CS 321 Programming Languages

Intro to OCaml – Part II

Baris Aktemur

Özyeğin University

Last update made on Tuesday 24th October, 2017 at 09:22.

Much of the contents here are taken from Elsa Gunter and Sam Kamin's OCaml notes available at
`http://courses.engr.illinois.edu/cs421`

Values are fixed at function declaration time.

```
# let x = 12;;
val x : int = 12

# let plusX = fun y -> x + y;;
val plusX : int -> int = <fun>

# plusX 20;;
- : int = 32

(* New declaration, not an update *)
# let x = 40;;
val x : int = 40

# plusX 20;;
- : int = 32
```

Closure

So how does this work?

A function value freezes the environment at the time of its declaration, and uses that environment whenever the function is applied.

So a function value stores the function parameter, function body, and the environment in effect when the function was defined.

This function value is called a **closure**.

Values are fixed at function declaration time.

```
# let x = 12;;
val x : int = 12
```

Env_1: $[x \mapsto 12]$

```
# let plusX = fun y -> x + y;;
val plusX : int -> int = <fun>
```

Env_2: $[plusX \mapsto \langle y \rightarrow x+y, Env_1 \rangle, x \mapsto 12]$

```
# plusX 20;;
- : int = 32
```

The function body uses Env_1 when evaluating its body.

```
# let x = 40;;
val x : int = 40
```

Env_3: $[x \mapsto 40, plusX \mapsto \langle y \rightarrow x+y, Env_1 \rangle, x \mapsto 12]$

```
# plusX 20;;
- : int = 32
```

The function body still uses Env_1 when evaluating its body.

Evaluation of function application with Static Scoping

Given an application expression $e_1 e_2$ in an environment ρ:

- Evaluate e_1 in ρ, obtain a closure $\langle y \rightarrow e_b,\ \rho_f \rangle$.
- Evaluate e_2 in ρ, obtain a value v.
- Bind v to y to extend ρ_f. That is, obtain $\rho_b = [y \mapsto v] + \rho_f$.
- Evaluate e_b in environment ρ_b.

Static scoping example

Evaluate `plusX 20`, assuming the environment Env_3: $[\text{x} \mapsto 40, \text{plusX} \mapsto \langle \text{y}{\rightarrow}\text{x+y}, \text{Env}_1 \rangle, \text{x} \mapsto 12]$

- Evaluate `plusX` in Env_3: gives $\langle \text{y}{\rightarrow}\text{x+y}, \text{Env}_1 \rangle$.
- Evaluate 20 in Env_3: trivially gives 20.
- Bind 20 to y to extend Env_1: gives $\rho_b = [\text{y} \mapsto 20, \text{x} \mapsto 12]$
- Evaluate $x + y$ in environment ρ_b: gives $12 + 20 = 32$.

Function application is tight

Function application has the highest precedence in the syntax.

```
# let timesTwo x = 2 * x;;
val timesTwo : int -> int
# let timesTwo = fun x -> 2 * x;;
val timesTwo : int -> int

# timesTwo 4 + 5;;
- : int = 13
# timesTwo (4 + 5);;
- : int = 18
```


Functions

```
(* A function with two parameters *)
# let max = fun n -> fun m -> if n - m > 0 then n else m;;
val max : int -> int -> int = <fun>
# let max n m = if n - m > 0 then n else m;;
val max : int -> int -> int = <fun>

(* Applying the function on two arguments *)
# max 34 45;;
- : int = 45

# max 67 23;;
- : int = 67
```

Scope and let-in

In what region of the program can a particular name be used?

- The scope of a parameter is the function body.
- The scope of a name bound using a **top-level** `let` declaration is the rest of the program.
- For local name bindings, you can use the `let` x `=` e_1 `in` e_2 expression. Here, the name x is available only inside e_2.

Scope and environment are very closely related!

```
# let num = 5
  in num * num;;
- : int = 25
# num;;

Error: Unbound value num
```


Scope

```
# let k =
    let p = 4
    in p * p;;

val k : int = 16

# k;;
- : int = 16
# p;;

Error: Unbound value p
```

Scope

```
# let timesTwo m = m *. 2.0;;
val timesTwo : float -> float = <fun>

# m;; (* m is not in this scope *)

Error: Unbound value m
```


Functions on tuples

```
# let plus pair =
    let (a,b) = pair
    in a + b;;
val plus : int * int -> int = <fun>

(* A shorter definition doing the same thing *)
# let plus (a,b) = a + b;;
val plus : int * int -> int

# plus (3, 4);;
- : int = 7
# plus 3 4;;

Error: This function has type int * int -> int
       It is applied to too many arguments; maybe you forgot a `;'.
```

Functions on tuples

```
# let firstOf pair =
    let (a,b) = pair
    in a;;
val firstOf : 'a * 'b -> 'a = <fun>
(* A polymorphic type that contains type variables.
   Read α * β → α.
   This function operates on pairs of any type.
*)

# firstOf (9, 5.4);;
- : int = 9  (* What's α, β in this case? *)
# firstOf (3.14, "abc");;
- : float = 3.14  (* What's α, β in this case? *)
```


Functions on tuples

```
# let secondOf pair =
    let (a,b) = pair
    in b;;
val secondOf : 'a * 'b -> 'b = <fun>
(* Again, the type is polymorphic *)

# secondOf (3, 5.4);;
- : float = 5.4  (* What's α, β *)
# secondOf (3, "abc");;
- : string = "abc"  (* What's α, β *)
```

Functions on tuples

`first` and `second` are already defined in the basic library.

```
# snd;;
- : 'a * 'b -> 'b = <fun>
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst (6, "asd");;
- : int = 6
# snd (6, "asd");;
- : string = "asd"
```


Exercise

Write a function that takes a pair and returns its reverse.

```
# let revpair p = ???;;
val revpair : 'a * 'b -> 'b * 'a = <fun>

# revpair (3,4);;
- : int * int = (4, 3)

# revpair ('a',4.5);;
- : float * char = (4.5, 'a')
```

Functions on tuples

```
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
# fst(double "hi");;
- : string = "hi"
# fst double "hi";; (* Function application is left-associative *)
                    (* This is parsed as: *) (fst double) "hi"

Error: This expression has type 'a -> 'a * 'a
       but an expression was expected of type ('b -> 'c) * 'd
```


Exercise

Write the types of the following functions. Use type variables when necessary.

```
# let mktriple p = (fst p, snd p, 3)

# let incr p x = (snd p + x, fst p + x)

# let crossAdd p1 p2 = fst p1 + snd p2

# let cross p1 p2 = (snd p1, snd p2)
```

Partial application of functions

```
# let max = fun n -> fun m -> if n - m > 0 then n else m;;
val max : int -> int -> int = <fun>

(* Apply max on one argument only *)
# max 10;;
- : int -> int = <fun>
(* Result is a function that takes an int argument *)

# let maxTen = max 10;;
val maxTen : int -> int = <fun>

# maxTen 15;;
- : int = 15
# maxTen 5;;
- : int = 10
```


Curried vs. Uncurried

```
# let addThree x y z = x + y + z;;
val addThree : int -> int -> int -> int = <fun>

# addThree 3 4 5;;
- : int = 12
```

How does this differ from the following?

```
# let addTriple (x, y, z) = x + y + z;;
val addTriple : int * int * int -> int = <fun>

# addTriple (3, 4, 5);;
- : int = 12
```

`addThree`: curried
`addTriple`: uncurried

Curried vs. Uncurried

```
# addThree (3, 4, 5);;
Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type int
# addTriple 3 4 5;;
Error: This function has type int * int * int -> int
       It is applied to too many arguments; maybe you forgot a `;'.
```


Curried vs. Uncurried

```
# addThree 3 4;;
- : int -> int = <fun>
# addTriple (3, 4);;
Error: This expression has type 'a * 'b
       but an expression was expected of type int * int * int
```

Nameless/Anonymous Functions

```
# (fun n -> n * 2) 5;;
- : int = 10
# let funPair = ((fun n -> n * 2), (fun y -> y + 10));;
val funPair : (int -> int) * (int -> int) =
  (<fun>, <fun>)
# let (f,g) = funPair;;
val g : int -> int = <fun>
val f : int -> int = <fun>
# f 21;;
- : int = 42
# g 21;;
- : int = 31
# funPair;;
- : (int -> int) * (int -> int) = (<fun>, <fun>)
# (fun n -> n * 3);;
- : int -> int = <fun>
```


Functions as arguments

```
# let thrice f = f(f(f(10)));;
val thrice : (int -> int) -> int
(* Note the parentheses in the type.
   Function types are right-associative.
   int -> int -> int would be parsed as
   int -> (int -> int)
*)

# thrice (fun n -> n + 2);;
- : int = 16
# thrice (fun n -> n * 2);;
- : int = 80
# thrice plusTwo;;
- : int = 16
```

Nameless Functions

```
thrice (fun n -> n + 2)
```

is the same as

```
let plusTwo n = n + 2
in thrice plusTwo
```

which is, in fact, nothing but

```
let plusTwo = fun n -> n + 2
in thrice plusTwo
```


Functions as arguments

```
# let thrice f x = f(f(f(x)));;
val thrice : ('a -> 'a) -> 'a -> 'a
(* A polymorphic type. *)

# thrice plusTwo 30;;
- : int = 36
(* What's 'a in this case? *)

# thrice (fun s -> "Hi! "^s) "there";;
- : string = "Hi! Hi! Hi! there"
(* What's 'a in this case? *)
```

Functions returning functions

```
# let foo b =
    if b then (fun x -> x * 2)
    else plusTwo;;
val foo : bool -> int -> int = <fun>

# let g = foo (3>2);;
val g : int -> int = <fun>
# g 10;;
- : int = 20
# let h = foo (3<2);;
val h : int -> int = <fun>
# h 10;;
- : int = 12
```


Functions everywhere

"Primitive" operations are in fact functions, too.

```
# (+);;
- : int -> int -> int = <fun>
# (-);;
- : int -> int -> int = <fun>
# (/);;
- : int -> int -> int = <fun>
# ( * );;
- : int -> int -> int = <fun>
```

Functions

```
# let apply f x y = f x y;;
val apply : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# apply (+) 4 6;;
- : int = 10

# let add = apply (+);;
val add : int -> int -> int = <fun>

# add 3 4;;
- : int = 7
```


Functions

```
# let compose f g x = f(g(x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let c1 = compose (fun n -> n * 2) plusTwo;;
val c1 : (int -> int)
# c1 10;;
- : int = 24

# let c2 = compose plusTwo (fun n -> n * 2);;
val c2 : (int -> int)
# c2 10;;
- : int = 22
# let thrice f = compose f (compose f f);;
val thrice : ('a -> 'a) -> ('a -> 'a)
(* Is this the only way? *)
```

Fact

Functions are first-class values in OCaml.

Recursive functions

```
# let rec factorial n =
    if n = 0 then 1
    else n * factorial (n - 1);;
(* rec is needed for recursive declarations *)
val factorial : int -> int = <fun>

# factorial 5;;
- : int = 120
```


Exercise

```
# let rec power x n = ???;;

val power : int -> int -> int = <fun>

# power 3 4;;
- : int = 81
```

Exercise

```
# let rec fib n = ???;; (* Exercise *)
val fib : int -> int = <fun>

# fib 0;;
- : int = 1
# fib 1;;
- : int = 1
# fib 2;;
- : int = 2
# fib 3;;
- : int = 3
# fib 4;;
- : int = 5
# fib 5;;
- : int = 8
# fib 6;;
- : int = 13
```


Mutual recursion

```
# let rec even n = if n=0 then true else odd (n-1)
  and odd n = if n=0 then false else even (n-1);;

val even : int -> bool = <fun>
val odd : int -> bool = <fun>
```

Exercise: Binomial numbers

Define a function named `binom` to compute $\binom{n}{m}$

```
# let rec binom n m = ???

val binom : int -> int -> int
# binom 4 2;;
- : int = 6
# binom 6 2;;
- : int = 15
# binom 6 3;;
- : int = 20
# binom 6 4;;
- : int = 15
```

Hint: Take a look at the Pascal triangle.

```
  m: 0  1  2  3  4  5  6
    --------------------
n:
0|   1
1|   1  1
2|   1  2  1
3|   1  3  3  1
4|   1  4  6  4  1
5|   1  5  10 10 5  1
6|   1  6  15 20 15 6  1
```