Programming Languages and Environments (Lecture 15)

LEI - Licenciatura em Engenharia Informática

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

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Syllabus

- Module system
- Functors
- Concurrency
  - Promises

Module System

- Name spaces
  - Groups of statements (usually) related isolated from other groups through the qualification of the names in a module.
  - Allows the reuse of the same names in different contexts without collisions.
  - Packages and classes in Java, files/modules in C, structures/modules in OCaml
- Abstraction
  - Allows you to selectively hide/reveal information (information hiding)
  - Code isolation, better development and maintenance, ownership, etc.
- Code reuse
  - Reuse without copy, modularity, (cf. inheritance in Java)
- (In OCaml) Module parameterisation
  - The Functors in OCaml are like functions from modules to modules (cf. traits in Scala)

Modules in OCaml

- The modules are defined by **structures** (**struct**)
- The types for the modules are **signatures** (**sig**)
- Default type definitions are public (**type**)
- Name implementations are private (**val**)

```
module MyModule = struct
  type primary_color = Red | Green | Blue

  let inc x = x + 1
  let dec x = x - 1
end
```

```
(* Inferred signature *)

module MyModule :
  sig
    type primary_color = Red | Green | Blue
    val inc : int -> int
    val dec : int -> int
 end
```

utop

Namespaces

- The names declared in a module can be used in a qualified way (with the module name and a period: **List.fold_right**)
- Or you can use the **open** directive to expand the names of the module used in the client module.
- The **StdLib** module is always open.

```
module M = struct
  let x = 42
end

M.x                     (* 42 *)
x                       (* Unbound value x *)

let y = M.(x * x + x) (* val y : int = 1806 *)

open M
x                       (* 42 *)
```

Module MyList

```
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
```

```
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
```

Java vs. Ocaml

- Java

```
s = new List();
s.insert(1);
```

- OCaml

```
let s  = MyList.empty;;
let s’ = MyList.insert 6 s;;
```

Name abstraction

- The types of the modules also allow you hide the definition of the types
- A subscription can have multiple compatible implementations (opaque)

```
module type Stack = sig
  type 'a stack
  val empty    : 'a stack
  val is_empty : 'a stack -> bool
  val push     : 'a -> 'a stack -> 'a stack
  val top      : 'a stack -> 'a option
  val pop      : 'a stack -> 'a stack option
end
```

```
module ListStack : Stack = struct
  type 'a stack = 'a list
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let top = function
    | [] -> None
    | x :: _ -> Some x
  let pop = function
    | [] -> None
    | _ :: xs -> Some xs
end
```

```
module MyListStack : Stack = struct
  type 'a stack = 'a MyList.list
  let empty = MyList.empty
  let is_empty = MyList.is_empty
  let push x s = MyList.insert x s
  let top = MyList.head
  let pop = MyList.tail
end;;
```

Namespaces (again)

```
module MyListStack : Stack = struct


  type 'a stack = 'a MyList.list
  let empty = MyList.empty
  let is_empty = MyList.is_empty
  let push x s = MyList.insert x s
  let top = MyList.head
  let pop = MyList.tail
end;;
```

```
module MyListStack : Stack = struct
  open MyList

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

Module ListStackCachedSize

- Implement and test!

```
module ListStackCachedSize : Stack = struct
  type 'a stack = 'a list * int

  let empty = ([], 0)
  let is_empty s =
    match s with
    | ([], _) -> true
    | _ -> false
  let push x s = (x::(fst s), (snd s)+1)
  let top s = match s with
    | ([], _) -> None
    | (x :: xs, _) -> Some x
  let pop s = match s with
    | ([], _) -> None
    | (x :: xs, n) -> Some (xs, n-1)
  let size s = snd s
end
```

Module Counter with Refs!

- Implement and test!

```
module type Counter = sig
  type t
  (** [create v] makes a new counter the initial value [v]. *)
  val create : int -> t

  (** [inc c] increments the counter by 1. *)
  val inc    : t -> unit

  (** [dec c] decrements the counter by 1. *)
  val dec    : t -> unit

  (** [get c] returns the current value. *)
  val get    : t -> int

  (** [reset c] sets the counter to zero. *)
  val reset  : t -> unit
end
```

```
module CounterRef : Counter = struct
  type t = int ref

  let create v = ref v

  let inc c = c := !c + 1

  let dec c = c := !c - 1

  let get c = !c

  let reset c = c := 0

end
```

Types and names

- The specialisation of module types can be done with an adaptation module.

```
module IntStack = (struct
  (* 1. Build on a generic "ListStack" module
        by fixing its element type to int. *)
  type stack = int MyListStack.stack

  (* 2. Re-export the operations from ListStack *)
  let empty = MyListStack.empty
  let push  = MyListStack.push
  let pop   = MyListStack.pop
  let top   = MyListStack.top
end : sig
  (* 3. Users of IntStack only see the abstract type [stack]
        and the four operations with the following types *)
  type stack
  val empty : stack
  val push  : int -> stack -> stack
  val pop   : stack -> stack option
  val top   : stack -> int option
end)
```

```
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 organisation in files separates the structure (struct) from the signature (sig)
- Files MyList.ml, Stack.mli, MyStackList.ml

LAP-2025 > MyList.ml > ...

```ocaml
module MyList = struct
  type 'a list = Nil | Cons of 'a * 'a list

  let empty = Nil

  let is_empty = function  -> true | _ -> false

  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
```

LAP-2025 > Stack.mli > ...

```ocaml
module type Stack = sig
  type 'a stack
  val empty    : 'a stack
  val is_empty : 'a stack -> bool
  val push     : 'a -> 'a stack -> 'a stack
  val top      : 'a stack -> 'a option
  val pop      : 'a stack -> 'a stack option
end
```

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

```ocaml
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 functions for modules)

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

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

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

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

Modules and Functors (module functions for modules)

```
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)
```

```
module Stack :
  sig
    module Make :
      functor (M : X) ->
        sig
          type stack = M.t list
          val empty : 'a list
          val push : 'a -> 'a list -> 'a list
          val pop : 'a list -> 'a list option
          val top : 'a list -> 'a option
        end
  end
```

Modules and Functors (module functions for modules)

```
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)
```

Concurrency

Concurrency

- Capacity to perform several calculations that overlap in time, and not require them to be done in sequence.
  - Graphical interfaces; to ensure quick responses to user actions.
  - Spreadsheets; to recalculate all calculations without interruptions.
  - Web Browser; to load and show pages incrementally.
  - Servers; to serve several customers without making them wait.
- How?
  - Interlacing: switching between the various active computes quickly.
  - Parallelism: using several physical processors (multi-core).
- Preemptive / Collaborative

Non-determinism

- A program that can have different results each time it runs is a non-deterministic program.
- The introduction of competition causes non-determinism, by not fixing the order in which operations are executed.
- When two imperative programs share state variables, the possibilities of changing that same state are indeterminate. Therefore, it is not easy to predict the exact behaviour of the programs.
- Interactions/interferences between programs are benign or malignant (race conditions)
- Functional languages make it easier to reason about programs because a pure expression always denotes the same value.

Non-determinism

```
x = 0
```

Thread A

```
A1: READ(x)
A2: WRITE(x, x+1)
```

Thread B

```
B1: READ(x)
B2: WRITE(x, x+1)
```

```
x = ?
```

Promises or Futures

- They represent computations that have not yet ended but that will denote a value somewhere in a future (deferred) instant.

- Usually associated with a concurrent computing to the main thread.

```
async function getNames() {
  const response = await fetch('https://server.com/users')
  const data = await response.json()
  return data.map(user => user.name)
}
```

- Async (Jane Street) and Lwt (Ocsigen) are two popular libraries for implementing asynchronous computing in OCaml.

Lwt-style promises

- They are data abstractions for an asynchronous computing model.

- Promises are references, their value can change.
- When it is created, it does not contain anything.
- A promise can be fulfilled and fulfilled for a value
- A promise can be rejected (filled by an exception)
- In both cases it is said to be resolved.

Signature of the Promise Module

```
module type PROMISE = sig
  type 'a state = Pending | Fulfilled of ‘a | Rejected of exn

  type 'a promise

  type 'a resolver

  (** [make ()] is a new promise and resolver. The promise is pending. *)
  val make : unit -> 'a promise * 'a resolver

  (** [return x] is a new promise that is already fulfilled with value [x]. *)
  val return : 'a -> 'a promise

  (** [state p] is the state of the promise *)
  val state : 'a promise -> 'a state

  (** [fulfill r x] fulfills the promise [p] associated with [r] with value [x], meaning that
      [state p] will become [Fulfilled x].  Requires: [p] is pending. *)
  val fulfill : 'a resolver -> 'a -> unit

  (** [reject r x] rejects the promise [p] associated with [r] with exception [x], meaning that
      [state p] will become [Rejected x].  Requires: [p] is pending. *)
  val reject : 'a resolver -> exn -> unit
end
```

Implementation of the Promise Module

```
module Promise : PROMISE = struct
  type 'a state =
    | Pending
    | Fulfilled of 'a
    | Rejected of exn

  type 'a promise = 'a state ref

  type 'a resolver = 'a promise

  (** [write_once p s] changes the state of [p] to be [s]. If [p] and [s] are both pending,
      that has no effect. Raises: [Invalid_arg] if the state of [p] is not pending. *)
  let write_once p s =
    if !p = Pending then p := s else invalid_arg "cannot write twice”

  let make () = let p = ref Pending in (p, p)

  let return x = ref (Fulfilled x)

  let state p = !p

  let fulfill r x = write_once r (Fulfilled x)

  let reject r x = write_once r (Rejected x)
end
```

Use of Promises

```
(** Lets see how to use promises *)
let compute_fact_sync n : int Promise.promise =
  let p, r = Promise.make () in
  begin
    try
      let result = fact n in       (* pure factorial function *)
      Promise.fulfill r result     (* settle with the computed value *)
    with exn ->
      Promise.reject r exn         (* an error, which never happens here*)
  end;
  p

let check name p =
  match Promise.state p with
  | Pending ->
      Printf.printf "%s is still computing…\n" name
  | Fulfilled v ->
      Printf.printf "%s = %d\n" name v
  | Rejected exn ->
      Printf.printf "%s failed\n" name

let p5 = compute_fact_sync 5 in check "5!" p5
```

Use of Promises in LWT

```
(* Pure-function *)
let rec fact n =
  if n <= 1 then 1 else n * fact (n - 1)

(* Initialize the preemptive pool with default bounds (min=0, max=4) *)
let () = Lwt_preemptive.simple_init ()

(* Offload [fact n] onto Lwt's preemptive pool *)
let compute_fact n : int Lwt.t =
  Lwt_preemptive.detach fact n

let () =
  let work = [
    compute_fact 20 >>= fun r -> Lwt_io.printf "20! = %d\n" r;
    compute_fact 25 >>= fun r -> Lwt_io.printf "25! = %d\n" r;
  ] in
  Lwt_main.run (Lwt.join work)
```

In utop:

```
#require "lwt.unix";;
#open Lwt;;
#open Lwt.Infix;;
```