The Module Language

Part II – Defining Functors, First class modules

OCaml PRO – Benjamin Canou

March 9-13 2015

Outline

Functor Definition

Functor definition
Example: Multisets
Functor signatures
Functor application

Functor definition

Syntax:

- Basic definition: `module F (X : T) = struct ... end`.
- The type of the argument must be given.
- The return type of the functor is infered and the most general.
- `F` is the name of the functor.
- `X` is the name of the argument usable in the body.
- `T` can be any signature.

Optionally,

- The result signature can be specified.
- Signature definition:
  `module F (X : T) : sig ... end = struct ... end`.
- `X` is usable in the signature body.
  `module F (X : T) : sig type t = X.t end = struct ... end`.
- Named definition:
  `module F (X : T) : S = struct ... end`.
- The argument and return can also be related with signature rewriting.
  `module F (X : T) : S with type t = X.t = struct ... end`.

Example: Multisets

We start from a quick and dirty implementation of multisets.

```
let empty = []
let rec add elt = function
  | [] -> [ (elt, 1) ]
  | (e, m) :: rest when elt = e ->
    (e, succ m) :: rest
  | (e, _) as p :: rest when elt > e ->
    p :: (elt, 1) :: rest
  | p :: rest ->
    p :: add elt rest
let rec multiplicity elt = function
  | [] -> 0
  | (e, _) :: rest when elt > e  -> 0
  | (e, m) :: rest when elt = e -> m
  | _ :: rest -> multiplicity elt rest
```

First, we want to give it an interface.

- For documentation.
- To hide the type and protect the sorted invariant.

```
module Multiset : sig
  type 'a t
  val empty : 'a t
  val add : 'a -> 'a t -> 'a t
  val multiplicity : 'a -> 'a t -> int
end = struct
  type 'a t = ('a * int) list
  (* ... *)
end
```

Now, we want to build a functorial interface.

- To build distinguished multiset types over a same type.
- To use ad-hoc comparison instead of OCaml's.

So we the functor wil require:

- The type of elements.
- And a `compare` function over them.

The functor parameter signature is:

```
module type ELEMENT = sig
  type t
  val compare : t -> t -> int
end
```

Example: Multisets

Then, we write the functor definition, replacing OCaml's comparison.

```
module Make_multiset (E : ELEMENT) = struct
  type t = (E.t * int) list
  let empty = []
  let rec add elt = function
    | [] -> [ (elt, 1) ]
    | (e, m) :: rest when E.compare elt e = 0 ->
      (e, succ m) :: rest
    | (e, _) as p :: rest when E.compare elt e > 0 ->
      p :: (elt, 1) :: rest
    | p :: rest ->
      p :: add elt rest
  let rec multiplicity elt = function
    | [] -> 0
    | (e, _) :: rest when E.compare elt e > 0 -> 0
    | (e, m) :: rest when E.compare elt e = 0 -> m
    | _ :: rest -> multiplicity elt rest
end
```

With the infered syntax, t is public.

- One can break the invariants from the outside.
- We have to restrict the result with a signature with t abstract.

```
module Make_multiset (E : ELEMENT) : sig
  type t
  val empty : t
  val add : E.t -> t -> t
  val multiplicity : E.t -> t -> int
end = struct
 (* ... *)
end
```

The inline signature is not very readable, and cannot be reused.
We want to name the signature or the result.

```
module type MULTISET = sig
  type t
  val empty : t
  val add : (* ?? *) -> t -> t
  val multiplicity : (* ?? *) -> t -> int
end
module Make_multiset (E : ELEMENT) : MULTISET = struct
  (* ... *)
end
```

Problem: with `t` abstract, E out of scope, what is the type of elements ?

Example: Multisets

Solution: we introduce the type of elements as an abstract type.

```
module type MULTISET = sig
  type t
  type elt
  val empty : t
  val add : elt -> t -> t
  val multiplicity : elt -> t -> int
end
```

Then we have to make it public again, with signature rewriting:

```
module Make_multiset (E : ELEMENT)
  : MULTISET with type elt = E.t = struct
  type elt = E.t
  type t = (elt * int) list
  (* ... *)
end
```

A functor must appear with its type in interfaces, as any other module.

- Simple syntax: `module F : functor (X : T) -> sig ... emd`
- With named result: `module F : functor (X : T) -> S`
- And rewriting: `module F : functor (X : T) -> S with type t = X.t`

Interface of the example:

```
module type ELEMENT = sig (* ... *) end
module type MULTISET = sig (* ... *) end
module Make_multiset : functor (E : ELEMENT) ->
  MULTISET with type elt = E.t
```

Let's write a little test program.

```
let () =
  let module SMS = Make_multiset (String) in
  let items = [ "a" ; "a" ; "c" ; "b" ; "c" ; "c" ] in
  let sms = List.fold_right SMS.add items SMS.empty in
  assert
    (List.map
       (fun l -> SMS.multiplicity l sms)
       [ "a" ; "b" ; "c" ; "d" ]
     = [ 2 ; 1 ; 3 ; 0 ])
```

Now we want to have two multiset instances.

```
let () =
  let module Wines = Make_multiset (String) in
  let module Beers = Make_multiset (String) in
  let wine_cellar = ref Wines.empty in
  let beer_cellar = ref Beers.empty in
  wine_cellar := Wines.add "Bordeaux" !wine_cellar ;
  wine_cellar := Wines.add "Puglia" !wine_cellar ;
  wine_cellar := Beers.add "Heineken" !wine_cellar ;
  wine_cellar := Wines.add "Bordeaux" !wine_cellar
```

Problem: for a small typo, now I have Heineken in my wine cellar !

OCaml considers equal all applications of a functor to the same module:

- This equality is based on the name, not the structure
  `String` is not the same as `struct include String end`.
- Module aliases are recognized.
  `module S1 = String ;; module S2 = String` makes `S1` and `S2` equal.

This optimization can be useful, but here, we don't want this aliasing.

So we don't use `String` directly:

```
let () =
  let module Wines =
    Make_multiset (struct include String end) in
  let module Beers =
    Make_multiset (struct include String end) in
  (* ... *)
```

And we get sanctionned, as expected.

```
                    Characters 361-373:
    wine_cellar := Beers.add "heineken" !wine_cellar ;
                                        ^^^^^^^^^^^^
Error: This expression has type Wines.t
       but an expression was expected of type Beers.t
```

First Class Modules

Fairly simple:

- Make a module a value:
  `let m = (module M : S)`
  Locally: `let m = (module M : S) in ...`
- Make a value a module:
  `module M = (val m : S)`
  Locally: `let module M = (val m : S) in ...`
- Type of a first class module: `(module S)`

First class modules

Usage examples:

- Run-time selection of an impelmentation.
- Plug-ins, combined with dynlink.
- Alternative to objects.

Example:

- Different implementations for a same type.
- Implementation stored in the value with the instance.

The type of (string x string) tables

```
module type TABLE = sig
  type table
  type param
  val init : param -> table
  val put : string -> string -> table -> unit
  val get : string -> table -> string
end
```

In memory implementation:

```
module In_memory_table = struct
  type table = (string, string) Hashtbl.t
  type param = unit
  let init () =
    Hashtbl.create 100
  let put n v table =
    Hashtbl.replace table n v
  let get n table =
    Hashtbl.find table n
end
```

Files implementation:

```
module On_disk_table = struct
  type table = string
  type param = string
  let init to_string of_string dir =
    Unix.mkdir dir 0o750 ;
    dir
  let put n v dir =
    let fp = open_out (Filename.concat dir n) in
    output_string fp v ;
    close_out fp
  let get n dir =
    let fn = Filename.concat dir n in
    if not (Sys.file_exists fn) then raise Not_found ;
    (* ... *)
end
```

An intermediate module type, storing the implementation and instance.

```
module type TABLE_INSTANCE = sig
  include TABLE
  val instance : table
end
```

The first class module type and its primitives.

```
type table = (module TABLE_INSTANCE)
let in_memory_table =
  let module Instance = struct
    include In_memory_table
    let instance = init ()
  end in
  (module Instance : TABLE_INSTANCE)
let get (module Table : TABLE_INSTANCE) n =
  Table.(get n instance)
let put (module Table : TABLE_INSTANCE) n v =
  Table.(put n v instance)
```

Code Review: OCamlGraph

Outline