01s — Overview/Review on OCaml

CS4215: Programming Language Implementation

Răzvan Voicu

Week 1 (January 9 - 13, 2017), supplementary material

1. Data Types
   - Basic Types
   - Tuple Type
   - Variant Type
   - Polymorphic Type
   - Error Values and Type
2. Expressions
3. Functions
4. Higher-Order Functions
5. Advanced Features

Strong Typing in OCaml

- Every expression has a type in OCaml
- We use type ascription `(expr:type)` to enforce it.
- Type polymorphism supported.
- Advanced types also supported : functions, objects, modules, GADT.

Basic Types Supported

```
let flag : bool = true                (* bool *)

let two : int = 2                     (* int *)
let double (x:int) = two * x          (* int *)

let doubleF (x:float) = 2.0 *. x (* float *)

let aChar : char = 'z'                (* char *)

let myname : string = "Razvan "^"Voicu" (* string *)
```

Tuple Type

- Tuple (or product) type of the form $(t_1 * \cdots * t_n)$ groups a number of types together. For example, `(int*string)` denotes a pair of int and string types.

```
let pair:(int*string) = (2,"hello")
let v1 = fst pair
let v2 = snd pair
```

- Predefined functions `fst` and `snd` retrieves the first and second element of its given pair, respectively.

Tuple Type

- One can define a triple (`int*string*string`), but this is *isomorphic* to nested pairs (`int*(string*string)`).
- Former is more efficient. Latter has better reuse.
- Corresponding access techniques:

```
(* using a new third function *)
let triple1 = (2,"hello","there")
let third (_,_,x) = x
let v3 = third triple1

(* using snd function twice *)
let triple2 = (2,("hello","there"))
let v3a = snd (snd triple2)
```

Variant Type

- Variant type can support enumerated and union types.

```
(* enumerated type *)
type color = Red | Blue | Green
(* union type *)
type intorstr = I of int | S of string
```

- It can also support recursive data structures, such as a list of integer.

```
type num = Zero | Succ num
type intList = Nil | Cons of int * intList
```

Type Polymorphism

- Better code reuse can be supported by type polymorphism.

```
type 'a list = Nil | Cons of 'a * ('a list)
type intList = int list
type 'a list_list = ('a list) list
```

- Polymorphic types are captured using type variables of the form `'a`, `'b`, `'c` etc.
- Polymorphic sum support union of two arbitrary types.

```
type ('a,'b) sum = L of 'a | R  of 'b
type intorstr = (int,string) sum
```

More Polymorphism Examples

- Polymorphic Binary Tree

```
type 'a tree = Leaf of 'a
             | Node of ('a tree) * ('a tree)
```

- Polymorphic Rose Tree with help of list.

```
type 'a roseTree = Node of 'a * (('a roseTree) list)
let rt1 = Node (0,[Node (1,[]); Node (2,[]);
                   Node (5,[Node (3,[])])])
```

Error Values

- Errors occur when unsuitable values are supplied to functions that violate their preconditions.
- Examples : division by zero or head of an empty list.
- Three ways of handling errors.
  - Use `failwith` construct to create a `Failure` error value.
  - Raise an exception to denote error.
  - Use specially encoded value as error, e.g. option type.

Explicit Failure Values

- An error value is created with `failwith` method.
- For example, the denominator of a division is 0, we can explicitly invoke `failwith` method:

```
let div1 (a:int) (b:int) : int =
  if b==0 then failwith "divide by zero error"
  else (a/b)
```

- This error value is denoted by a Failure exception. We associate each error/exception value with the $\perp$ (pronounced as bottom) type that can unify with any type.
- In the above example, the first branch returns an exception value of $\perp$ type. This unifies with second branch of `int` type.

Errors via Exception

- User-defined exceptions can also be used to denote error values.
- They also belong to the $\perp$ type. Example:

```
exception Divide_by_zero;;
let div2 (a:int) (b:int) : int =
  if b==0 then raise Divide_by_zero
  else (a/b)
```

- For errors that can be repaired, we may utilize try-catch exception handling mechanism (see later).

Errors via Option Type

- We may also extend our output types with special values to capture error scenarios
- Option type can serve this purpose with `None` to denote the error scenario.

```
type 'a option = None | Some of 'a
```

- But need to change output type from `int` to `(int option)`.

```
let div3 (a:int) (b:int) : int option =
  if b==0 then None
  else Some (a/b)
```

Let Construct

- Let construct binds local values (including functions)
- Let construct can be nested. Example:

```
let two2 =
  let y = 1 in
  let z = y+0 in
  y+z
```

- Let `rec` construct is used for recursive methods.

```
let rec fact n = if n=0 then 1 else n*(fact (n-1))
```

Conditional

- Conditional `if e1 then e2 else e3` is strict on the test, but lazy on the branches.
- Conditional construct dispatches on boolean `e1` value.
- An example:

```
let max x y =
  if x>y then x else y
```

Conditional

- Conditional is a syntactic sugar for pattern-matching on boolean values.
- Equivalent max function using match construct.

```
let max x y =
  match x>y with
    | true -> x
    | false -> y
```

Pattern-Matching

- Pattern-matching allows deconstruction of data structures.
- An example over list:

```
let rec sum xs =
  match xs with
    | [] -> 0
    | x::xs -> x+(sum xs)
```

- An example over binary tree:

```
let rec count t =
  match t with
    | Leaf _ -> 1
    | Node (x,y) -> (count x)+(count y)
```

Conditional and Access Methods

- Pattern-matching is more concise than using conditional and deconstructor methods.
- An alternative implementation for `sum` method.

```
let rec sum xs =
  if xs==[] then 0
  else (List.hd xs)+sum(List.tl xs)
```

- But we still need definitions for `hd,tl` methods.

Pattern-Matching on Partial Functions

- `List.hd` implemented with incomplete pattern (missing on the `[]` case).

```
let hd xs =
  match xs with
    | x::xs -> x
```

- `List.tl` implemented with an error for `[]` case.

```
let tl xs =
  match xs with
    | [] -> failwith "tl on null error"
    | x::xs -> xs
```

Loop Iteration

- For-loops are useful for imperative coding with side-effects.
- It uses syntax of form: `for v=e1 to e2 do e3 done`
- An example:

```
let foo2 n =
  for i=1 to n do
    print_int i; print_string " "
  done;
  print_newline ()
```

- There is another construct for `while` loop.

Loop Iteration

- Loop iterations are never strictly required. They are just syntactic sugar for tail-recursion.
- An equivalent imperative loop implemented using a tail-recursive `helper` method:

```
let foo3 n =
  let rec helper i =
    if i>n then print_newline ()
    else (print_int i; print_string " "; helper (i+1))
  in helper 1
```

Try-Catch Construct

- Exception handling is similar to Java, except that it is expression-oriented.
- It has the form (`try e1 with p -> e2`), where `e1` and `e2` are of the same type.
- An example where `List.nth` throws an exception when `n` is negative or does not have a corresponding list element.

```
let safe_nth ls n =
  try
    Some (List.nth ls n)
  with _ -> None
```

Try-Catch Construct

- We may also have different handlers for different exceptions. Example:

```
let safe_nth ls n =
  try
    Some (List.nth ls n)
  with
   | Failure _ ->
     (print_endline ("arg "^(string_of_int n)^
         " is too large"); None)
   | Invalid_argument _ ->
     (print_endline "arg is negative"; None)
```

Pure vs Impure Functions

- Pure functions are mathematical functions where the outputs depends solely (or deterministically) on the inputs.
- Impure functions also perform some side-effects, such as updating global variables or performing print operations.
- OCaml Philosophy : use pure functions where possible, but you may resort to impure functions where necessary.
- Keep impure effects localized, where possible. Why?

Anonymous Functions

- We can define a new function without giving it a name.
- Examples:
  - `(fun x -> x * x)` denotes a squaring function.
  - `(fun x -> x + 1)` denotes an increment function.
- We can define, construct and deconstruct functions in a similar fashion as data. How to deconstruct?

Function Type

- Every value has a type, including functions.
- Some examples:
  - `(fun x -> x * x)` has type `int -> int`
  - `count` has type `'a tree -> int`
  - `fst` has type `'a * 'b -> 'a`

Tupled vs Curried Functions

- Curried function takes one argument at a time, by returning a function that takes the next argument.
- Example `addA:int -> int -> int` Same as `addA:int -> (int -> int)`

  ```
  let addA x y = x+y
  let v1 = addA 1 2
  ```

- Tupled function takes all arguments as a tuple/product.
- Example `addB:(int * int) -> int`

  ```
  let addB (x,y) = x+y
  let v2 = addB (1,2)
  ```

Partially Applied Functions

- We can support functions that are partially applied with just some of their arguments.
- Examples:

```
let inc = addA 1
let inc = fun y -> addB (1,y)
```

- Partial applications allow specialized functions to be easily written.

List Append

- `List.append` function joins two lists together, and is also denoted by infix `@` symbol.
- Its type is `'a list -> 'a list -> 'a list`.
- Its implementation:

```
let rec append xs ys =
  match xs with
    | [] -> ys
    | x::xs -> x::(append xs ys)
```

- Guess what is its time-complexity?

List Combine

- `List.combine` can zip two lists into a list of pairs.
- Its type is `'a list -> 'b list -> ('a * 'b) list`.

```
let rec combine xs ys =
  match xs,ys with
    | [],[] -> []
    | x::xs,y::ys -> (x,y)::(combine xs ys)
    | _,_ -> raise (Invalid_argument "combine")
```

- It throws an exception when the two lists have different lengths.

List Reverse

- `List.rev` would reverse the contents of its given list.
- A straight-forward implementation is:

```
let rec rev xs =
  match xs with
    | [] -> []
    | x::xs -> (append (rev xs) [x])
```

- Is this efficient? Guess its time-complexity?

List Reverse (efficient)

- A linear-time implementation that is also tail-recursive is shown below.

```
let rev xs =
  let rec helper xs acc =
    match xs with
      | [] -> acc
      | x::xs -> helper xs (x::acc) in
  helper xs []
```

- It uses a `helper` method with an accumulating parameter to keep elements in the reversed order, before it is returned.

First-Class Functions

- Data can be passed as arguments, returned as results or stored inside other data structures.
- Why not functions?
- If allowed, such first-class versatility greatly increases the expressive power of programming languages.
- Advantage : better reuse and shorter code.
- Disadvantage? : needs training/investment.

Compose

- Larger programs are composed from smaller components.
- Let us define `compose : ('b->'c)->('a->'b)->'a->'c`

```
let compose f g x = f (g x)
```

- Examples:

```
let sumsquare = compose sum squares
let id = compose List.rev List.rev
```

- Well-known unix pipe command is a special case.

```
let unix_pipe f g = compose g f
```

Map

- Lists are pervasive. Let us look at a higher-order method that applies a function argument to every element of its list.
- Scheme : `map f [a;b;c] = [f a; f b; f c]`
- Implementation:

```
let rec map (f:'a->'b) (xs:'a list) : 'b list =
  match xs with
    | [] -> []
    | x::xs -> (f x)::map f xs
```

- This method is from the map-reduce paradigm.
- An example of its use:

```
let list3 = map (fun x -> x*x) [1;2;3;4;5]
```

Reduce (rightwards)

- We may apply a function to reduce a data structure to a value.
- Scheme: `fold_right f [a;b;c] z = f a (f b (f c z))`
- Implementation:

```
let rec fold_right f xs b =
  match xs with
    | [] -> b
    | x::xs -> f x (fold_right f xs b)
```

- An example : `fold_right (*) [1;2;3] 1`
- Type: `('a->'b->'b) -> 'a list -> 'b -> 'b`

Reduce (leftwards)

- We may use a tail-recursive function for reduction too.
- Scheme: `fold_left f z [a;b;c] = f (f (f z a) b) c`
- Implementation:

```
let rec fold_left (f:'b->'a->'b) (b:'b) (xs:'a list) :
  match xs with
    | [] -> b
    | x::xs -> fold_left f (f b x) xs
```

- An example : `fold_left (+) 0 [1;2;3]`
- Type: `('b->'a->'b) -> 'b -> 'a list -> 'b`
- If `f` is associative, can use either `fold_left` or `fold_right`.

Filter

- We can use `List.filter` to select elements from a list that satisfy a given predicate. Its implementation:

```
let rec filter (p:'a->bool) (xs:'a list) : 'a list =
  match xs with
    | [] -> []
    | x::xs -> if p x then x::(filter p xs)
               else filter p xs
```

- Example : `filter (fun x-> x>1) [1;2;3]` returns `[2;3]`.
- Type: `('a->bool) -> 'a list -> 'a list`

Implementation

- Higher-order functions are general and can be used to implement many other functions, including other higher-order functions.
- Implementing `map` using `fold_right`

```
let map2 f xs =
  fold_right (fun x b -> (f x)::b) xs []
```

- Implementing `map` using `fold_left`

```
let map3 f xs = List.rev (
  fold_left (fun b a -> (f a)::b) [] xs)
```

- Implementing `filter` using `fold_right`

```
let filter2 p xs = fold_right (fun x b ->
    if p x then x::b else b) xs []
```

Why Higher-Order Functions?

- Higher-order functions:
  - facilitate code reuse.
  - result in shorter codes.
  - uses fewer functions.
  - easier maintenance.
- Disadvantage?
  - Performance? but compilers/machines getting better.
  - Needs training? which advanced tool doesn't?

Records

- Record is a product type with labels.
- Instead of pair, we can use below for rational number:

```
type ratio = {num:int; denom: int}
```

- Easy to define new records:

```
let r = {num=3; denom=4}
let r2 = {r with denom=5}
```

- Easy to use pattern-matching:

```
let get_denom r =
  match r with
    | {denom=v} -> v
```

Mutable Fields

- Immutable fields are the default.
- Mutable fields have to be explicitly specified.

```
type ratio = {num:int; mutable denom: int}
```

```
let r = {num=3; denom=4}
let _ = r.denom <- 5
```

- As side-effecting programs are harder to debug, do use mutable fields sparingly.

Objects and Classes

- Objects allow data and methods to be placed together.
- Good for mutable objects. Design a counter class:

```
class counter = object
  val mutable c = 0
  method inc = c <- c+1
  method clear = c <- 0
  method getVal = c
end;;
```

- We can create a number of counters.

```
let ctr1 = new counter
let ctr2 = new counter
let _ = for i=1 to 5 do ctr1 # inc done
let _ = for i=1 to 6 do ctr2 # inc done
```

A Single Object

- If we are interested in only a single object, we do not need to define an entire class.
- Example:

```
let mycounter = object
  val mutable c = 0
  method inc = c <- c+1
  method clear = c <- 0
  method getVal = c
end;;
```

Class with Parameter

- Like functions, class can take parameters:
- Example:

```
class counter2 (init:int) = object
  val mutable c = init
  method inc = c <- c+1
  method clear = c <- 0
  method getVal = c
end;;
```

- We can create counters with different initial values.

```
let ctr3 = new counter2 0
let ctr4 = new counter2 1000
```

A Polymorphic Class

- We can support polymorphic classes.

```
class ['a] stack = object
  val mutable stk = []
  method push (x:'a) =
    stk <- x::stk
  method pop  =
    match stk with
      | [] -> ()
      | x::rest -> stk <- rest
end;;
```

Class Inheritance

- We can support class inheritance where methods of superclass are inherited to facilitate code reuse.
- Note an extra printer parameter below.

```
class ['a] stackPr (pr:'a->string) = object
  inherit ['a] stack as super
  method string_of =
    let rec helper xs =
      match xs with
        | [] -> "" | [x] -> pr x
        | x::rs -> (pr x)^";"^(helper rs)
    in "["^(helper stk)^"]"
end;;
```

Modules

- Modules encapture types, values and methods.
- Modules are separate units of compilation.
- OCaml has one of the most advanced module systems.
- Module implementation and signature are separated.
- Module can take other modules as parameters.

StackInt Module

- A module to implement stack of integers.

```
module StackInt =
struct
  type e = int
  let stk = ref []
  let push (x:e) =
    stk := x::!stk
  let pop () =
    match !stk with
      | [] -> ()
      | x::rest -> stk := rest
end;;
```

Polymorphic Module

- A module to support polymorphic stack.

```
module Stack =
struct
  type 'a t = ('a list) ref
  let create () : 'a t = ref []
  let push (x:'a) stk =
    stk := x::!stk
  let pop stk =
    match !stk with
      | [] -> ()
      | x::rest -> stk := rest
end;;
```

Polymorphic Module

- A polymorphic `Stack` implemented with mutable record.

```
module StackR =
struct
  type 'a t = { mutable elems : 'a list}
  let create () : 'a t = {elems = []}
  let push (x:'a) stk =
    stk.elems <- x::stk.elems
  let pop stk =
    match stk.elems with
      | [] -> ()
      | x::rest -> stk.elems <- rest
end;;
```

Abstract Module with Different Implementatins

- We can define the type for an abstract module.

```
module type Stack_ty =
  sig
    type 'a t (* abstract type *)
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> unit
  end;;
```

- This signature can be used by different stack implementations.

```
module Stk1 = (Stack : Stack_ty);;
module Stk2 = (StackR : Stack_ty);;
```

Abstract Module with Different Types

- We can also have a single implementation to be treated as different types by abstract module. For example, the `Float` module can be used to implement different currencies.
- Concrete `Float` implementation:

```
module Float =
struct
  type t = float
  let unit = 1.0
  let plus = (+.)
  let prod = ( *. )
  let getVal (x:t) = x
end;;
```

Abstract Currency Module

- An abstract module for multiple currencies.

```
module type CURRENCY =
sig
  type t
  val unit : t
  val plus : t -> t -> t
  val prod : float -> t -> t
  val getVal : t -> float
end;;
```

Abstract Modules support Distinct Types

- Two modules for different currencies

```
(* two implementations *)
module Euro = (Float : CURRENCY);;
module Yen  = (Float : CURRENCY);;

let mkEuro n = Euro.prod n Euro.unit
let mkYen n = Yen.prod n Yen.unit
```

- Note that `Euro.t` and `Yen.t` are distinct types, even though both are implemented using `float`.

Functors

- Functors are modules that takes modules as parameters. This can support highly reusable modules.
- Consider a module to support comparison.

```
type comparison = Less | Equal | Greater
module type ORDERED_TYPE =
sig
  type t
  val compare: t -> t -> comparison
end;;
```

Set Functor

- We can define an ordered set as a functor that takes `ORDERED_TYPE` as a module parameter.

```
module Set = functor (Elt: ORDERED_TYPE) ->
struct
  type element = Elt.t
  type set = element list
  let empty = []
  let rec add x s = ...
  let rec member x s = ...
end;;
```

- This allow us to support ordered sets of arbitrary types, as long as they have the `compare` operator.

Modules

- An example is `OString` with a `compare` operator.

```
module OString =
struct
  type t = string
  let compare x y =
    if x = y then Equal
    else if x < y then Less
    else Greater
end;;
```

- With it, we can define an ordered set with string elements.

```
module StringSet = Set(OString);;
```

Next Week

- Lab on OCaml (Week 2)
- Introduction of ePL
- Lab on ePL (Week 3)