First Impressions of the O’Caml Programming Language.

Erik de Castro Lopo
<erikd@mega-nerd.com>

Sydney Linux User Group
August 27, 2004

These slides are available for redistribution and modification under the terms of the Creative Commons Attribution Licence.
http://creativecommons.org/licenses/by/2.0/
In order to understand recursion, one must first understand recursion.
Why O’Caml

- I wanted a better higher level language.
- I wanted a language with strict, static types.
- Wanted generic programming with a clean syntax.
- I already code in C and Python.
- Don’t like C++ but do use it when C and Python don’t fit.
- I’ve tried Java, Perl and others.
My Progress So Far

- I've been programming O'Caml for about five weeks.
- Written about 3000 lines of code.
- Many small demo and test programs.
- One larger project of about 2500 lines.
- Rough subjective metric: one line of O'Caml is worth about 10 lines of C, 6 lines of C++ and 2 lines of Python.
What is O’Caml?

- A functional language.
  
  A pure function is one with no side-effects
  
  example:  \( \sin(x) \), \( \text{strcmp}\ (\text{str1}, \text{str2}) \)
  
  counter example:  \( \text{strcat}\ (\text{dest}, \text{src}), \text{fread()} \)

- An imperative language (mainly used at the top level).

- An object oriented language (but, can do a lot without OO).
License

- Written at INRIA : Institut National de Recherche en Informatique et en Automatique.

- The compilers are licensed under the QPL.

- The O’Caml libraries are licenses under the LGPL with extra exception:
  
  Can link a closed source code to Ocaml libraries to produce a closed source binary.
Pros of O’Caml

• High level language, with strict static types.

• Portable (*nix, Win32 and MacOSX).

• Runs as a script, compiles to bytecode (cw Java) or native binaries.

• Native bounds checked strings, lists, tuples and arrays (multi-dimensional).

• Typesafe (no type casting and no runtime type errors).
• Garbage collection.

• Library contains hash tables, sets, queues etc.

• Can define new, custom operators (ie ++, ** etc).

• Parameterized modules for generic programming.

• Exceptions.

• ocamllex and ocamlyacc.

• GTK+ bindings.

• Functional language goodness : closures, higher order functions, currying.
Cons of O’Caml

- Not as popular as other languages.

- More difficult for C, C++ and Java programmers to learn than Python or Perl.

- Poor low level language.

- String handling is primitive in comparison to Perl/Python.

- Imperative programming support has limitations.

- Syntax is a little strange.
• Standard library is poorly documented (IMO docs need examples).

• Type inferencing works 99% of the time, not 100% of the time.

• No forward declaration of function prototypes.

• No circular dependencies between modules.

• Compiler needs better error recovery/reporting.
What is it good for?

- Programs requiring complex data structures.

- Apps where security is important:
  - Network daemons.
  - Setuid root programs.

- Apps where Perl/Python might be appropriate but the source needs to be encrypted (ie compiled).

- Numerical/scientific/mathematical programming.
• Compilers, interpreters, virtual machines and language translators.

• Apps where correctness is important.

• CGI scripting/programming.
What Programs Have Been Written in O’Caml?

- The O’Caml compilers and virtual machines.
- The MLDonkey P2P program.
- An OpenPGP key server.
- mod_ocaml: an Apache module for running O’Caml scripts.
- PsiLAB: a program for scientific research and data analysis.
- Programs for proving theorems in mathematics.
Running/Compiling Ocaml Code

- Running as a script:
  ```
  ocaml script.ml
  ```

- Compiling to bytecode:
  ```
  ocamlc -o program program.ml
  ```

- Compiling to native binary:
  ```
  ocamlopt -o program program.ml
  ```

- Compiling to bytecode and native binaries can also be done with multiple files.
Types in O’Caml

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

# let sum x y = x +. y ;;
val sum : float -> float -> float = <fun>

# let sum (x:int) (y:float) = (float_of_int x) +. y ;;
val sum : int -> float -> float = <fun>

# type rational = { n : int ; d : int } ;;
type rational = { n : int; d : int; }

# let ( ** ) a b = { n = a.n * b.n ; d = a.d * b.d } ;;
val ( ** ) : rational -> rational -> rational = <fun>
```
let rec join lst =
    match lst with
    | [] -> ""
    | [ s ] -> s
    | head :: tail -> head ^ ", " ^ (join tail)
    ;;

let slist = [ "qwe" ; "asd" ; "zxc" ] ;;
Printf.printf "List : %s\n" (join slist) ;;

(* Or just use the String module. *)
Printf.printf "List : %s\n" (String.concat ", " slist) ;;
let getenv envvar =
   try Sys.getenv envvar with Not_found -> ""
   ;;
let printvars () =
   List.iter (fun envvar ->
      Printf.printf "%s = %s\n" envvar (getenv envvar) ; )
   [ "AUTH_TYPE" ; "CONTENT_LENGTH" ; "CONTENT_TYPE" ;
      "DOCUMENT_ROOT" ; "GATEWAY_INTERFACE" ; "HTTP_ACCEPT" ;
      "HTTP_REFERER" ; "HTTP_USER_AGENT" ; "PATH_INFO" ;
      "PATH_TRANSLATED" ; "QUERY_STRING" ; "REMOTE_ADDR" ;
      "REMOTE_HOST" ; "REMOTE_IDENT" ; "REMOTE_USER" ]
   ;;
print_endline "content-type: text/plain\n\n" ;;
printvars () ;;
Variant Types

type arith =
    Value of string
   | Int of int
   | Float of float
   | Plus of arith * arith
   | Minus of arith * arith
   | Times of arith * arith
   | Divide of arith * arith
   | Power of arith * arith
;;

let expr = Power (Plus (Value "x", Float 1.0), Int 3) ;;
Handling Variant Types

```ocaml
let rec whatever expr =
  match expr with
  | Value v -> Value v
  | Int i -> Int i
  | Float f -> Float f
  | Plus (Float a, Float b) -> Float (a +. b)
  | Plus (a, b) -> Plus (whatever a, whatever b)
  | Minus (a, b) -> Minus (whatever a, whatever b)
  | Times (a, b) -> Times (whatever a, whatever b)
  | Divide (a, b) -> Divide (whatever a, whatever b)
  | Power (a, b) -> Power (whatever a, whatever b)
  ;;

let expr = Power (Plus (Value "x", Float 1.0), Int 3) ;;
let outout = whatever expr ;;
```
O’Caml Objects

```ocaml
class accumulator =
    object (self)
    val mutable x = 0
    method add y =
        x <- x + y
    method print () =
        Printf.printf "value = \%d\n" x ;
end;;

let a = new accumulator ;;

a#add 12 ;;
a#add 30 ;;
a#print() ;;
```
let rec intersect lst =
  function
      [] -> []
  |   head :: tail ->
      if List.mem head lst then
      head :: intersect lst tail
      else
      intersect lst tail
  ;;
  
let a = [ "a" ; "c" ; "b" ; "d" ] ;;
let b = [ "c" ; "f" ; "d" ; "e" ] ;;

let inter = intersect a b ;;
Also Worth Looking At

- Modules.
- Ocamllex and ocamlyacc.
- Generic programming.
- O’Caml and GTK+.
- Exceptions.
- Camlp4.
Resources

- http://www.ocaml.org/
- GODI (CPAN for O’Caml)
- http://caml.inria.fr/bin/wilma/caml-list
- ocaml-beginners@groups.yahoo.com
Conclusion

- I like it.

- Functional programming works.

- O’Caml is a language for solving practical problems.

- For some problems O’Caml is a significantly better tool than any other language I’ve used.

- I’m not completely convinced that type inferencing is better than explicit types.