The core Caml system, 2009–2010

Xavier Leroy

INRIA Paris-Rocquencourt

OCaml users meeting, 2010-04-16

This meeting brought to you by...

Sylvain Le Gall at OCamlCore (general organization).

INRIA Paris-Rocquencourt conference bureau (local arrangements).

The Caml Consortium (funds).

Outline

1. Caml development news
2. Caml consortium news
3. New language features in OCaml 3.12
4. Closing remarks

Recent releases

Minor release 3.11.1 (june 2009):

- 45 problem reports fixed.

Minor release 3.11.2 (january 2010):

- 32 problem reports fixed
- Debugger (`ocamldebug`) updated and improved (X. Clerc).
- 8 feature wishes granted.

Next release

Major release 3.12.0:

- Surprisingly many new language features! (See later.)
- More bug fixing & wish granting.
- Almost no backward-incompatible changes.

Next release

Tentative planning:

- Done: feature freeze.
- May–June: finish merging and documentation; update `camlp4` and `ocamldoc`; bug fixing.
- Early June: first beta release.
- Early July: final release.

As usual, testing and feedback are much appreciated.

Manpower

On the rise, esp. thanks to external contributors:

- Alain Frisch (Lexifi)
- Mark Shinwell (Jane Street)

Plus the usual suspects:

- The "historic" INRIA team.
- Jacques Garrigue (Nagoya university).
- Xavier Clerc (INRIA research programmer, part-time).

Equivalent to about 1 person full-time.

Legal status of contributions from outside INRIA was clarified.
(Contributor License Agreement.)

Outline

1. Caml development news
2. Caml consortium news
3. New language features in OCaml 3.12
4. Closing remarks

Members

One new member this year: MLState.

11 members total:

before	2007	2008	2009	2010
Dassault Aviation	Intel	CEA	SimCorp	MLState
Dassault Systèmes	Jane Street	OCamlCore		
Lexifi	Citrix			
Microsoft				

Actions of the Consortium

What the Consortium does:

- Sell permissive licensing conditions on the Caml code base.
- Enable lightweight corporate sponsoring.
- A place to discuss needs with power users from industry.
- Public relation.
- Brings "pocket money" e.g. for sponsoring this meeting.

New this year:

- Acts as a "sounding board" for discussing new features.
- Two members contributing directly to the Caml code base.

Latest meeting of the Consortium

December 2009, in Paris.

Well attended: 12 participants + 4 INRIA.

Fruitful discussions of possible extensions and future developments
(a majority of which materialized in 3.12.0)
(continuing on the Consortium mailing list).

Outline

1. Caml development news
2. Caml consortium news
3. New language features in OCaml 3.12
4. Closing remarks

1. Record notations

In record patterns and record expressions, a component *id* stands for *id* = *id*, and *M*.*id* stands for *M*.*id* = *id*.

```
open Complex

let polar d theta =
  let re = d *. cos theta and im = d *. sin theta
  in { re; im }

let conj { re; im } = { re; im = -. im }
```

1. Record notations

A record pattern can end with `; _`, meaning "this pattern doesn't list all fields of the record type, but this is intentional".

```
open Complex

let proj { re = x } = x                (* warn if warning R active *)

let proj { re = x; _ } = x             (* does not warn *)
```

Warning (turned off by default) if no `;_` and some fields are missing.

2. Explicit method override

`method!` defines a method like `method` does, but mark intent to override a method of the same name already defined in a superclass.

```
class sub_c = object
    inherit c
    method! m = ...
    method  n = ...
end
```

Error if `c` does not already defines a method named `m`.

Warning (turned off by default) if `c` defines a method named `n`.

(Same for `val!` and `inherit!`.)

3. Local open (`let open ... in ...`)

By popular demand and also because the corresponding Camlp4 extension was not robust enough:

```
let polar d theta =
  let open Complex in { re = d *. cos theta; im = d *. sin theta }
```

3. Local open (alternative notation)

M`.(`*e*`)` equivalent to `let open` *M* `in` *e*

```
module Float = struct
  let ( + ) = ( +. )
  let ( * ) = ( *. )
end

let norm x y = Float.(sqrt(x * x + y * y))
```

(Taking a leaf from Christophe Troestler's "delimited overloading" package, but much less powerful.)

4. Polymorphic recursion

Variables bound by `let` and `let rec` can receive an explicit polymorphic type `'a.`τ

```
let id : 'a. 'a -> 'a = fun x -> x          (* OK *)

let id : 'a. 'a -> 'a = fun x -> 1          (* Error *)

let id :     'a -> 'a = fun x -> 1          (* OK with 'a = int *)
```

4. Polymorphic recursion

Enables recursive definitions where the recursively-bound functions can be used at several types within the recursion.

```
type term =
  A of int | B of (string * term) list | C of (int * term) list

let rec shift = function
  | A x -> A (x + 1)
  | B l -> B (shift_list l)
  | C l -> C (shift_list l)

and shift_list: 'a. ('a * term) list -> ('a * term) list = function
  | [] -> []
  | (key, t) :: rem -> (key, shift t) :: shift_list rem
```

(Plus: non-regular recursive datatypes, e.g. Okasaki's data structures.)

5. First-class modules

Encapsulate a module as a core language value (with an explicit type), then recover the module from this value.

$$\begin{aligned}
\textit{expr} &::= \ldots \mid \texttt{(module } \textit{module-expr} \texttt{ : } \textit{package-type}\texttt{)} \\
\textit{module-expr} &::= \ldots \mid \texttt{(val } \textit{expr} \texttt{ : } \textit{package-type}\texttt{)} \\
\textit{type} &::= \ldots \mid \texttt{(module } \textit{package-type}\texttt{)} \\
\textit{package-type} &::= \textit{modtype-path} \texttt{ with } t_1 = \tau_1 \texttt{ and } \ldots t_n = \tau_n
\end{aligned}$$

(An extension of Claudio Russo's proposal, part of Moscow ML.)

5. First-class modules

Typical use: selecting at run-time among several implementations of a signature.

```
module type DEVICE = sig ... end
let devices : (string, (module DEVICE)) Hashtbl.t
            = Hashtbl.create 17

module SVG = struct ... end
let _ = Hashtbl.add devices "SVG" (module SVG : DEVICE)

module PDF = struct ... end
let _ = Hashtbl.add devices "PDF" (module PDF: DEVICE)

module Device =
  (val (try Hashtbl.find devices (parse_cmdline())
        with Not_found -> eprintf "Unknown device %s\n"; exit 2)
   : DEVICE)
```

5. First-class modules

More advanced uses:

- Functors that take a list of structures as argument.
- Encodings of first-class values with existential types.
- Encodings of some Generalized Algebraic Data Types.

6. Named types as parameters to functions

`(type` t`)` in the parameter list of a function.

- Within the function, `t` is a new, abstract type name.
- Outside, `t` becomes a regular type variable α
  (which can be generalized or instantiated as usual).
- No run-time effect (no type is actually passed).

6. Named types as parameters to functions

Usage: bridging module-level constructs and core-level polymorphism.

```
let sort_uniq (type s) (cmp : s -> s -> int) (l: s list) =
    let module S =
        Set.Make(struct type t = s let compare = cmp end) in
    S.elements (List.fold_right S.add l S.empty)
```

The function `sort_uniq` has type

$$\forall\alpha.\ (\alpha \to \alpha \to \texttt{int}) \to \alpha\ \texttt{list} \to \alpha\ \texttt{list}$$

6. Named types as parameters to functions

Another example: local exceptions in polymorphic functions.

```
let new_exn (type t) () =
    let module M = struct exception E of t end in
    (fun x -> M.E x), (function M.E x -> Some x | _ -> None)
```

The function `new_exn` has type

$$\forall\alpha.\ \texttt{unit} \to (\alpha \to \texttt{exn}) \times (\texttt{exn} \to \alpha\ \texttt{option})$$

7. Recovering the type of a module

`module type of` *M* denotes the type of the module expression *M*.

It can be used in conjunction with `include` to enrich the signature of an existing module:

```
module type MYHASH = sig
    include module type of Hashtbl
    val add_all: ('a, 'b) t -> ('a, 'b) t -> unit
end

module MyHash : MYHASH = struct
    include Hashtbl
    let add_all t1 t2 = iter (add t1) t2
end
```

8. Substitution & removal of types in signatures

S `with type` t `:=` τ

- Deletes the declaration `type` t from signature S
- Replaces all uses of t in S with τ.

Contrast with S `with type` t `=` τ, which

- Enriches the declaration `type` t as `type` $t = \tau$
- Keeps the declaration of t.

8. Substitution & removal of types in signatures

Application: combine signatures that have identically-named types.

```
module type S1 = sig type t val op1: ... end
module type S2 = sig type t val op2: ... end

module type S1plus2 =
  sig                              (* or: *) sig
    type t
    include S1 with t := t                    include S1
    include S2 with t := t                    include S2 with t := t
  end                                       end
```

Cannot do with regular `with type` $t = \tau$ constraints, because multiple `t` components remain.

Outline

1. Caml development news
2. Caml consortium news
3. New language features in OCaml 3.12
4. Closing remarks

Personal wishes

Hope you will like OCaml 3.12!

How can you help?

- By testing & providing quick feedback.
- By volunteering to work on parts we handle poorly (esp. the Windows port and the Windows binary distributions).
- By joining community efforts, esp. in the area of packaging and distribution.

Keep up the good work!