OCaml Tutorial

Abram Hindle

Kitchener Waterloo Perl Monger

http://kw.pm.org

abez@abez.ca

October 19, 2006

OCaml

- Functional Language
- Multiple paradigms: Imperative, Functional, Object Oriented
- Heavy Generic support
- Interpreted or Byte code compiled or native
- Free as in Freedom (LGPL)
- Type Inferenced
- Cross Platform

Why use OCaml?

- Fast, according the programming language shootouts OCaml is often better speed than even C++
- Statically Typed. Everything except marshalling is type safe. You can’t break type safety without obvious hacks.
- Numerical Computation
- Performance oriented applications: statistics, mathematics, audio, multimedia
- Reasonable external library support
- Easy to integrate with existing C and C++ libraries.
- Threads (native or interpreted)

OCaml Lists

-
  ```
  (* construct a list *)
  let l = 1 :: [] in
  let l = [ 1 ; 2; 3 ] in
  let l = [ 1 ] @ [ 2 ; 3 ] in
  let l = 1 :: 2 :: 3 :: [] in
  let fst::rest = l in
  let fst::snd::third::rest = l in
  ```

OCaml List Operations

-
  ```
  let third = List.nth 2 [1 ;2 ;3] in
  let squares = List.map (fun x -> x * x) [ 1 ; 2 ; 3] in
  let sum = List.fold_left (+) 0 [ 1 ; 2 ;  3] in
  let product = List.fold_left ( * ) 1 [ 1 ; 2 ;  3] in
  let gt4 = List.map ( fun x -> (x, x > 4) ) [ 1 ; 2 ; 3 ; 4
  let gt4 = List.filter (fun x -> x > 4) [2 ; 4 ; 6; 8] in
  let tf  = List.exists (fun x -> 10 = x) [ 1 ; 2 ; 10] in
  ```

OCaml Array Operations

-
  ```
  let third = Array.nth 2 [| 1 ;2 ;3 |] in
  let squares = Array.map (fun x -> x * x) [| 1 ; 2 ; 3 |] i
  let sum = Array.fold_left (+) 0 [| 1 ; 2 ;  3 |] in
  let product = Array.fold_left ( * ) 1 [| 1 ; 2 ;  3 |] in
  let gt4 = Array.map ( fun x -> (x, x > 4) ) [| 1 ; 2 ; 3 ;
  let gt4 = Array.filter (fun x -> x > 4) [| 2 ; 4 ; 6; 8 |]
  let tf  = Array.exists (fun x -> 10 = x) [| 1 ; 2 ; 10 |] i
  ```

OCaml Functions

-
  ```
  let f x = x in
  let f (a,b) = (b,a) in
  let f = (* closure *)
    let x = 9 in
    (fun y -> y * x)
  in
  let rec f n =
    if (n > 0) then f (n - 1) else n
  in
  ```

OCaml Functions

-
  ```
  (* lets use pattern matching *)
  let rec f = function
     0 -> 0
   | n -> f (n - 1)
  in
  ```

OCaml Conditionals

-
  ```
  let res = if (cond) then value1 else value2 in
  let res = match x where
       x::xs -> Some (x::xs) (* pattern matching *)
     | []    -> None
  in
  let not_none = match x where
       None -> false
     | _ -> true
  in
  ```

OCaml Types

-
  ```
  let a = (x,y) ;; (* tuples can be of mixed types *)
  type color = { r : int ; g : int ; b : int };;
  let b = { r = 1.0 ; g = 0.5; b = 0.5 } ;; (* structs *)
  type cheese = Cheese of string;;
  let c = Cheese(‘‘Havarti’’);;
  type coord = ((a:int) * (b:int));;
  ```

OCaml types and class SML Style

-
  ```
  type pizza =   Crust of pizza | Pepperoni | Olives
       | Cheese of pizza list ;;
  let pizza = Crust(Cheese(
     [ Pepperoni ; Olives ; Crust(Pepperoni)]
  ));;
  let rec just_crust_and_cheese =
    function
       Crust(x) -> just_crust_and_cheese x
     | Cheese([]) -> true
     | Cheese(x)  -> List.for_all just_crust_and_cheese x
     | _  -> false
  ;;
  just_crust_and_cheese (Crust(Cheese([])));;
  just_crust_and_cheese pizza;;
  ```

OCaml line endings

- `in` means assign the value of the express to this symbol in this scope. Much like mathemtical notation
- `;` semi-colon is similar to the perl comma operator. It means ignore the return value of this expression (usually used with Unit expression)
- `;;` Used to terminated global scope, this is if you want to make globals or globally accessible functions
- _ Couldn't find a good slide for _ it just means match anything or ignore the value. Many programs are run by `let _ = expr1 ; expr2 ; expr2 ;;`

OCaml values are not mutable

- Most values are not mutable (arrays and strings are mutable)
- Even struct entries are not mutable. if you change them you are copying them.
  - `type foo = { num : int; mutable name: string }`
- Arrays have mutable values
- References are possible:
  - `let i = ref 0`
- To change a struct or a reference:

  -
    ```
    (* deref i and add 1 to it and assign it *)
    i := !i + 1; array.(!i) <- !i; (* array assn *)
    (* assign a value to an entry in a struct *)
    f.name <- ‘‘lolcakes’’;
    ```

Helpful OCaml modules

- The default modules handle things like Unix syscalls to do networking and some synchronization primitives. Even wimpy regexes.
- PCRE helps OCaml alot, the interface is very clear.
- Camlimages - image library
- SDL - for generaly multimedia
- Lablgtk - GTK bindings
- ocaml-gsl - Gnu Scientific Library

OCaml Sucks

- The comment and integer multiply cause little syntax bugs
- Can’t declare operator classes like haskell. Basically no operator overloading. Floats and ints don’t share same operator but everything shares ¿, = ,¡ and compare
- Can’t generalize classes easily (use :¿ operator)
- Not a lot of libraries. Not a lot of tools.
- Arrays limited to 4mb of entries. Strings are limited to 4mb in size.
- When to use ;, in, or ;; is often confusing.
- Name Spaces can clash

OCaml Sucks pt2

- No default easy way to write binary ints or floats out to file handles or strings.
- Some of the API is really lacking and often you need external libs to make up for it.
- Many libs are old or out of date.
- Documentation regarding the C interface is lacking (no description of how to iterate through a linked list)
- Printf is a hack. You have to declare types properly as a format not a string to pass a template into Printf.
- Negative floating point numbers should be put in parentheses.

OCaml debugging tips

- If you can interpret or compile to byte code you can use ocaml's interpretter to help debug
- Add more types. If you're not sure how an integer is being used stop using integers, make a type like `NumWaiters of int` to help check the types.
- If things get really painful syntactically you can always use Camlp4 but that probably won't help you debug.
- Learn how OCaml describes types, most compilation issues deal with not converting types or the compiler thinks you are using it wrong.
- When debuging start putting type hints everywhere like:

```
let fabs (x:float) = if x >= 0. then x else (-1.0) *. x in
```

OCaml summary

- Flexible language which allows for a variety programming styles
- Statically Typed
- Fast
- Sometimes cryptic and annoying
- Using OCaml’s type system is like programming while writing millions of assert statements which only get run at a compile time.
- I use OCaml for performance and I use perl for text processing and web automation and general scripts.
- I didn’t cover classes, modules or functors