Introduction to Modules in OCaml

Woosuk Lee

CSE 6049 Program Analysis

Hanyang University, Korea

Some slides are borrowed from https://github.com/prosyslab-classroom/is593-2020-spring/blob/master/slides/lecture0.pdf

Module

A set of related definitions of types, values, and exceptions

```
module ListStack = struct
  type ‘a stack = ‘a list
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let peek = function
    | []    -> failwith "Empty"
    | x::_  -> x
  let pop = function
    | []     -> failwith "Empty"
    | _::xs -> xs
end
```

Module

```
# ListStack.push 1 ListStack.empty;;
- : int list = [1]

# open ListStack;;

# push 2 empty ;;
- : int list = [2]

# let open ListStack in
  peek (push 1 (push 2 empty));;
- : int = 1
```

Module Type

A set of declarations of types, values and exceptions

```
module type StackSig = sig
  type 'a stack
  val is_empty : 'a stack -> bool
  val push :'a->'a stack -> 'a stack
  val peek : 'a stack -> 'a
  val pop : 'a stack -> 'a stack
end
```

Module Type Allows to Hide Internals

```
module ListStack : StackSig = struct
  . . .
end
```

```
# ListStack.push 1 ListStack.empty;;
- : int ListStack.stack = <abstr>
```

Module Type

- `module A : Sig = struct … end`
- OCaml type checker checks if
  - `A` contains everything in the module type `Sig`, and
  - Something in `A` not declared in `Sig` is accessed from the outside

Example

```
module type S1 = sig
  val x:int
  val y:int
end
module M1 : S1 = struct
 let x = 42 end
(* type error:
   Signature mismatch:
   The value `y' is required but not provided
*)
```

Example

```
module type S2 = sig
 val x:int
end
module M2 : S2 = struct
 let x = 42
 let y = 7
end
M2.y
(* type error: Unbound value M2.y *)
```

Question

- Which one is without any compilation error?

A.
```
module M =
  (struct let inc x = x+1 end : sig end)
```

B.
```
module M =
  (struct let inc x = x+1 end : sig val inc end)
```

C.
```
module M =
  (struct let inc x = x+1 end
  : sig val inc : int -> int end)
```

D. All

Question

- Which one is without any compilation error?

A.
```
module M =
  (struct let inc x = x+1 end : sig end)
```

B.
```
module M =
  (struct let inc x = x+1 end : sig val inc end)
```

Type should be specified

C.
```
module M =
  (struct let inc x = x+1 end
  : sig val inc : int -> int end)
```

D. All

Functors

- A function from modules to modules
- Similar to template in C++ or generic in Java

```
module type X = sig
  val x : int
end

module IncX (M: X) = struct
  let x = M.x + 1
end

module A = struct let x = 0 end
(* A.x is 0 *)

module B = IncX(A)
(* B.x is 1 *)
```

OCaml Standard Library

- A collection of useful modules
- Data structures, algorithms, system calls, etc
  - For example, the List module has a number of utility functions for lists http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
- Manual: http://caml.inria.fr/pub/docs/manual-ocaml/libref
- Code: https://github.com/ocaml/ocaml/tree/trunk/stdlib

Stdlib

- The `Stdlib` module is automatically opened
- A lot of basic operations over the built-in types (numbers, booleans, strings, I/O channels, etc)
- http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html

Set

- The set data structure and functions
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/Set.html
- The `Set.Make` functor constructs implementations for any type
- The argument module must have a type `t` and a `compare` function

```
module IntPairs = struct
  type t = int * int
  (* a total ordering function with type t -> t -> int is required *)
  (* compare x y is -1 if x < y, 0 if x = y, 1 otherwise *)
  let compare (x0, y0) (x1, y1) =
    match compare x0 x1 with  (* this compare is a builtin function *)
    | 0 -> compare y0 y1
    | c -> c
end
module PairSet = Set.Make(IntPairs)
```

Set

- Builders

```
(* type elt is the type of the set element *)

(* val empty : t *)
let emptyset = PairSet.empty

(* val add : elt -> t -> t *)
let x = PairSet.add (1,2) emptyset

(* val singleton : elt -> t *)
let y = PairSet.singleton (1,2)

(* val remove : elt -> t -> t *)
let z = PairSet.remove (1,2) y

(* set operators with type t -> t -> t *)
let u = PairSet.union x y
let i = PairSet.inter x y
let d = PairSet.diff x y
```

Set

- Iterators

```
(* val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a *)
let sum_left = PairSet.fold (fun (i, _) s -> i + s) x 0

(* val iter : (elt -> unit) t -> t *)
let _ = PairSet.iter (fun i, _) -> print_int i) x

(* val map : (elt -> elt) -> t *)
let double = PairSet.map (fun (i, j) -> (2 * i, 2 * j)) x
```

- Searching

```
(* val mem : elt -> t -> bool *)
let membership = PairSet.mem (1, 2) x

(* val filter : (elt -> bool) -> t -> t *)
let big_left = PairSet.filter (fun (i, j) -> i > j) x
```

Map

- The map data structure and functions
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html
- The `Map.Make` functor constructs implementations for any type
- The argument module must have a type `t` and a `compare` function

```
module IntPairs = struct
  type t = int * int
  (* a total ordering function with type t -> t -> int is required *)
  (* compare x y is -1 if x < y, 0 if x = y, 1 otherwise *)
  let compare (x0, y0) (x1, y1) =
    match compare x0 x1 with  (* this compare is a builtin function *)
    | 0 -> compare y0 y1
    | c -> c
end
module PairMap = Map.Make(IntPairs)
```

Map

- Builders

```
(* type key is the type of the map keys *)
(* type 'a t is the type of maps is from type key to type 'a *)

(* val empty : 'a t *)
let emptymap = PairMap.empty

(* val add : key -> 'a -> bool *)
let x = PairMap.add (1,2) "one-two" emptymap

(* val singleton : key -> 'a -> 'a t *)
let y = PairMap.singleton (1,2) "one-two"

(* val remove : key -> 'a t -> 'a t *)
let z = PairMap.remove (1,2) y
```

Map

- Iterators

```
(* val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b *)
let sum = PairMap.fold (fun (i, j) _ s -> (i + j) x 0

(* val iter : (elt -> unit) t -> t *)
let _ = PairMap.iter (fun (i, _) -> print_int i) x

(* val map : ('a -> 'b) -> 'a t -> 'b t *)
let double = PairMap.map (fun str -> String.length str) x
```

- Searching

```
(* val mem : key -> 'a t -> bool *)
let membership = PairMap.mem (1, 2) x

(* val filter : (key -> 'a -> bool) -> 'a t -> 'a t *)
let big_left = PairMap.filter (fun (i, j) _ -> i > j) x
```