CS4215—Programming Language Implementation

Chin Wei Ngan

Sunday 8th January, 2017

Chapter 3

An Overview on OCaml

We provide an overview of the OCaml programming language in this chapter. The OCaml language is an expressive programming language that can be used to build small and large software systems, including language translators. The OCaml compiler, interpreter, debugger and parser front-ends are themselves implemented in the OCaml language.

One of the best things about the OCaml language is that it is a *strongly typed* programming language, whereby every expression is expected to have a type that can either be type checked or has its type inferred. This aspect is especially important since many programming errors can be caught as type errors, early on in the software development process. We use the type ascription notation (e:t) to capture the fact that expression e is expected to have type t. As OCaml is a rich programming language with features such as higher-order functions, modules, objects, it has a corresponding rich structure for types to denote the static properties of expressions formed from these language features.

3.1 Data Types

We start this section with some types that can be used for capture commonly used data structures. The OCaml code below highlight some of the basic types available in the language.

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

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

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

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

(* string *)
let myname : string = "Chin␣" ^ "Wei␣Ngan"
```

Note that int type is for finite precision integer that is 31-bits (or 63-bits for the 64-bit processor architecture) wide. If you like to use an arbitrary precision integer, you can make use of the Big_int module. Most of these primitive types and their operations can be found in the Pervasives module that is imported by default. The floating point operations are denoted using special symbols that ends with a dot, such as {+., −., ∗., /.}. A rich set of string operations can be found in a separate String module. Take note that modules' names must always start with an upper-case. There is also a related lexical convention that alphanumeric constructors are to start with an upper-case alphabet, while names of variables must start with a lower-case alphabet. Other related lexical convention can be found here

3.1.1 Tuple Type

One of the easiest way to build more complex data structures is to use the tuple construnction. A simple example is (2,"hello") which pairs up an integer with a string value into a tuple (or product) type. Such an example will have the tuple type (int ∗ string) where the first component is an int type, while the second component is a string type. A simple code fragment is highlighted below where fst , snd are primitives to access the first and second component of the pair type.

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

In general, the tuple (or product) type cna take an arbitrary number of components. For example, the type (t1 ∗ ... ∗ tn) captures a tuple of n possibly distinct types t1,···,tn as its components. groups a number of types together. As an example, a tuple of three types can be formed using (int∗string∗string), but note that this is *isomorphic* to nested pairs (int∗(string∗string)). Let us highlight the equivalence (or isomorphism) with the following example to access the third element of a triple type.

```
(* 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)
```

The triple type is more efficiently implemented that a nested pair. However, we would need to provide new access methods for each component of each of the tuple type. If you are merely interested in getting something to work quickly, and would like to make better reuse of existing codes, it is sufficient to use the pair type to construct any tuple type. Thus, the pair (t1 ∗ t2) type is a fundamental type that can be used to build any tuple type.

3.1.2 Variant Type

Another way to build new data types is to make use of the *variant type* (also known as algebraic data type). In general, a new variant type can be constructed using the syntax:

```
type t =   V1 of t1
         | V2 of t2
         | ..
         | Vn of tn
```

where V1,..,Vn denotes data constructors that must begin with an uppercase alphabet to denote each variation of our variant type, while t1 ,.., tn are the component types of the respective variations. (In case, there isn't a component type, we can merely omit it which is equivalent to using the unit type as its component type. The unit type is a type with only a single value (), and is essentially the type of commands or methods that do not return any value.) Variant types can be used to support enumerated and union types, as illustrated below.

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

The intorstr type allow us to capture a new union type that can either be an integer type (if it is prefixed with the I data constructor) or a string type (if it is prefixed with the S data constructor).

One of the best things about variant type is that it can be used to create recursive data structures, such as a list or trees. Three simple examples are shown below.

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

Note that num shows how we can define natural numbers recursively in Peano-style. The intList type is used to construct a list of integers. The empty list is denoted using Nil. A list with one element of value 3 is captured as Cons 3 Nil. A list of two elements is captured as Cons i1 (Cons i2 Nil).

3.1.3 Polymorphic Type

In order to support the construction of reusable program codes, OCaml can provide types that are polymorphic (or generic) in nature. A polymorphic type is a type with one or more type variables, that are denoted using 'a, 'b, 'c etc. Type variables are universally quantified types, and can be used to denote types of any expression value. A simple example is the identity function below which can be applied to values of any type:

```
let id (x:'a) :'a = x
let v1 = id 3
let v2 = id "hello"
```

This id method is generic in that it can be applied to any type, including integer and string, as highlighted in the above code fragment. As another example, we give a polymorphic list type, as illustrated below:

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

This polymorphic list type can be used to support arbitrary list type whose elements are all of type 'a. Due to its polymorphic nature, we can use it to implement a list of integer using int list, and even a polymorphic list of lists using ('a list) list. Take note that types are written in post-fix form using (tArg t) where t is a type constructor that takes type tArg as its argument, before returning its new type denoted by (tArg t).

As more examples of polymorphic type, let us consider a polymorphic binary tree where elements of type 'a are captured in its leaves.

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

Each node of a binary tree contains only two sub-trees. If we wish to support an arbitrary number of sub-trees on each node, we can define a polymorphic rose tree, as shown below.

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

Note how we use polymorphic list type to implement an arbitrary number of branches, including zero. With this, a leaf node is denoted using just Node(v,[]).

We can also support polymorphic union types. The following is a union of two arbitrary types:

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

An example of its use is (int,string) sum which denotes a union type for integer and string.

3.1.4 Error Values and ⊥ Type

Errors can occur when unsuitable arguments are supplied to functions that violate their preconditions. A simple example of this is an attempt to divide a number by zero, or to try access the head of an empty list. One way to handle error scenarios is to raise an appropriate exception, whenever errors occur. Another way is to encode a special value for error scenarios. Let us look at the implications of handling error scenarios to our type system, through three ways for handling errors.

Firstly, we can use a pre-defined failwith method to generate a Failure exception with a string to identify the kind of error encountered. For example, the denominator of a division is `0`, we can explicitly return a Failure exception, as follows:

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

Exceptions (and error values in general) can be denoted by ⊥ (pronounced as bottom) type that can unify with any type. In the above example, the first branch returns an error value of ⊥ type. This unifies with second branch of `int` type. Hence , we still have type consistency.

A second way to handle error scenarios to use a user-definable exception. We can declare an exception prior to invoking it via a raise construct, as illustrated below:

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

Apart from type consistency, it is useful to note that errors may be repaired using try-catch exception handling code (see later in Sec 3.2.5).

A third option to handling error is to use specially encoded values for errors. A good polymorphic type to handle this is the option type, shown below, where None can be used to denote some error scenario.

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

If we were to use this option type, we would have to extend the output type of our division method from int to int option.

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

3.2 Expressions

OCaml is an expression-oriented language with a rich set of general programming constructs. Programs are essentially composed of types and values, with definition of values spanning from primitive data types to more complex user-defined types, functions, objects and even the module system itself. This section will describe a number of control constructs that are used to express computation, and also to return the computation's result.

3.2.1 Let

Let us start off with the **let** construct that can be used to bind values (including functions) that we have created. At the outermost level, the values bound by **let** constructs are globally visible within each module. However, we may also bind let construct to values that are visible only locally, as illustrated by the following example:

```
let v2 =
  let y = 1 in
  let z = y+0 in
  y+z
```

Though two2 is visible in the entire module (unless shadowed), the values of y,z are only visible locally in their respective scope. Furthermore, if we redefine a variable, its latest occurrence will shadow prior occurrences with the same name. As an example:

```
let v3 =
  let y = 1 in
  let z = y+(let y=2 in y*y) in
  y+z
```

In the third line, the first occurrence of y is bound to 1, while the last two occurrences of y are bound to 2, since the second let construct on y has managed to shadow the outer **let** construct on y.

As we have already seen earlier, **let** construct can also be used to define functions. However, if we wish to define recursive functions, we have to insert an explicit **rec** keyword to indicate that the given definition is inductive. An example is shown below:

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

3.2.2 Conditional

Conditional is another basic programming construct. While those found in imperative languages, such as Java or C, are statement-based; the conditional construct for OCaml is expression-oriented. A conditional construct of the form **if** e1 **then** e2 **else** e3 is expected to return the values of either e2 or e3, depending on the value of the test e1. Each condition is strictly evaluated on its test (namely e1), but is lazy on the two branches. That is, only the branch that is required will be evaluated for the conditional, depending on the value of its test.

If conditional had not been lazy on its branches, a simple evaluation of the fact code below would immediately go into an infinite loop due to the recursive nature of fact. Being lazy, allow us to reach the base case in case the test n=0 ha been computed to be true.

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

Another example of conditional expression is shown next:

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

Conditional is a syntactic sugar for pattern-matching on boolean values. Each conditional of the form **if** e1 **then** e2 **else** e3 is simply translated to the following more general **match** construct, that will be described next.

```
  match e1 with
    | true  -> e2
    | false -> e3
```

Here, the expression e1 is evaluated to a boolean value. If its value matches with true, the expression e2 (in the first branch) will be evaluated. However, if its value matches with false, the e3 expression (in the second branch) will be evaluated instead.

3.2.3 Match

As with all data structures, we need a way for building them and also another way for taking them apart. The **match** construct provide us a way to look inside the data structure to obtain the components of the data structure. As a simple example, let us first look at how pattern matching can be used to retrieve a component from the tuple type. Below is how the snd function would be implemented.

```
let snd xs =
  match xs with (x,y) -> y
```

To deal with variant types, the **match** construct allows multiple branches to be supported for each of the variations. As an example, let us look how a recursive method over the list data type is being implemented:

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

There is a pattern [] for the empty case, and a pattern x::xs for the node case of the list data structure. Both these cases came from the variant type declaration for list, and we allow their respective codes to be executed, depending on its current input value. A similar structure can also be seen for binary tree, where we have either a Leaf pattern, or a Node pattern, as highlighted below.

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

The general form of pattern-matching is illustrated by the following construct:

```
  match e with
    | p1 -> e1
        :
    | pn -> en
```

OCaml will first evaluate expression e, and then match its values with patterns p1 ,.., pn. This matching stops with the first pattern pi that it matches, with its corresponding branch ei evaluated and returned. In general, we expect the patterns to be exhaustive. However, if it is not exhaustive, a warning will be given at compile-time, but this could lead to an exception if the matching should also fail at run-time. An example of the match construct that is not exhaustive is the following:

```
  match e with
    | [y]     -> e1
    | y::ys -> e2
```

This matching would fail with an exception if expression e evaluates to [] which is a pattern not covered by this code fragment. If we can show (or prove) that e never evaluates to [] , this code will be completely safe.

3.2.4 Loops

For- and while-loops are iterative constructs from the imperative programming paradigm, and are useful for writing code whose primary purpose is to produce some side-effects. OCaml supports both kinds of loops. For-loop is implemented with the following syntax: **for** v=e1 **to** e2 **do** e3 **done**. A simple example is shown below which attempts to print n numbers in a single line:

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

There is also a corresponding syntax, namely **while** e1 **do** e2 **done**, for while-loops, but we omit it an example here since it will require us to use a mutable integer whose values may be updated by the loop iteration. This topic on mutable values will only be visited later in Sec **??**.

Nevertheless, both for-loop and while-loop are <u>not</u> strictly required for the OCaml programming language. They are just syntactic sugar for tail-recursion.

We can easily implement each imperative loop by an equivalent tail-recursive method, as shown below with a tail-recursive helper method, for our earlier example.

```
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
```

3.2.5 Try-Catch

The try-catch is a fundamental construct for handling exceptions in OCaml. This exception handling is similar to that used in Java, except that it is expression-oriented. It has the form (**try** e **with** p1 −> e1 | .. | pn −> en) where e,e1 ,.., are of the same unifiable type. This construct will first evaluate e. If this executes successfully without any exception (or error), its value will be directly returned by the try-catch construct. However, if an exception is encountered, its will be matched against the (exception) patterns p1 ,.., pn. This thrown exception will be caught by the first matched pattern, say pi, with its corresponding handler ei returned as a result from this construct. If none of the exception patterns are matched, the earlier exception escapes the current try-catch construct and may be caught by some outer try-catch block. As an example of its use, let us consider the List.nth method which throws an exception whenever its parameter n is negative or if its list has a length that is not less than n. We can convert this method into a safe version without exception, as shown below:

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

Here, we use an option type to encode the result of the List.nth method. The constructor Some .. will denote a successful execution, while the constructor None is used to denote an erroneous scenario signalled by an exception. Occasionally, a try-catch construct with multiple handlers may be used instead. An example is the following where we also print a suitable error message whenever List.nth raises an exception:

```
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)
```

3.3 Functions

Functions are first-class values in OCaml and form the basis of reusable program code. We have already seen how to construct function through the **let** construct, and have already seen how to use them through function application (or call). However, an important concept to recognize is the distinction between pure and impure functions.

3.3.1 Pure vs Impure Functions

Pure functions are mathematical functions where the outputs depends solely (or deterministically) on the inputs. In contrast, **impure functions** are functions that also perform some side-effects, such as updating global variables or performing some print operations. In general, pure functions are much easier to reuse than impure functions, since you will have to carefully consider execution history when tracing the latter. For example, think of a recursive factorial function that computes its factorial by updating a global variable, during recursion! To understand, such a impure method, you would have to carefully consider execution history. In contrast, a recursive method to compute factorial without any side-effects is directly related to the mathematical definition of the same concept. The OCaml Philosophy is to encourage you to use pure functions where possible, but you may resort to impure functions, with side-effects, as a last resort. You should not be too worried about machine efficiency since this is considerably cheaper than cost of your time that could be incurred from trying to understand and debug problematic methods with side-effects.

3.3.2 Anonymous Functions

As methods are first-class entity, you should be able to define a method without having to give a name. This is only possible for non-recursive methods but it is useful since it save you effort in thinking of a suitable name. We use the notation $(funv_1 \cdots v_n - > e)$ to denote an anonymous function with n parameters. Two simple examples are shown below:

- `(fun x -> x * x)` denotes a squaring function.
- `(fun x -> x + 1)` denotes an increment function.

3.3.3 Types for Functions

Every value has a type, and this includes functions. We use the notation t1 $->$ t2 to denote a function that takes an argument of type t1, before returning a result of type t2. Some simple examples are illustrated below:

- `(fun x -> x * x)` has type `int -> int`
- `count` has type `'a tree -> int`
- `fst` has type `'a * 'b -> 'a`

3.3.4 Tupled vs Curried Functions

There are two ways to write a function which takes n parameters. Let us consider the use of an anonymous function. The first way is to use a tuple of the n parameters, as follows: (**fun** (p1 ,.., pn) −> e). This is referred to as a *tupled function*. The second way is to n parameters in consecutive order, as follows: (**fun** p1 .. pn −> e). This is referred to as a *curried* function that takes one argument at a time, and then returning a function that takes the next argument and so on. As a matter of fact, a curried function of the above form is equivalent to (**fun** p1 −> (**fun** p2 −> .. (**fun** pn −> e)..)), which clearly shows it taking one argument at a time and returning a function that takes the next argument.

Let us look at another curried function that is defined with a **let** construct, as shown below.

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

This function will have the type addA:int −> int −> int which shows that it takes an int type and then returning a function from int to int. If we look at the way curried functions are applied, say addA 1 2, you can also express it equivalently as ((addA 1) 2) which applies the argument one at a time.

The corresponding function in tupled form is illustrated below:

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

It has the type addB:(int ∗ int) −> int, which shows that it takes a tuple of arguments simultaneously. Tupled functions and curried functions are isomorphic to each other. Given any function, we can always define an equivalent one in the other form. So, what would be your preference? In OCaml, there is a general preference to using curried functions, seems your code will be more concise with fewer brackets. Furthermore, the use of a tuple construction often adds an extra indirection to the memory layout.

3.3.5 Partially Applied Functions

We can support functions that are partially applied with just some of their arguments. Partial applications allow specialized functions to be easily written. Two examples of it using curried and tupled functions are shown below:

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

In the case of tupled function, we have to use an anonymous function to make it take one parameter first. We may also use anonymous function also for curried functions in case we wish to take the parameters in a different order.

3.3.6 List-Based Functions

As list is pervasively used in OCaml programs, we will now look at how a couple of their functions are implemented. A comprehensive OCaml list library can be found here Let us first look at a function that would join two lists together, called List.append. For convenience, this function is also denoted by infix `@` symbol. Its type is 'a list -> 'a list -> 'a list. How about its implementation? We can use pattern-matching and recursion to implement it as:

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

Note that this is a pure function, as list are by default immutable. How about the time-complexity? How many recursive calls do we make? If you study the code carefully, you will note that the number of recursive calls made is proportional to the length of the first list, and as each call takes finite step, we will have a time-complexity of O(length(xs)).

Let us look at the next list function, called List.combine that would combine two lists of values into a list of pairs. An example of its use is combine [1;2] ['a';'c'] would be be zipped into a list of pairs [(1,'a');(2,'c')]. The polymorphic type of the method is 'a list -> 'b list -> ('a * 'b) list, and we can implement it using:

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

The two lists are expected to be the same length. If we supply lists of different lengths, an Invalid_argument exception will be thrown.

3.3.7 Is Recursion Efficient?

One of the myth about using recursion is that it is less efficiently implemented than iterative loop. While this may be naively true, since loops can be executed within constant stack space, while recursion would typically require the use of stacks for its recursive method calls; the reality may not be further from the truth since tail-recursive methods are essentially equivalent to iterative loops. And, compilers could easily optimize tail-recursion to run in constant stack space. However, one thing is for sure. Recursive methods are much more expressive than iterative loops, since you can mimick every loop using tail-recursion but not all recursive methods can be directly converted to iterative loop.

Let us look at the expressivity of recursion by looking at different ways of implementing some methods, starting with a simple method to reverse a list of elements, so that the elements would be accessible in the opposite order. A simple implementation of List.rev is shown below:

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

We use the append method to add each head element to the end of a recursively reversed list. Given rev [1;2;3] , we would obtain [3;2;1] , but this is achieved using a series of append calls, namely append (append (append [] [3]) [2]) [1] . What is the time-complexity, especially when append has a complexity that is linear to the first list? If we have a list of length n, the innermost takes 1 step, the next call takes 2 steps while the outermost call takes n steps. If we sum these steps using 1+2+..+n, we would actually obtain $O(n^2)$ complexity.

Is there a more efficient way to reverse list? Yes, what we could do is to keep an accumulating parameter that keeps the reversed list so far, as shown by the following implementation.

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

The parameter acc of helper is given the initial [] value. With its recursive call, the accumulating parameter grows larger. For example, if our input list was [1;2;3] , the parameter encountered in successive recursive call would be [1] , then [2;1] , then [3;2;1] before the now reversed list is returned as helper's result. As there are n calls to helper, and each call takes constant time, our time complexity for this method would be a much better $O(n)$ complexity.

An often cited example for inefficient recursive method os the following naive fibonacci definition.

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

This method has exponential time-complexity, and you will probably find it hard to execute on calls with even medium-sized arguments. The main cause of algorithmic inefficiency is the presence of many repeated calls, rather than the problem of recursion per se.

You can implement a more efficient fibonacci function by the following definition where the helper method is returning a pair of fibonacci values.

```
let fib2 n =
  let rec helper n =
    if n=0 then (0,1)
    else let (a,b)=helper (n-1) in (b,a+b) in
  fst(helper n)
```

That is each recursive method helper n that returns two values, namely (fib n, fib (n+1)). The redundant calls are eliminated by reusing a pair of fibonacci valuesto compute the next pair of two fibonacci values. This method is actually linear-time, but it is not tail-recursive.

We shall next show how a tail-recursive version of fibonacci method is being implemented. Let us add two extra parameters to our helper method, as shown below.

```
let fib3 n =
  let rec helper n a b =
    if n=0 then a
    else helper (n-1) b (a+b) in
  helper n 0 1
```

These two values are initially 0 and 1, the smallest fibonacci values, but will successively be computed to reflect the next pair of fibonacci values. The main difference with tupled version is that we are computing the fibonacci values in a bottom-up version. This tail-recursive version is actually equivalent to most loop-based implementation of fibonacci. The take-away from these examples is that you can write efficient code with recursion, but you will need to understand how efficient algorithmic patterns are being designed. As a small poser, you can actually also implement a logarithmic-time version of fibonacci. Can you think about how that can be done?

3.4 Higher-Order Functions

Contents

3.4.1 Functions as First-Class Values

▯containslstlisting]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.