Introduction to OCaml

Jed Liu

Department of Computer Science
Cornell University

CS 6110 Lecture
26 January 2009

Based on CS 3110 course notes
and an SML tutorial by Mike George

Installing OCaml

- Linux:
  ```
  yum install ocaml
  apt-get install ocaml
  emerge dev-lang/ocaml
  ```
- Windows:
  http://caml.inria.fr/ocaml/release.en.html
  Get the **Microsoft-based native Win32 port**

- OCaml toplevel system demo

Declaring Variables

```
let sixEleven = 611

(* A local variable declaration *)
let fortyTwo =
  let six = 6
  and nine = 7
  in six * nine
```

Base Types

```
let x : int = -7
let y : char = 'a'
let z : string = "moo"
let w : float = 3.14159
let v : bool = true
```

- OCaml has type inference
- Type declarations are optional in many places
- But having them makes it much easier to debug type errors!

Tuples and Datatypes

```
(* Tuples (a.k.a. product types) *)
let t1 : int * int = (3, 5)
let t2 : string * bool * char = ("moo", true, 'q')
let t3 : unit = () (* The empty tuple *)

(* A simple datatype (like enum or union) *)
type suit = Spades | Diamonds | Hearts | Clubs
let c : suit = Clubs
```

More Datatypes

```
(* Datatype constructors can carry values *)
(* and be recursive (and look like CFGs)  *)
type var = string
type exp = Var of var
         | Lam of var * exp
         | App of exp * exp

let id : exp = Lam ("x", Var "x")
let w : exp =
  App (Lam ("x", App (Var "x", Var "x")),
       Lam ("x", App (Var "x", Var "x")))
```

- Can build up tuples and datatypes...
- How to break them down and actually use them?

Pattern Matching

```
let t1 : int * int = ...

(* Binds two variables at once *)
let (a, b) = t1

(* Use _ for "don't care" *)
let (_, b) = t1

(* Can match constants too *)
let (a, 5) = t1
```

- OCaml warns about non-exhaustive patterns

More Pattern Matching

```
let suitname : string =
  match c with
    Spades -> "spades" | Diamonds -> "diamonds"
  | Hearts -> "hearts" | Clubs -> "clubs"

(* Base types are just special datatypes *)
(* and can also be pattern-matched       *)
let b : bool = ...
let x : int =
  match b with
    true -> 1
  | false -> 0
```

More Pattern Matching

```
let suitname : string =
  match c with
    Spades -> "spades" | Diamonds -> "diamonds"
  | Hearts -> "hearts" | Clubs -> "clubs"

(* Base types are just special datatypes *)
(* and can also be pattern-matched       *)
let b : bool = ...
let x : int =
  match b with
    true -> 1
  | false -> 0

(* Says the same thing and is better style: *)
let x : int = if b then 1 else 0
```

A Warning about Pattern Matching

```
(* What does this evaluate to? *)
let pair = (42, 611)
let x = 611
match pair with
  (x, 611) -> 0
| (42,  x) -> 1
| _        -> 2
```

A Warning about Pattern Matching

```
(* What does this evaluate to? *)
let pair = (42, 611)
let x = 611
match pair with
  (x, 611) -> 0
| (42,  x) -> 1
| _        -> 2
```

- Patterns can refer to datatype constructors and constants
- But cannot refer to pre-existing variables
- They can only *declare* new variables

Functions

```
(* A variable with a function value *)
let square : int -> int =
  fun (x:int) -> x * x (* anonymous fun! *)

(* Same thing, more succinct *)
let square (x:int) : int = x * x
```

Recursive Functions

```
let rec fact (x:int) : int =
  if x = 0 then 1
  else x * (fact (x-1))

(* Mutually recursive functions *)
let rec isOdd (x:int) : bool =
  x != 0 && isEven (x-1)
and isEven (x:int) : bool =
  x = 0 || isOdd (x-1)
```

More Functions

```
(* How many arguments does this take? *)
let rec gcd (a, b) : int =
  if b = 0 then a
  else gcd (b, a mod b)
```

More Functions

```ocaml
(* How many arguments does this take? *)
let rec gcd (a, b) : int =
  if b = 0 then a
  else gcd (b, a mod b)

(* More explicitly: *)
let rec gcd (p : int * int) : int =
  match p with (a, b) ->
    if b = 0 then a
    else gcd (b, a mod b)
```

Curried Functions

```
let rec gcd (a, b) : int =
  if b = 0 then a
  else gcd (b, a mod b)

(* Preferred style: *)
let rec gcd' (a:int) (b:int) : int =
  if b = 0 then a
  else gcd' b (a mod b)
(* Has type int -> int -> int *)

(* More explicitly: *)
let rec gcd' (a:int) : int -> int =
  fun (b:int) ->
    if b = 0 then a
    else gcd' b (a mod b)
```

A Minor Tangent...

- We have

```
gcd : int  *  int -> int
gcd': int -> (int -> int)
```

- Through currying and uncurrying, these types are somehow "equivalent"
- Squint hard and you might see logical propositions...

$$A \wedge B \implies C$$
$$A \implies (B \implies C)$$

...which are logically equivalent!

Local Declarations (including local functions)

```
(* Newton's method of approximation *)
let rec newton f guess : float =
  let goodEnough = abs_float (f guess) < 0.0001
  in
    if goodEnough then guess
    else
      let
        f' x = (f x -. f (x -. 0.0001)) /. 0.0001
      in
        let newGuess =
            guess -. (f guess) /. (f' guess)
        in newton f newGuess
```

Polymorphism

```
(* What is this function’s type? *)
let id x = x
```

Polymorphism

```
(* What is this function’s type? *)
let id x = x

(* More explicitly *)
let id (x : ’a) : ’a = x

(* A polymorphic datatype *)
type ’a lst =
    Empty
  | Cons of (’a * ’a lst)

let rec map (f:’a -> ’b) (l:’a lst) : ’b lst =
  match l with
    Empty -> Empty
  | Cons (hd, tl) -> Cons (f hd, map f tl)
```

Lists

- OCaml has lists built-in
  - `[]` is the empty list
  - `::` is the cons operator
  - `@` is the append operator
  - `[1; 2; 3]` is a three-element list (note the semicolons)

```
let rec reverse (l : 'a list) : 'a list =
  match l with
    [] -> []
  | hd :: tl -> (reverse tl) @ [hd]
```

- A fancy list pattern:

  `[a; (42, [611]); (b, c::d)]`

Putting It All Together

- Demo: `#use "fv.ml"`

Summary

- Types, tuples, datatypes
- Pattern matching
- Higher-order functions, anonymous functions, currying
- Polymorphism

Resources

- CS 3110 notes

  http://www.cs.cornell.edu/courses/cs3110/2008fa/
- Objective CAML Tutorial

  http://www.ocaml-tutorial.org/
- SML vs. OCaml

  http://www.mpi-sws.org/ rossberg/sml-vs-ocaml.html
- OCaml manual

  http://caml.inria.fr/pub/docs/manual-ocaml/