Typed Functional Programming In OCaml

Fabrice Le Fessant
fabrice.le_fessant@{inria.fr,ocamlpro.com}

November 6, 2013

Introducing Myself

- Full-time researcher at INRIA, programming languages and distributed systems
- 2001 : PhD on **JoCaml**, a DSL for concurrency, distribution and mobility
- 2002 : **MLdonkey**, first multi-protocol peer-to-peer client (edonkey, gnutella, bittorrent, etc.)
- 2007 : **MNPlight**, first iPhone application able to install mp3s on a jailbroken iPhone 1

→ all in the **OCaml programming languag**e

Introducing Myself

- Full-time researcher at INRIA, programming languages and distributed systems
- 2001 : PhD on **JoCaml**, a DSL for concurrency, distribution and mobility
- 2002 : **MLdonkey**, first multi-protocol peer-to-peer client (edonkey, gnutella, bittorrent, etc.)
- 2007 : **MNPlight**, first iPhone application able to install mp3s on a jailbroken iPhone 1
- 2011 : **OCamlPro**, a company to support the use of OCaml in industrial projects

A Poll !

How many of you have some experience of:

- Lisp or Scheme ?
- F# or Scala ?
- Haskell ?
- OCaml ?

What is OCaml ?

- A **General-purpose** Programming Language **developed** for about **30 years** at INRIA
- **Used** *from the beginning* to **develop** many applications at INRIA :
  - Coq proof assistant, Zenon, Alt-ergo
  - Hevea (LaTeX → Html)
  - spamoracle (bayesian spam filter)
  - synDEX (scheduler for embedded systems)
  - Coccinelle (Linux Kernel bug checker)

→ **OCaml is definitively not a lab toy !**

OCaml, as a FP language (1)

- What is Functional Programming ?
  - A way of programming, closer to mathematics

    → make it easier to implement complex algorithms

    → make it possible to reason about the correctness of implementations
  - Usual features of FP languages:

    → immutable variables, immutable values

    → functions as values

    → use of (tail) recursion instead of loops

    → strong type-checking

OCaml, as a FP language (2)

- Where is OCaml among FP languages:
  - Hybrid FP languages: Scala, F#, Clojure,etc.

    → FP extensions, ”a taste of FP”... but tainted
  - Untyped FP languages: Lisp, Scheme, Erlang,etc.

    → FP, lack the power of strong type systems
  - Pragmatic FP languages: OCaml, SML

    → add other styles over FP, best of both worlds ?
  - Pure FP languages: Haskell

    → closer to maths, but hard to program with
  - Proof languages: Coq, Isabelle, etc.

    → write a math proof, generate code from it

OCaml in the Industry

OCaml was designed at the beginning for formal methods applications: compilers, verifiers, provers...

- Microsoft : SLAM driver verifier
- Esterel Technologies : Scade KCG Compiler (scade-to-C, qualified level A DO-187B)
- AbsInt : Astree no-RTE checker
- EADS : Penjili, C code checker
- Dassault Systemes : Lucid/Esterel Compiler
- Airbus/Atos Origin: Toaster C style-checker

Success Stories: Citrix

- 2002: Cambridge University releases Xen

  → need a program to control Xen in VM0

- 2004: 30 developers, C, Python et Ruby...
- 2006: many m$ spent, yet, no product...
- 2006: new team of 4 OCaml devs, hired to write the doc, start a prototype in OCaml
- 2007: product available in OCaml, XenSource sold 500m$ to Citrix
- 2011: Citrix holds 15% of the virtualisation market (Amazon EC2 for example)

Success Stories: Jane Street

- 2000: Jane Street starts high-frequency trading in Excel + Visual Basic, too unreliable
- 2003: begin conversion from VB to C#
- 2003: one intern starts writting OCaml code
- 2005: management decides to try OCaml for key trading systems
- 2006: half of the system already in OCaml
- 2012: 10 billion$ per day of automatic trading, everything in OCaml with 100+ OCaml devs

OCaml is a multi-paradigm language

- **Functional** (functions are values, tail recursion)
- **Modular** (interfaces, functors and first-class modules)
- **Imperative** (mutable values, loops, exceptions)
- **Object-oriented** (objects and classes)
- **Statically** and **Strongly Typed**
- Execution is **strict** by default, **lazy** on demand
  - **Strict** = computation done where it is written
  - **Lazy** = computation delayed until useful

OCaml Implementation

- **Native-code compiler** for x86/amd64, arm,...
- **Bytecode compiler, interpreter (REPL) and debugger** for fast development loop
- **Efficient incremental** garbage collector with **compaction**
- **Compact** uniform data **representation**
- Small but efficient **standard library**
- **FFI bindings** with many C libraries (databases, crypto, GUIs, etc.)

Performances ?

- Strong Typing → No runtime checks !
- Highly optimised GC for short lifetime values
- Native-code compiler
  with few but efficient optimisations
  (constant folding, inlining, register coalescing)
- Strict execution

→ expectable performance

→ close to non-optimized C speed
(about 15% slower)

→ easy to optimise manually

A Taste of OCaml

- **Warning**:

  OCaml has a weird syntax

  - Difficult to learn at the beginning... :-(
  - Makes programs easier to read on the long term :-)

Basic Values

Simple values

```
let str = ”Hello world”
let four = 2 * 2
let pi2 = 3.14 *. 2.  (* No operator overloading ! *)
let list = [ 1 ; 2 ; 3 ; 4 ; 5 ]
let list = 1 :: 2 :: 3 :: 4 :: 5 :: []
let tuple = (x, y, z)
let array = [| ('a', 97); ('b', 98); ('c', 99) |]
let record = { x = 1; y = 12 }
```

There are no NULL pointer in OCaml, all values **must be initialized !**

Calling Functions

Functions arguments are **currified:**

```
let add (x,y) = x + y          (* one argument ! *)
let add x y = x + y            (* two arguments ! *)
let three = add (add 1 1) 1
```

Functions can be **partially applied**:

```
let add_one = add 1    (* val add_one : int → int *)

let list = [ 1;2;3;4;5 ]
let list_plus_two = List.map ( add 2 ) list
                    (* → [ 3; 4; 5; 6; 7 ] *)
```

Recursive Functions

Recursivity is intuitive to work on lists and trees

```
let rec fold_left f acc list =
  match list with     (* fold_left f x [a;b;c] ↔  *)
    [] → acc          (*       f (f (f x a) b) c  *)
  | head :: tail →
      fold_left f (f acc head) tail
```

```
let sum_list = List.fold_left add 0
                 (* val sum_list : int list → int *)
let mul x y = x *. y
let mul_list = List.fold_left mul 1.
               (* val mul_list : float list → float *)
```

Imperative Style

Side-effects, loops (while, for) and exceptions

```
let read_lines filename =
   let ic = open_in filename in
   let lines = ref [ ] in
   try                (* ref: mutable value *)
     while true do
        lines := input_line ic :: !lines
            (* := modifies, ! for extraction *)
     done; assert false
   with End_of_file → (* exceptions are cost-free ! *)
     close_in ic; List.rev !lines
```

FP Style

```
let (|>) x f = f x                      (* x |> f ↔ f x *)
let normal_users = ”/etc/passwd”
    |> read_lines
    |> List.map (Str.split_delim (Str.regexp ”:”))
    |> List.map (fun list → match list with
       |  login :: _passwd :: uid :: _ ->
                            (login, int_of_string uid)
        |  _ -> assert false )
    |> List.filter (fun (_, uid) -> uid >= 1000)
    |> List.map fst
```

Closures

Variable bindings last for ever in functions...

```
let new_counter () =
  let x = ref 0 in fun () → x := !x + 1; !x
      (* new_counter : unit → (unit → int) *)
let new_foo = new_counter ()
let foo_id1 = new_foo ()                   (* → 1 *)
let new_bar = new_counter ()
let bar_id1 = new_bar()                    (* → 1 *)
let bar_id2 = new_bar()                    (* → 2 *)
let bar_id3 = new_bar()                    (* → 3 *)
let foo_id2 = new_foo()                    (* → 2 *)
```

Where are type annotations ?

- OCaml is a statically typed language

  with one of the most expressive type-systems (variants, records, optional args, GADTs, polymorphic variants, objects, classes, etc.)
- Compiler is supposed to verify types !

  → but I didn't see any type annotations ?
- In OCaml, types are automatically infered:
  - You don't need to write them
  - The compiler will guess them, and complain if they don't match what is expected

Type-inference

```
(* val read_lines: string → string list *)
let read_lines filename =
   let ic = open_in filename in
               (* filename : string & ic : in_channel *)
   let lines = ref [ ] in
                                (* lines: '_a list ref *)
   try
     while true do
        lines := input_line ic :: !lines
                             (* lines: string list ref *)
     done; assert false
   with End_of_file -> close_in ic;
     List.rev !lines
```

Polymorphic Functions

Our function on list works on any list !

```
let rec fold_left f acc list = match list with
    [] → acc
  | head :: tail →
      fold_left f (f acc head) tail
      (* ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)



let sum_list = List.fold_left add 0
                 (* val sum_list : int list → int *)
let mul x y = x *. y
let mul_list = List.fold_left mul 1.
             (* val mul_list : float list → float *)
```

Defining New Types

```
type expression =
  { exp = expression_desc;
    loc : Location.t; }
and expression_desc =
  | Num of int
  | Var of string
  | Let of string * expression * expression
  | Binop of operator * expression * expression
and operator = Plus | Minus | Times | Div
```

Algebraic Data Types avoid accessing the wrong arguments of an enum selector.

Pattern-Matching

```
let rec eval env v = match v.desc with
  | Num i -> i
  | Var x -> List.assoc x env
  | Let (x, e1, body) -> let val_x = eval env e1 in
     eval ((x, val_x) :: env) body
  | Binop (Plus, e1, e2) ->
     (eval env e1) + (eval env e2)
  | Binop (Minus, e1, e2) ->
    (eval env e1) - (eval env e2)
```

It is possible to match **deep and complex patterns**, that are always compiled in the **optimal number of runtime tests**.

More Checks

```
let rec eval env v = match v.desc with
  | Num i -> i
  | Var x -> List.assoc x env
  | Let (x, e1, body) -> let val_x = eval env e1 in
     eval ((x, val_x) :: env) body
  | Binop (Plus, e1, e2) ->
     (eval env e1) + (eval env e2)
  | Binop (Minus, e1, e2) ->
    (eval env e1) - (eval env e2)
```

Warning 8: **this pattern-matching is not exhaustive.**
Here is an **example** of a value that is not matched:
Binop (Times | Div, _, _)

Simple Networking

```
(* start_server : int → (int → unit) → unit *)
let start_server port handle_connection =
  let server = Unix.socket PF_INET SOCK_STREAM 0 in
   Unix.setsockopt server SO_REUSEADDR true;
   Unix.bind server ADDR_INET (inet_addr_any, port);
   Unix.listen server 3;
   while true do
     let (client, addr) = Unix.accept server in
     ignore (Thread.create handle_connection client)
   done
```

Modules and Interfaces

Interface file: server.mli

```
val start_server : int → (int → unit) → unit
val read_lines : string → string list
  ...
```

Implementation file: server.ml

```
open Unix
let read_lines filename = ...
let start_server port handle_connection = ...
```

The compiler checks the **consistency of all compiled files** in the **whole project**: the compiler is often used as a **refactoring assistant** !

OCaml Ecosystem

- OPAM, a **source package manager** to install OCaml and its open-source contributions
  - http://opam.ocamlpro.com/
- Js_of_ocaml, a powerful **OCaml-to-JavaScript optimizing compiler**, to run OCaml typed-checked applications in the browser
  - http://oscigen.org/js_of_ocaml/
- Mirage: **bare-metal applications for Xen** in OCaml, speed and security in a **Cloud OS** !
  - http://openmirage.org/

Formal Methods

- Use of Mathematics in the design of Hardware/Software applications
- **Strong type-checking** with OCaml
- **Abstract Interpretation**:
  - Astree, no runtime error in Airbus C code

    http://www.absint.com/astree/
- **Verification of formal specifications**:
  - Frama-C: used by Airbus on critical boot code

    http://frama-c.com/

Formal Methods

- **Mecanized Proof** of an Algorithm and **automatic code generation**:
  - CompCert, a full C compiler, proved within the Coq proof assistant
  - http://compcert.inria.fr/
  - http://coq.inria.fr/

Discussion

- Questions ?

- OCaml:
  - Web: http://www.ocaml.org/
  - Try it online: http://try.ocamlpro.com/
  - Install: http://opam.ocamlpro.com/