : Number with type t = t) in
  let open N in                                        (* local open *)
  let r = ref (int 0)  and len = Array.length arr in
  for i = 0 to Pervasives.(len - 1)                    (* local open *)
            do r := !r + arr.(i) done;
  !r / int (Array.length arr)
val average : (module Number with type t = 'a) -> 'a array -> 'a
                          (* number has a polymorphic package type *)

average (module Numfloat:Number with type t = float) [|2.; 8.; 7.|]
- : float = 5.6666666666666696
```

Example 6: Singleton types and GADTs

```
module TypEq : sig
  type ('a, 'b) t                                    (* Type equation *)
  val apply : ('a, 'b) t -> 'a -> 'b
  val refl : ('a, 'a) t        val sym : ...
end = ...
module rec Typ : sig      (* typ and pair are mutually recursive *)
  module type PAIR =
    type t and t1 and t2
    val eq: (t, t1 * t2) TypEq.t
    val t1: t1 Typ.typ        val t2: t2 Typ.typ
  end
  type 'a typ =
    | Int of ('a, int) TypEq.t
    | String of ('a, string) TypEq.t
    | Pair of (module PAIR with type t = 'a)      (* ∃t1 ∃t2 ... *)
end = Typ
```

Example 6: Singleton types and GADTs

```
...                    (* exact annotation for polymorphic recursion *)
let rec to_string : 'a. 'a typ -> 'a -> string =
  fun (type s) t x ->                        (* s is locally abstract *)
    match t with
    | Int eq -> string_of_int (TypEq.apply eq x)
    | String eq -> Printf.sprintf "%S" (TypEq.apply eq x)
    | Pair p ->
        let module P = (val p : PAIR with type t = s) in
        let (x1, x2) = TypEq.apply P.eq x in
        Printf.sprintf "(%s,%s)"
          (to_string P.t1 x1) (to_string P.t2 x2)
```

Details in the manual; next talk gives another approach.

Package types: `(module S with ...)`

- Module types used for first-class modules must be defined in advance
- Type equality is nominal, but with constraints introduce parameters (of kind type)
- They may not contain type variables at packing and unpacking, but polymorphism may be introduced through locally abstract types
- Problem: polymorphic equality of first-class modules may break abstraction

Restrictions of first-class modules

- One cannot unpack inside the body of a functor

  OCaml's functors are applicative, so one would be able to write unsafe programs.

```
module type S = sig type t val x : t end
let r = ref (module struct type t = int let x = 0 end : S)
module F(X:sig end) = (val !r : S)
module A = struct end
module M = F(A) ;;
module M : sig type t = F(A).t val x : t end
r :=  (module struct type t = float let x = 0. end : S) ;;
module N = F(A) ;;
module N : sig type t = F(A).t val x : t end                (* same t! *)
```

  However, using modules in functions already provides a kind of (weak) generative functor.

- All operations require explicit types see next page

Type inference for first-class modules

Using the same mechanism as for first-class polymorphism, one can omit most annotations while keeping principality. (not in 3.12)

```
module type ID = sig val id : 'a -> 'a end ;;
let f (module Id:ID) = (Id.id 1, Id.id true);;(* unpack pattern *)
f (module struct let id x = x end);;      (* pack w/o annotation *)

let rec to_string : 'a. 'a typ -> 'a -> string =
  fun (type s) (t : s typ) x ->                (* type propagation *)
    match t with
    | Int eq -> string_of_int (TypEq.apply eq x)
    | String eq -> Printf.sprintf "%S" (TypEq.apply eq x)
    | Pair (module P) ->                     (* unpack w/o annotation *)
        let (x1, x2) = TypEq.apply P.eq x in
        Printf.sprintf "(%s,%s)"
          (to_string P.t1 x1) (to_string P.t2 x2) ;;
```

Objective Caml's Signature Language

- When one uses lots of modules, signatures also need to be scalable.
- Until now, OCaml signatures were rather weak.
  - No way to obtain the signature of a library module
  - Problems when composing signatures
- Ramsey et al. [2005] proposed a solution, but it was too large for addition.

Convert between module and signature

- Convert a signature to a module

  If it contains only types, we can use a recursive module.

```
module rec M : S = M
```

- Obtain the signature of module

  The new construct `module type of` does that.

```
module MyList : sig
  include module type of List
  val remove : 'a -> 'a list -> 'a list
end = struct
  include List
  let rec remove a = ...
end
```

Signature composition

We want to compose two signatures specifying operations on the same type.

```
module type Printable =
  sig type t  val print : t -> unit end
module type Comparable =
  sig type t  val compare : t -> t -> int end

module type PrintableComparable = sig          (* desired result *)
  type t
  val print : t -> unit
  val compare : t -> t -> int
end
```

Using `include` is not enough

If we attempt to `include` both signatures, we get an error about multiple definitions of `t`.

```
module type PrintableComparable = sig
  include Printable
  include Comparable with type t = t
end
Error: Multiple definition of the type name t.
       Names must be unique in a given structure or signature.
```

Functor-based solution

This can be solved by using functors, but at the cost of extra complexity.

```
module type T = sig type t end
module PrintableF(X:T) = struct
  module type S = sig val print : t -> unit end
end
module ComparableF(X:T) = struct
  module type S = sig val comparable : t -> t -> int end
end
module PrintableComparableF(X:T) = struct
  module type S =
    sig include PrintableF(X).S include ComparableF(X).S end
end
```

Destructive substitution in signatures

We extend the syntax of `with` constraints, so that on can remove a type or module component after substituting it.

```
module type ComparableInt = Comparable with type t := int ;;
module type ComparableInt =
  sig val compare : int -> int -> int end
```

This syntax has actually the same effect as `ComparableF(Int).S`, but originally one couldn't generate `Comparable` from `ComparableF`.

```
  module ComparableF(X:T) = struct
    module type S = Comparable with type t := X.t
  end
```

Applications of destructive substitution

- Signature composition: erase the common type after substitution

```
module type PrintableComparable = sig
  include Printable
  include Comparable with type t := t
end
```

- Removing a type or module: just repeat the definition

```
module type PrintableInt' =
  (Printable with type t = int) with type t := int
```

- Rename a type or module

```
module type Printable' = sig
  type printable
  include Printable with type t := printable
end
```

Limitations of destructive substitution

For implementation reasons, there are some restrictions.

- Only components at the top of a signature can be substituted.
- Only "paths" can replace them.
- The number and order of type parameters cannot be changed.
- It may be impossible to display the result of a substitution correctly. But this has no impact on the semantics.

Conclusion

- Modules and their integration with the core language were improved in Objective Caml 3.12.
- Thanks to that, many new and/or cleaner programming patterns become available.
  - plugins
  - GADTs
  - signatures composition
- The changes are small, but the whole language benefits from them.