# Objective Caml module system

Francesco Zappa Nardelli

Moscova Project — INRIA Rocquencourt

...based on a talk by Xavier Leroy.

BISS 2005, Bertinoro, Italy — March 14-18, 2005

# Modular programming

- Separation of a program into parts that can be compiled independently

  *Being able to compile big programs*

- Add structure to the code, simplyfing code reutilisation

  *Being able to understand big programs*

# ML modules

(MacQueen, Milner, Harper) — Largely independent from the base language.

- A little typed functional language.
- Deals with collections of definitions (of values, of types) of the base language.
- Their types: collections of declarations/specifications (types of values, type declarations).
- Nested modules (like data-structures).
- Functions (functors) and function application.

# Basic modules

The *structures*: collections of definitions.

`struct` *definition* `...` *definition* `end`

The definitions correspond to the base language constructs:

| | |
|---|---|
| definitions of values and functions | `let` $x = \ldots$ |
| definitions of types | `type` $t = \ldots$ |
| definitions of exceptions | `exception` $E$ [ `of` ...] |
| definitions of classes | `class` $C = \ldots$ |
| definitions of sub-modules | `module` $X = \ldots$ |
| definitions of types of modules | `module type` $S = \ldots$ |

# Naming of a module

We name modules using the `module` binder.

```
module S = struct ... end
```

Example:

```
module S =
  struct
    type t = int
    let x = 0
    let f x = x+1
  end
```

# Use of a module

Dotted notation to reference module fields: $module.field$

```
... (S.f S.x : S.t) ...
```

Other option: the `open` *module* directive allows omitting the prefix and the dot:

```
open S
... (f x : t) ...
```

## Nested modules

A module can be part of another module:

```
module T =
  struct
    module R = struct let x = 0 end
    let y = R.x + 1
  end
```

The dot notation and the `open` construct extend smoothly:

```
T.R.x

open T.R
```

## The types of the basic modules

The *signatures*: collections of specifications (of types).

`sig` *specification* `...` *specification* `end`

| | |
|---|---|
| specification of values | `val` $x : \sigma$ |
| specification of types, abstract | `type` $t$ |
| specification of types, manifest | `type` $t = \tau$ |
| specification of exceptions | `exception` $E$ |
| specification of classes | `class` $C : \ldots$ |
| specification of sub-modules | `module` $X : M$ |
| specification de module types | `module type` $S\ [\ = M]$ |

Naming of a module type: `module type SIG = sig ... end`

## Sub-signaturing

For every structure definition, the system infers the most precise signature: all the components of the structure are visible, with their most general type.

```
struct                              sig
    type t = int                        type t = int
    let f x = x                         val f : 'a -> 'a
end                                 end
```

To hide some fields, or to restrict their type, we use signature constraints: $(\mathit{structure} : \mathit{signature})$.

The constraint checks that the structure satisfies the signature, and reduces into the structure seen with the restricted signature.

# Example of restriction

```
module S =
  (struct
     type t = int
     let x = 1
     let y = x + 1
   end :
   sig
     type t
     val y : t
   end)
```

`S.x;;` *error "S.x is unbound"*
`S.y + 1;;` *error "S.y is not of type int"*

# Multiple views of the same module

```
module Stamp =
  struct
    type t = int
    let new () = ...
    let equal s1 s2 = (s1 = s2)
  end

module type ABSTRSTAMP =
  sig
    type t
    val equal: t -> t -> bool
  end

module AbstrStamp = (Stamp : ABSTRSTAMP)
```

# Modules and separate compilation

A compilation unit $A$ is composed of two files:

- the implementation file $A$`.ml`:
  a sequence of definitions
  like the content of `struct...end`

- the interface file $A$`.mli` (optional):
  a sequence of specifications
  like the content of `sig...end`

Another compilation unit $B$ can refer to $A$ as if $A$ is a structure, using the dot notation $A$`.x` or the `open` $A$ construct.

# Separate compilation of a program

Source files: `a.ml`, `a.mli`, `b.ml`

Compilation steps:

```
ocamlc -c a.mli                 (compiles the interface of A, creates a.cmi)
ocamlc -c a.ml                  (compiles the implementation of A, creates a.cmo)
ocamlc -c b.ml                  (compiles theimplementation of B, creates b.cmo)
ocamlc -o myprog a.cmo b.cmo  (final linking)
```

The program behaves like the monolithic code below:

```
module A = (struct content of a.ml end : sig content of a.mli end)
module B = struct content of b.ml end
```

The order of the module definitions corresponds to the order of the object files `.cmo` on the command line of the linker.

# Parametric modules

A *functor* is a function from modules into modules:

$$\texttt{functor}(S\texttt{:}\ \ signature)\ \texttt{->}\ module$$

The $module$ (body of the functor) is explicitely parametrised on the module $S$. It refers to the components of its parameter using the dot notation.

```
module T = functor(S : SIG) ->
  struct
    type u = S.t * S.t
    let y = S.g(S.x)
  end
```

# Functor application

Fields of `T` cannot be directly accessed: `T` must be applied to an implementation of the signature `SIG` (like a standard function application).

```
module T1 = T(S1)
module T2 = T(S2)
```

`T1`, `T2` are used like normal structures:

```
(T1.y : T2.u)
```

`T1` and `T2` share completely their code.

# Long example: polynomials over an arbitrary ring

We will implement the standard operations on polynomials with coefficients over an arbitrary ring, passed as a parameter.

The parameter "ring" is a structure respecting the signature below:

```
module type RING =
  sig
    type t              (* type of the elements of the ring *)
    val zero: t
    val one: t
    val plus: t->t->t   (* sum *)
    val opp: t->t       (* opposite *)
    val prod: t->t->t   (* product *)
  end
```

# Polynomials (implementation 1)

```
module PolyFull = functor (Ring : RING) ->
  struct
    module A = Ring
    type t = A.t array                  (* type of polynomials *)
    let zero = [| A.zero |]             (* polynomial 0 *)
    let one = [| A.one |]               (* polynomial 1 *)
    let monome c n =                    (* monome c.X^n *)
      let p = Array.create (n+1) A.zero in
      p.(n) <- c; p
    let coeff n p = p.(n)               (* coefficient of X^n in p *)
    let degree p =                      (* degree of p *)
      let d = ref (Array.length p - 1) in
      while !d >= 0 && p.(!d) = A.zero do decr d done;
      !d
```

```
  let plus p1 p2 =                        (* sum *)
    let r = Array.create (max (Array.length p1) (Array.length p2)) A.zero in
    for i = 0 to Array.length p1 - 1 do r.(i) <- p1.(i) done;
    for i = 0 to Array.length p2 - 1 do r.(i) <- A.plus r.(i) p2.(i) done;
    r
  let opp p = ...                         (* opposite *)
  let prod p1 p2 = ...                    (* product *)
end
```

## Use of the functor `PolyFull`

Polynomials with integer coefficients:

```
module Integers =
  struct
    type t = int
    let zero = 0
    let one = 1
    let plus x y = x + y
    let opp x = -x
    let prod x y = x * y
  end
module PolyIntegers = PolyFull(Integers)
```

Polynomials with integer coefficients with two variables = polynomials of polynomials of integers:

```
module Poly2Integers = PolyFull(PolyIntegers)
```

This is correct, because the structure obtained from `PolyFull` satisfies the signature `RING`.

# Polynomials (implementation 2)

```
module PolyEmpty = functor (Ring : RING) ->
  struct
    module A = Ring
    type t = (A.t * int) list          (* type of polynomials *)
    let zero = []                      (* polynomial 0 *)
    let one = [(A.one, 0)]             (* polynomial 1 *)
    let monome c n = [(c, n)]          (* monome c.X^n *)
    let rec degree = function
        [] -> 0
      | [(coeff, d)] -> d
      | (coeff, d) :: reste -> degre reste
    let rec coeff n p =
      match p with
        [] -> A.zero
```

```
    | (c, d) :: reste ->
        if d = n then c else
        if d > n then A.zero else coeff n reste
  let plus p1 p2 =  ...                       (* addition *)
  let opp p = ...                             (* opposite *)
  let prod p1 p2 = ...                        (* product *)
end
```

# Representation independence

`PolyFull` and `PolyEmpty` implements the same interface and are mostly interchangeable:

```
module PolyIntegers = PolyEmpty(Integers)
module Poly2Integers = PolyEmpty(PolyIntegers)
```

but not completely interchangeable: the representation type of the polynomials is visible!

```
module P1 = PolyFull(Integers);;          (* P1.t = int array *)
P1.degree [| 1;2;3;4 |];;                  (* - : int = 3 *)

module P2 = PolyEmpty(Integers);;          (* P2.t = (int * int) list *)
P2.degree [| 1;2;3;4 |];;                  (* type error *)
```

# Hiding the representation of the polynomials

We apply a constraining signature to the result of the functors `PolyFull` and `PolyEmpty`, to hide the type `t`.

First attempt:

```
module type POLYNOME =
  sig
    module A : RING
    type t
    val zero: t                          val one: t
    val monome: A.t -> int -> t          val degree: t -> int
    val coeff: int -> t -> A.t           val opp: t -> t
    val plus: t -> t -> t                val prod: t -> t -> t
  end
module PolyFull = functor (Ring: RING) -> (struct ... end : POLYNOME)
```

Problem: we hide the type `A.t` of the coefficients, and the result of `PolyFull` cannot be used:

```
module PolyIntegers = PolyFull(Integers);;
PolyIntegers.monome 15 2;;
```

*This expression has type int but is used with type PolyIntegers.A.t*

The right solution: rely on a manifest type to keep the identity of the type `A.t` of the result:

```
module PolyFull = functor (Ring: RING) ->
  (struct ... end : POLYNOME with type A.t = Ring.t)
```

Then:

```
module PolyIntegers = PolyFull(Integers)
```

We have `PolyIntegers.t` abstract,
but `PolyIntegers.A.t` = `Integers.t` = `int` as desired.

## The `with` notation

Allows adding type equalities to an existing signature. The expression `POLYNOME with type A.t = Ring.t` is a shorthand for the signature:

```
sig
  module A : sig type t = Ring.t
                 val zero: t         val one: t
                 val plus: t->t->t   val opp: t->t
                 val prod: t->t->t
             end
  type t
  val zero: t                      val one: t
  val monome: A.t -> int -> t   val plus: t -> t -> t
  val opp: t -> t                  val prod: t -> t -> t
  val degree: t -> int              val coeff: int -> t -> A.t
end
```

# Modules vs. polymorphism

You can implement polynomials without functors, using ML polymorphism:

```
type 'a polynome
type 'a ring =
  { zero: 'a; one: 'a; plus: 'a->'a->'a; opp: 'a->'a; prod: 'a->'a->'a }
val zero: 'a ring -> 'a polynome
val one: 'a ring -> 'a polynome
val monome: 'a ring -> 'a -> int -> 'a polynome
val plus: 'a ring -> 'a polynome -> 'a polynome -> 'a polynome
val opp: 'a ring -> 'a polynome -> 'a polynome
val prod: 'a ring -> 'a polynome -> 'a polynome -> 'a polynome
val degree: 'a ring -> 'a polynome -> int
val coeff: 'a ring -> int -> 'a polynome -> 'a
```

First problem: ring operations are one auxiliary argument of the operations on polynomials.

*Functors = global parametrisation*
*Base language = local parametrisation*

Second problem: the type of polynomials reflects only the type of the elements of the ring, not the ring.

```
let entiers =
  { zero = 0; one = 1; plus = fun x y -> x + y; ...}
let z_sur_3z =
  { zero = 0; one = 1; plus = fun x y -> (x + y) mod 3; ...}
let p1 = monome entiers 15 2
let p2 = monome z_sur_3z 2 3
let p3 = prod entiers p1 p2
```

Mixing `p1` and `p2` is absurd, but they have the same type `int polynome`.

The implementation using functors detects statically this error thanks to the the type system:

```
module Z_sur_3Z =
  struct type t = int  let zero = 0  let one = 1  let plus x y = (x+y) mod 3 ... end
module PolyIntegers = PolyFull(Integers)
module PolyZ3Z = PolyFull(Z_sur_3Z)
```

The types `PolyIntegers.t` and `PolyZ3Z.t` are both abstracts, and as such incompatibles:

```
PolyZ3Z.degree PolyIntegers.one
```

> *This expression has type PolyIntegers.t*
> *but is here used with type PolyZ3Z.t*

## Special case: $\mathbf{Z}/2\mathbf{Z}$

Other advantage of functors: give direct implementations for some types.

Example: to implement $\mathbf{Z}/2\mathbf{Z}$ we give a structure `Z2Z` based on bit arrays, with the same interface that the result of the `PolyFull` functor:

```
module Poly_Z2Z : (POLYNOME with type A.t = int) =
  struct
    module A = Z_over_2Z
    type t = bitvect
    let plus = bitvect_xor
    let prod = bitvect_prod
    let opp = id
    ...
  end
```