A Brief Presentation of OCaml

Simon Cruanes

September 14, 2015

Summary

1. The Basics
2. Advanced Features
3. Ecosystem
4. Comparison with Other Languages

History and Early Life

- Born 20 years ago (1995)
- Family: ML languages
- Siblings: SML (Standard ML)
- Designed for writing Coq (proof assistant)

  (ML invented to write proof assistant)

First Taste

```
let x = 1 ;;

let l = [1;2;3] ;;

assert (x+1 = 2) ;;

assert (List.map (fun x -> x+1) l = [2;3;4]);;
```

note: the `;;` only necessary in toplevel!

Details

- **`let`** introduces a variable binding
- `x` and `l` are immutable
- use `=` for equality

Types

OCaml is strongly-typed.

```
# let x = 1 ;;
val x : int = 1

# let l = [1;2;3] ;;
val l : int list = [1;2;3]

# List.map ;;
– : (’a –> ’b) –> ’a list –> ’b list = <fun>
```

- types are inferred automatically
- List.map is polymorphic (’a, ’b are type variables)

Types (continued)

```
# let rec map f l = match l with
    | [] -> []
    | x :: tail ->
        let y = f x in
        y :: map f tail
;;
val map : (’a -> ’b) -> ’a list -> ’b list = <fun>
```

Even polymorphic types are inferred.

A Survey of Types

Many flavours of types:

primitives `int, bool, float...`

records (C-like structures)

```
    type ’a list_len = {
        the_list : ’a list;
        the_len : int;
    }
```

sum types (better than C enums)

```
type ’a option = None | Some of ’a

type ’a tree =
    | Empty
    | Node of ’a * ’a tree * ’a tree
```

strings `string` (immutable) and `bytes` (mutable); no unicode

tuples

```
# (1, "foo", false) ;;
– : (int * string * bool) = (1, "foo", false)
```

Pattern-Matching

```ocaml
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree

let rec size t = match t with
    | Empty -> 0
    | Node (_, l, r) -> 1 + size l + size r
(* size : 'a tree -> int *)

let to_list t =
    let rec aux acc t = match t with
        | Empty -> acc
        | Node (x, l, r) ->
            let acc = aux acc r in
            let acc = x :: acc in
            aux acc l
    in
    aux [] t
(* to_list : 'a tree -> 'a list
   infix traversal *)
```

really powerful! (nested, guards, or-patterns. . .)

Mutability

OCaml is impure: values can be mutated

- variables are immutable
- some *record fields* can be mutated
- `'a array`, `bytes`: mutable arrays
- `'a ref` defined as record

```
type 'a ref = {
    mutable contents : 'a;
}

let (!) r = r.contents
(* (!) : 'a ref -> 'a *)

let (:=) r x = r.contents <- x
(* (:=) : 'a ref -> 'a -> unit *)
```

Summary and Remarks

- Functional Language (immutability, 1st-class functions)
- strong typing with excellent inference
  - rich variety of types (sums, records, tuples, etc.)
  - types might be written for readability
- expressive and quite efficient
- fast GC on one core
- compiles into native code

Summary

1 The Basics

2 Advanced Features

3 Ecosystem

4 Comparison with Other Languages

Modules

Powerful module system (inherited from SML)

```
module A = struct
    type t = { foo : int }
    let f x = x.foo + 1

    module B = struct
        type t = { bar: string }
        let g x = x.bar ^ x.bar
    end
end

let x = {A.foo = 42} ;;
A.f x ;;  (* 43 *)

let y = {A.B.bar = "cou"} ;;
A.B.g y ;;  (* "coucou" *)
```

Modules (continued)

Functor: function from module to module

```
module type ORD = sig
    type t
    val compare : t -> t -> int
end

module Set(E : ORD) : sig
    type elt = E.t
    type t

    val empty : t
    val add : elt -> t -> t
    val mem : elt -> t -> bool
end
```

Here, Set is a functor (builds a set structure for an ordered type)

Others

- Objects (powerful, but complicated!)

  subtyping, inheritance, structural types ($\sim$ Golang)...
- GADTs (more expressive sum types)
- polymorphic variants (structural sums)
- named and optional parameters
- 1st-class modules (pass modules as values)
- and more...

Summary

1. The Basics
2. Advanced Features
3. Ecosystem
4. Comparison with Other Languages

Tools

Compiler: compiles fast, many warnings, quite hackable

Merlin: awesome completion/typing in vim/emacs/...

Opam: nice package manager (∼ 1000 paquets)

Build Systems: Several competing systems

Debugger: meh.

Profiler: use gprof or perf

C bindings: stubs, Ctypes (ffi)

Libraries

- several competing Stdlibs
- web frameworks (ocsigen, opium)
- networking, json, etc.
- bindings to Sqlite and postgres
- Lwt: monadic concurrency (futures), scalable
- ...

libs are good quality overall

Users

- Research (majority): logic, bioinfo. . .
- Industry: Finance (Janestreet), aeronautics, a few startups. . .
- FP amateurs
- OCamlpro
- language maintained by Inria

Consortium for the industrial users

Summary

1. The Basics

2. Advanced Features

3. Ecosystem

4. Comparison with Other Languages

. . . with Haskell

Differences:

OCaml	Haskell
Strict	Lazy
Impure (mutability)	Pure
no overloading	typeclasses
modules, functors	only modules
Predictible Performance	Hard to predict
1 core	Multi-core
opam	cabal

. . . with Erlang

Differences:

OCaml	Erlang
Typed	Not typed
modules, functors	only modules
compilation	reload on the fly
Predictible Performance	same?
1 core	multi core
monadic concurrency	builtin actors, OTP

. . . with Scala

Differences:

OCaml	Scala
native code	JVM
lightweight	verbose
simple	more complicated
no overloading	implicits
modules, functors	only modules
small stdlib	stdlib, scalaz

. . . with Java

Differences:

OCaml	Java
native code	JVM
expressive	verbose
type inference	nope
immutability	effects at distance
few libs	many libs

. . . with Others

- Also: SML, F, experimental langs (Eff, Mezzo, 1ML)
- Can compile to Javascript (!)
- less hype than Haskell, but maybe more robust (no space leak or weird performance or cabal)
- used in real industries, real programs

Conclusion

- simple, expressive, safe language
- more advanced features when you need them
- ecosystem is blooming (opam)
- exciting research (Mirage, a unikernel)
- extensions in dev: multicore, modular implicits (> typeclasses)