Programming Languages and Environments (Lecture 12)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Syllabus

- Modules

Module systems

- Name spaces
  - Groups of declarations (usually) related, isolated from other groups through name qualification in a module.
  - It allows the reuse of the same names in different contexts without collisions.
  - Examples: Packages and classes in Java, files/modules in C, structures/modules in OCaml.
- Abstraction
  - Allows selectively hiding/revealing information (*information hiding*).
  - Code isolation, better development and maintenance, ownership, etc.
- Code reuse
  - Reuse without copying, modularity (cf. inheritance in Java).
- Module parametrization (in OCaml)
  - Functors in OCaml are like functions from modules to modules (cf. traits in Scala).

Modules in OCaml

- Modules are defined by **structures** (`struct`).
- The types for the modules are **signatures** (`sig`).
- The type definitions by default are **public** (`type`).
- The implementations of names are **private** (`val`).

```
module MyList = struct
  type 'a list = Nil | Cons of 'a * 'a list
  let empty = Nil
  let rec length = function
    | Nil → 0
    | Cons (_, xs) → 1 + length xs
  let insert x xs = Cons (x, xs)
  let head = function
    | Nil → None
    | Cons (x, _) → Some x
  let tail = function
    | Nil → None
    | Cons (_, xs) → Some xs
end
```

[7] ✓ 0.0s

```
module MyList :
  sig
    type 'a list = Nil | Cons of 'a * 'a list
    val empty : 'a list
    val length : 'a list → int
    val insert : 'a → 'a list → 'a list
    val head : 'a list → 'a option
    val tail : 'a list → 'a list option
  end
```

Name spaces

- The names declared in a module can be used in a qualified manner (with the name followed by a dot: `List.fold_right`).
- Alternatively, the **`open`** directive can be used to expand the names of the module in the client module.
- The **`Stdlib`** module is always open.

```
module MyStack = struct
  type 'a stack = 'a MyList.list
  let empty = MyList.empty
  let push x xs = MyList.insert x xs
  let pop = MyList.tail
  let top = MyList.head
end
```

[17] ✓ 0.0s

```
module MyStack :
  sig
    type 'a stack = 'a MyList.list
    val empty : 'a MyList.list
    val push : 'a → 'a MyList.list → 'a MyList.list
    val pop : 'a MyList.list → 'a MyList.list option
    val top : 'a MyList.list → 'a option
  end
```

```
module MyStack = struct
  open MyList

  type 'a stack = 'a list
  let empty = empty
  let push x xs = insert x xs
  let pop = tail
  let top = head
end
```

[14] ✓ 0.0s

```
module MyStack :
  sig
    type 'a stack = 'a MyList.list
    val empty : 'a MyList.list
    val push : 'a → 'a MyList.list → 'a MyList.list
    val pop : 'a MyList.list → 'a MyList.list option
    val top : 'a MyList.list → 'a option
  end
```

Name abstraction

- The types of modules also allow hiding the definition of types.
- A signature can have multiple compatible (opaque) implementations.

```
module type Stack =
  sig
    type 'a stack
    val empty : 'a stack
    val push : 'a → 'a stack → 'a stack
    val pop : 'a stack → 'a stack option
    val top : 'a stack → 'a option
  end

module MyStack : Stack

module AnotherStack : Stack
```

```
module type Stack = sig
  type 'a stack
  val empty : 'a stack
  val push : 'a → 'a stack → 'a stack
  val pop : 'a stack → 'a stack option
  val top : 'a stack → 'a option
end

module MyStack:Stack = struct
  type 'a stack = 'a MyList.list
  let empty = MyList.empty
  let push x xs = MyList.insert x xs
  let pop = MyList.tail
  let top = MyList.head
end

module AnotherStack : Stack = struct
  type 'a stack = 'a list
  let empty = []
  let push x xs = x :: xs
  let pop = function
    | [] → None
    | _ :: xs → Some xs
  let top = function
    | [] → None
    | x :: _ → Some x
end
```

Types and names

- The specialization of modules can be done with an adaptation module.

```
module IntStack = (struct
  type stack = int MyStack.stack
  let empty = MyStack.empty
  let push = MyStack.push
  let pop = MyStack.pop
  let top = MyStack.top
end : sig
  type stack
  val empty : stack
  val push : int → stack → stack
  val pop : stack → stack option
  val top : stack → int option
end)
```

[36] ✓ 0.0s

```
module IntStack :
  sig
    type stack
    val empty : stack
    val push : int → stack → stack
    val pop : stack → stack option
    val top : stack → int option
  end
```

Modules and Files

- The organization into separate files separates the structure (`struct`) from the signature (`sig`).
- Files like `MyList.ml`, `MyStack.mli`, and `MyStack.ml` are examples.

s > LAP 2024-12 > MyList.ml > ...

```
type 'a list = Nil | Cons of 'a * 'a list
'a list
let empty = Nil
'a list -> int
let rec length = function
  | Nil → 0
  | Cons (_, xs) → 1 + length xs
'a -> 'a list -> 'a list
let insert x xs = Cons (x, xs)
'a list -> 'a option
let head = function
  | Nil → None
  | Cons (x, _) → Some x
'a list -> 'a list option
let tail = function
  | Nil → None
  | Cons (_, xs) → Some xs
```

> LAP 2024-12 > MyStack.mli > ...

```
type 'a stack
val empty : 'a stack
val push : 'a → 'a stack → 'a stack
val pop : 'a stack → 'a stack option
val top : 'a stack → 'a option
```

s > LAP 2024-12 > MyStack.ml > ...

```
type 'a stack = 'a MyList.list
'a
let empty = MyList.empty
'a -> 'b -> 'c
let push x xs = MyList.insert
'a
let pop = MyList.tail
'a
let top = MyList.head
```

Modules and Functors (module-to-module functions)

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

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

[23] ✓ 0.0s

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

```
module IncX : functor (M : X) → sig val x : int end
```

Modules and Functors (module-to-module functions)

```ocaml
module type X = sig
  type t
end

module Stack = struct
  module Make (M : X) = struct
    type stack = M.t list
    let empty = []
    let push x xs = x :: xs
    let pop = function
      | [] -> None
      | _ :: xs -> Some xs
    let top = function
      | [] -> None
      | x :: _ -> Some x
  end
end

module IntStack = Stack.Make (struct type t = int end)

let s = IntStack.empty
let _ = assert (IntStack.top s = None)
let _ = assert (IntStack.top (IntStack.push 1 s) = Some (1))
let _ = assert (IntStack.pop (IntStack.push 1 s) = Some (IntStack.empty))
```

[71] ✓ 0.0s

Modules and Functors (module-to-module functions)

```
module Pair = struct
  type t = int * string
  let compare (x1, y1) (x2, y2) =
    if x1 < x2 then -1
    else if (x1 = x2 && y1 < y2) then -1
    else if (x1 = x2 && y1 = y2) then 0 else 1
end

module Str = Set.Make(Pair)

let s = Str.empty
let s = Str.add (1, "a") s
let s = Str.add (2, "b") s
let s = Str.add (3, "c") s
let s = Str.add (4, "d") s
let _ = assert Str.(mem (1, "a") s)
```

[83] ✓ 0.0s