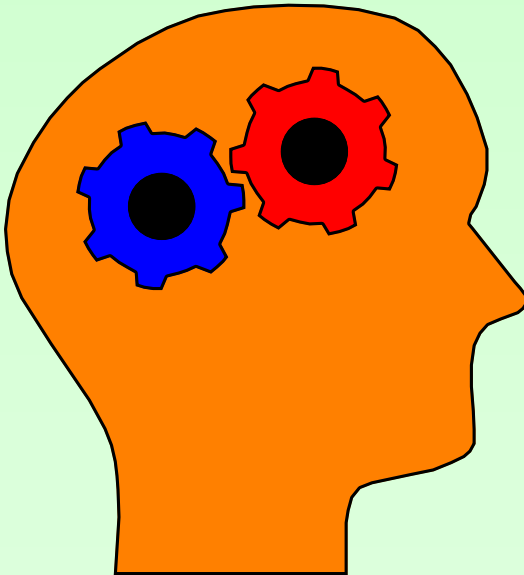




CS2104: Programming Languages Concepts

Lecture 10 : OCaml Basics and Imperative Programming



“Basics of OCaml and Imperative Programming

Lecturer : Chin Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Topics

- Basics of OCaml
- Mutable Reference & Records
- Input/Output Functions
- Iterators and Loops
- Arrays, String, Hash-Tables Modules
- Memoization

OCaml vs Haskell

OCaml	Haskell
strict (by default)	lazy (by default)
impure	pure
let & let rec	let (rec by default)
Impure IO	Monadic I/O
Modules/Functors	Type classes
OOP	Layout Rule
	Comprehension

Non-Recursive Let

- In OCaml, `let` binding is non-recursive.

```
let x = 1 in      // 1
let x = x-3 in    // 1-3
```

- For (mutual) recursive, use `let rec` instead:

```
let rec f x = ..g x..
      and g x = ..f x..g x..
in ...
```

- Non-recursive `let` supports shadowing and allows reuse of names.

Post-fix Type Application

- Type variables are written as ``a` in OCaml
- In Haskell, prefix type application used

`map :: (a->b) -> ([]) a -> ([]) b`

- In OCaml, post-fix type application used

`map :: (`a->`b) -> `a list -> `b list`

Strict Evaluation

- `let` binding is strictly evaluated.

```
let x = e1 in      // e1 is strictly evaluated
let y = e2 in      // e2 is strictly evaluated
..
```

- Arguments are strictly evaluated but the order of evaluation is not specified

```
f e1 e2 // {e1,e2} are strictly evaluated
```

- Use `let` if argument order is important.

```
let v1 = e1 in // e1 is evaluated
let v2 = e2 in // e2 is evaluated
f v1 v2        // call is invoked
```

Lazy Module

- Laziness in OCaml is captured through type

```
t Lazy.t
```

- Deferred computation is built using:

```
let (v: `a Lazy.t) = lazy (e: `a)
```

- It is strictly evaluated through:

```
force : `a Lazy.t -> `a  
... force v ..
```

- Can check if already evaluated using:

```
is_val : `a Lazy.t -> bool
```

Pure versus Imperative

Pure versus Imperative

- Thus far, used mostly pure functions (without side-effects).
- Benefits of Pure Functions
 - Easier to reason/debug
 - Less error prone
 - Easily composable

When is Imperative Needed?

- Imperative feature also important
 - To model the real-world
 - For some high performance considerations (e.g. hash tables & mutable graphs)
- Compromise:
 - Use pure functions *where possible* (by default), and imperative features *where necessary*.

Mutable Reference and Fields

Mutable Reference

- Reference type supports pointer to a memory location that can be updated.

```
(* creates a mutable reference ptr *)  
let ptr = ref 10;;
```

```
(* prints content of ptr *)  
print_endline (string_of_int !ptr);;
```

```
(* update the content of ptr *)  
ptr := !ptr + 1;;
```

```
print_endline (string_of_int !ptr);;
```

Creating Mutable Reference

- Command `ref` has type `'a → 'a ref`.

```
(* creates a mutable reference ptr *)  
let ptr = ref 10;;  
  
let ptr2 = ref "hello";;
```

- In above example:
 - `ptr` has `(int ref)` type.
 - `ptr2` has `(string ref)` type

Dereferencing Mutable Locations

- Command `!` has type `'a ref → 'a`.

```
(* returns content of a mutable reference *)  
let v:int = !ptr;;
```

```
let s:string = !ptr2;;
```

- In follow-up example:
 - `v` has `int` type.
 - `s` has `string` type

Updating Mutable Reference

- Command `:=` has type `'a ref → 'a → unit`.

```
(* updating a mutable reference *)
```

```
let () = ptr := !ptr + 1;;
```

```
let () = ptr2 := !ptr2 ^ " there!";;
```

- In follow-up example:
 - `ptr` content will change to `11`.
 - `ptr2` will change to `"hello there!"`
- Assignment returns unit (or void) type.

Mutable Fields of Record

- Records are immutable by default, but we can selectively provide fields that are mutable.
- Example:

```
(* declaring a record type *)
type ('a, 'b) pair =
  { mutable first : 'a;
    second : 'b
  } ;;
```

```
(* constructing a record value *)
let p1 = {first = 1; second = "cs2104"};;
```


Retrieve Fields of Record

- We can retrieve mutable and immutable fields in the same way, namely **record.field**.

```
print_endline ("First: "^(string_of_int p1.first));;  
print_endline ("Second: "^(p1.second));;
```

Updating Mutable Fields Only

- Only mutable fields can be updated.

```
(* update to mutable field *)  
p1.first <- p1.first + 1;;
```

```
(* compile time error! *)  
p1.second <- p1.second^" hello";;  
➔ Error: The record field label second is not mutable
```

Implementation of Ref Type

- How are reference types related to mutable fields of records?
- Each ref type is implemented as a record with exactly one mutable field!
- Thus, mutable fields are the fundamental construct in OCaml.

Implementation of Ref Type

```
(* type declaration *)
```

```
type 'a ref = { mutable contents : 'a };;
```

```
(* construction *)
```

```
let ref v = { contents = v };;
```

```
(* retrieval *)
```

```
let (!) r = r.contents;;
```

```
(* update *)
```

```
let (:=) r v = r.contents <- v;
```

Weak Polymorphism

- Types of mutable fields must not be polymorphic. Following is invalid:

```
let (p4: ('a list) ref) = ref [];;
```

```
p4 := 1 :: !p4;
```

```
p4 := "hello" :: !p4;; (* error *)
```

- Mutable reference must have a single type, so that update and retrieval remains consistent.

Weak Polymorphism

- Immutable values can be polymorphic:

```
let id x = x;;  
(* id: forall 'a. 'a -> 'a *)
```

- Mutable reference can only be monomorphic.

```
let p_id = ref id;;  
(* p_id : ('_a -> '_a) ref *)
```

**weak polymorphism
that must be bound
to a monotype**

➔ Error: The type of this expression, ('_a -> '_a) ref, contains type variables that cannot be generalized

Weak Polymorphism

- Weakly polymorphic type must be instantiated to a monotype within the same module.

```
let p_id = ref id;;  
(* p_id : ('_a -> '_a) ref *)
```

```
let v = !p_id 3;;  
(* forces '_a be instantiated to int *)
```

```
let s = !p_id "hello";;  
(* contradictory use cause type error *)
```

An Example

- Remembering first value.

```
let memo =  
  let cache = ref None in  
  (fun x ->  
    match !cache with  
    | Some y -> y  
    | None -> cache := Some x; x) ;;  
(* val memo : '_a -> '_a = <fun> *)
```

```
memo 3;; (* --> 3 *)
```

```
memo 4;; (* --> 3 *)
```


Wrongly Placed

- Cache is now local to each call !
where a new cache is created for each call.

```
let memo =  
  (fun x ->  
    let cache = ref None in  
    match !cache with  
    | Some y -> y  
    | None -> cache := Some x; x) ;;
```

```
memo 3;; (* --> 3 *)
```

```
memo 4;; (* --> 4 *)
```

Loops and Iterators

Loop Iterators

- OCaml provides for/while loops to make it easier for imperative programming.

```
for i = 0 to 3 do  
    printf "i = %d\n" i done;;
```



```
i = 0  
i = 1  
i = 2  
i = 3  
- : unit = ()
```

Note that *i* itself is
local and immutable

Loop Iterators

- A count-down for-loop.

```
for i = 3 downto 0 do  
    printf "i = %d\n" i done;;
```



```
i = 3  
i = 2  
i = 1  
i = 0  
- : unit = ()
```

Loop Iterators

- A while-loop.

```
let i = ref 3;;  
while (!i>=0) do  
  printf "i = %d\n" !i;  
  i := !i-1  
done;;
```

```
➔      i = 3  
        i = 2  
        i = 1  
        i = 0  
- : unit = ()
```

Loop Iterators

- Implementation of for_loop using higher-order

```
let for_loop init final stmt =  
  let rec aux i =  
    if i <= final  
    then (stmt i; aux (i+1))  
  in aux init
```

- Exercise : write recursive codes for while and for_downto loops.

List Iterator

- We also have iterators over data structures.

```
List.iter : ('a -> unit) -> 'a list -> unit
```

```
List.iter f [a1; ...; an]
```

applies function **f** in turn to **[a1; ...; an]**.

It is equivalent to

```
begin f a1; f a2; ...; f an; () end.
```

Sequences

Sequences

- Sequences of the form: `e1; e2; ..; en` are executed for their effects.
- It is equivalent to:

```
let _ = e1 in  
let _ = e2 in  
...  
en
```

Sequences

- Safer to use :

```
let () = e1 in  
let () = e2 in  
...  
en
```

- All except `en` are of the unit type. Why?

Evaluation Order

- Evaluation-order of arguments are implementation-dependent!
- What is the outcome of below?

```
let eval x =  
    printf "Elem %d\n" x; x ;;  
[eval 1; eval 2]
```



Elem 2

Elem 1

- : int list = [1; 2]

Evaluation Order

- Similarly:

```
let eval x =  
    printf "Elem %d\n" x; x ;;  
(eval 1, eval 2)
```



Elem 2

Elem 1

- : int * int = (1, 2)

Mutable Modules

Mutable Modules

- Quite a number of modules have mutable data structures.
- Examples are : Arrays, Strings, Hash Tables.
- They could be used to support both imperative and functional-style programming.

Array

- Key operators.

```
val make : int -> 'a -> 'a array
```

```
(* make n v returns a mutable array of  
   size n, with initial value v *)
```

```
val length : 'a array -> int
```

```
(* returns size of array *)
```

```
val get : 'a array -> int -> 'a
```

```
(* get a n returns n-th elem of array a *)
```

```
val set: 'a array -> int -> 'a -> unit
```

```
(* set a n v updates n-th elem of  
   array a with new value v *)
```

Array

- Short-hands.

```
val get : 'a array -> int -> 'a  
(* Array.get a n = a(n) *)
```

```
val set: 'a array -> int -> 'a -> unit  
(* Array.set a n v = a(n) <- v *)
```


Array

- Useful Higher-Order Functions.

```
val mapi : (int -> 'a -> 'b) ->  
  'a array -> 'b array
```

```
val fold_left : ('a -> 'b -> 'a) ->  
  'a -> 'b array -> 'a
```

```
val fold_right : ('b -> 'a -> 'a) ->  
  'b array -> 'a -> 'a
```

String

- We have used only string in a functional way.
- String are not only compact (8 characters per memory word), it also supports mutation:

```
val set: string -> int -> char -> unit
(* set s n c updates n-th elem of
   string s with value char c
   short-hand s.[n]<-c
*)
```

Hash Table (Hashtbl)

- Hash tables implement generic dictionary.

```
module Hashtbl
```

```
type ('a, 'b) t
```

```
val create : ?random:bool -> int  
            -> ('a, 'b) t
```

optional labeled
parameter



(* `Hashtbl.create n` will create a new hash table of initial size `n`, while `~random:true` allows a randomized seed to be used at creation *)

Hash Table

- Access operators.

```
val add : ('a, 'b) t -> 'a -> 'b -> unit  
(* (add tbl x y) to add (x,y) to tbl *)
```

```
find : ('a, 'b) t -> 'a -> 'b  
(* (find tbl x) returns current binding of x *)
```

```
remove : ('a, 'b) t -> 'a -> unit  
(* (remove tbl x) remove current binding of x *)
```

Memoization

Memoization

- Redundant calls can be eliminated by tabulation using dynamic programming.
- General Technique : Memoization.
- Can use hash table to store previously computed value for retrieval rather than re-computation.

Recursion

- Recall naïve fibonacci.

```
let rec fib n =  
    if n<=1 then 1  
    else fib (n-1) + fib(n-2) ;;
```

- Contains many redundant fib calls when computed with moderately sized inputs.

Using Memo-Functions

- This stores previously computed values in say a hash table. Example:

```
let fib_hash = Hashtbl.create 10;;

let rec fib_memo n =
  if n <= 1 then 1
  else try
    Hashtbl.find fib_hash n
  with _ ->
    let r = fib_memo (n-1) + fib_memo (n-2) in
    let _ = Hashtbl.add fib_hash n r in
    r
```

- Trade off memory space for time.

Pure Memo-Function

- Localize the effect of memo-table:

```
let fib_memo2 n =  
  let tbl = Hashtbl.create 10 in  
  let rec aux n =  
    if n<=1 then 1  
    else try  
      Hashtbl.find tbl n  
    with _ ->  
      let r = aux (n-1) + aux (n-2) in  
      let _ = Hashtbl.add tbl n r in  
      r  
  in aux n
```

- Pure function when viewed from outside.
Tradeoffs?

Input/Output

for reference

Input/Output

- Apart from data mutation, I/O is the other major source of side-effects.
- I/O involves interactions with the real world.
- Many I/O libraries including Asynchronous I/O library.

Buffered I/O Library

- This allows buffering to optimize interaction with the Unix I/O library.
- Two types of channels:
 - `in_channel` (for reading)
 - `out_channel` (for writing)

Common I/O Channels

`stdin` (* standard input *)

`stdout` (* standard output*)

`stderr` (* standard error *)

Interact with Terminal

```
let test () =  
    output_string stdout  
        "What is your name?";  
    flush stdout;  
    let ans = input_line stdin in  
        output_string stdout  
            ("Hello "^ans^"\n");;
```

Output to a File

```
let file = open_out "test.out";;  
(* creates an out_channel file *)  
  
output_string file "Hello There!";;  
(* writes to file *)  
  
close_out file;;  
(* closes the file *)
```

Formatted Printing

- Formatted printing takes a format string to determine how printing is to be done.

```
let pr = Printf.printf  
    ("%i is an integer, %F is a float"^^  
     "\"%s\" is a string\n");;
```

```
pr 3 4.5 "five";;
```

➔ 3 is an integer, 4.5 is a float, "five" is a string

Formatted Printing

- OCaml uses a type-safe solution:

```
let fmt = "%i is an integer, %F is a  
float"^^"\n%s\n" is a string\n");;
```

```
(int -> float -> string -> 'c, 'b, 'c) fmt
```

```
let pr = Printf.printf fmt;;
```

```
pr : int -> float -> string -> ()
```

```
pr 3 4.5 "five";;
```