25 years of OCaml

Xavier Leroy
OCaml 2021
Collège de France and Inria
We are proud to announce the availability of Objective Caml version 1.00.

Objective Caml is an object-oriented extension of the Caml dialect of ML. It is statically type-checked (no "message not understood" run-time errors) and performs ML-style type reconstruction (no type declarations for function parameters). This is arguably the first publicly available object-oriented language featuring ML-style type reconstruction.

Objective Caml is a class-based OO language, and offers pretty much all standard features of these languages, including "self", single and multiple inheritance, "super", and binary methods, plus a number of less common features such as parametric classes. [...]

Objective Caml is based on (and supersedes) the Caml Special Light system. It inherits from Caml Special Light a powerful module calculus, Modula-style separate compilation, a fast-turnaround bytecode compiler, and a high-performance native-code compiler. Upward compatibility with Caml Special Light is very high.
50 years of ML
The early years : from LCF to Core ML
Proofs are terms of type \( \text{thm} \), built using functions such as

\[
\text{trans} \ (t_1 : \text{thm}) \ (t_2 : \text{thm}) : \text{thm} =
\]

if \( t_1 \) is \( "A = B" \) and \( t_2 \) is \( "B = C" \) then return \( "A = C" \) else fail
To write these terms, Milner wanted a “meta-language” that was

• applicative (functional);
• interactive (with a REPL);
• strongly typed, to enforce type abstraction on type `thm` (making sure the user cannot build “0 = 1” : `thm`)

LISP would not do, hence Milner invented “ML”, a functional/imperative language with strong static typing and type abstraction.
Types of function parameters can be inferred from their uses (e.g. `fun x y -> x && not y`).

What if they cannot? (e.g. `fun x -> x`).

- Hindley: give type $\alpha \rightarrow \alpha$ for some fixed, unknown type $\alpha$.
- Milner: give a type schema $\forall \alpha. \alpha \rightarrow \alpha$ denoting a polymorphic function.
Built-in product types $t_1 \times t_2$ and sum types $t_1 + t_2$.

Other datatypes are defined as abstract types + constructor functions + accessor functions.

Example: binary trees with values of type $\ast$ at leaves.

```
abstract type * tree = * + * tree # * tree
    with leaf n = abstree(inl n)
        and node (t1, t2) = abstree(inr(t1, t2))
    and isleaf t = isl(reptree t)
    and leafval t = outl(reptree t) ? failwith 'leafval'
    and leftchild t = fst(outr(reptree t) ? failwith 'leftchild'
    and rightchild t = snd(outr(reptree t) ? failwith 'leftchild'
```
Inductive types and pattern matching

(R. Burstall, G. Cousineau, D. MacQueen, R. Milner, ...; HOPE, Prolog)

From “typed Lisp”...

... to Core ML

type 'a tree =
  | Leaf of 'a
  | Node of 'a tree * 'a tree

letrec sumtree t =
  if isleaf t then
    leafval t
  else
    sumtree (leftchild t)
    + sumtree (rightchild t)
Hindley-Milner polymorphic typing and type inference

Call-by-value functional language

Inductive types, pattern matching
A rich lineage

SASL
Miranda
LazyML
Hope
LCF ML
Lisp
Prolog
CAML
SML90
Haskell
Haskell98
Alice
MoscowML
Reason
F#

Agda
From CAML to Caml Special Light

(G. Cousineau, G. Huet, M. Mauny, A. Suarez, P. Weis)

Core ML + facilities for “embedded languages”
(parsers, quotations, anti-quotations)

Developed along the Coq proof assistant, as Coq’s implementation language.

```
let calc env = calcrec
  where rec calcrec = function
    'Constant(n) -> n
  | 'Variable(x) -> assoc x env
  | << ^e1 + ^e2 >> -> calcrec(e1) + calcrec(e2)
  | << ^e1 * ^e2 >> -> calcrec(e1) * calcrec(e2) ;;
```
CAML = ML running on the CAM


\[
\begin{align*}
[0] &= \text{snd} \\
[n + 1] &= \text{fst} ; [n] \\
[\lambda.M] &= \text{cur}([M]) \\
[M \; N] &= \text{push} ; [M] ; \text{swap} ; [N] ; \text{cons} ; \text{app}
\end{align*}
\]

Pro : one of the first formalizations of function closures.

Cons : inefficient ; one “cons” for each binding.
Je connais un langage où il y a un gros travail de compilation à faire.

*Let me tell you about a programming language where there is much compilation work to do.*

*(Guy Cousineau, spring 1988)*
The ZINC experiment (1989)

(X. L., D. Doligez)

• Core ML (simplified from CAML).
• Efficient generational GC.
• An abstract machine (the ZAM) where bindings use a stack.
• A bytecode interpreter written in C.

(X. L., D. Doligez, P. Weis, M. Mauny)

A completion of the ZINC experiment, practically usable, esp. for teaching.

- Type checking and type inference.
- Separate compilation and linking.
- Modula-2 modules: implementation file (.ml) + interface file (.mli).
- Toplevel interactive REPL.
- Bootstrapped.
- Available for Unix, Mac OS, and MS-DOS!
Many undergraduate CS courses used Caml Light, esp. in France.
Advanced language features for programming “in the large”: modules (structures) with multiple interfaces (signatures); parameterized modules (functors) with sharing constraints; …

As presented in the *Definition of Standard ML*: complex type-checking rules based on an internal, DAG-like representation.

Can we explain SML modules in type-theoretic terms? ($\forall$, $\exists$ quantification; dependent types; …)
A Type-Theoretic Approach to Higher-Order Modules with Sharing

Robert Harper†, Mark Lillibridge‡
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract
The design of a module system for constructing and maintaining large programs is a difficult task that raises a number of theoretical and practical issues. A fundamental issue is the management of the flow of information between program units at compile time via the notion of an interface. Experience has shown that fully opaque interfaces are awkward.

Using manifest types (X. L.) / translucent sums (R. Harper and M. Lillibridge) to express type propagation and sharing.

functor (X: sig type t ... end) -> sig type u = X.t ... end
Caml Special Light (1994–1996)

The language: the core Caml Light language + an SML-style module language using syntactic signatures and manifest types.

The implementation: the Caml Light runtime system + an improved ZAM2 bytecode compiler and interpreter + a native-code compiler.
Announcing Caml Special Light 1.06, the first public release of the Caml Special Light system.

Caml Special Light is a complete reimplementation of Caml Light that adds a powerful module system in the style of Standard ML. The module system is based on the notion of manifest types / translucent sums; it supports Modula-style separate compilation, and fully transparent higher-order functors (see the papers in the POPL 94 and 95 proceedings).

Caml Special Light comprises two compilers: a bytecode compiler in the style of Caml Light (but up to twice as fast), and a high-performance native code compiler for the following platforms: [...]

Ojective Caml
Object orientation in the 1990s

Inheritance Is Not Subtyping
William R. Cook  Walter L. Hill  Peter S. Canning
Hewlett-Packard Laboratories
P.O. Box 10490 Palo Alto CA 94303-0969

A wave that swept industry and software engineering

Non-OO programming languages were seen as irrelevant.

A puzzle for P.L. theory

Hard to explain O.O. in type-theoretic terms.
(Structural vs. nominal types; inheritance vs. subtyping; elusive encodings; …)
Row polymorphism for objects

(D. Rémy, J. Vouillon)

Using **rows** to keep track of method names and types, and **row variables** to keep track of other, not yet known methods.

\[
\langle m : \text{int}; \rho_1 \rangle \quad \langle k : \text{bool}; \rho_2 \rangle
\]

\[
\langle m : \text{int}; \emptyset \rangle \quad \langle m : \text{int}; k : \text{bool}; \rho \rangle
\]

Perfect for inferring the type of an object from its uses:

```
fun x -> x#name ^ string_of_int x#rank
  : < name : string; rank : int; .. > -> string
```

Note: parametric polymorphism, not subtype polymorphism.
Caml Special Light
+ objects with row polymorphism in the core language
+ a sub-language for classes (object generators),
  including multiple inheritance, self type specialization, …

class printable_colored_point y c as self =
  inherit colored_point y c
  inherit printable_point y as super
  method print =
    print_string "("; super#print; print_string ", ";
    print_string (self#color); print_string ")"
end
We are proud to announce the availability of Objective Caml version 1.00.

Objective Caml is an object-oriented extension of the Caml dialect of ML. It is statically type-checked (no "message not understood" run-time errors) and performs ML-style type reconstruction (no type declarations for function parameters). This is arguably the first publically available object-oriented language featuring ML-style type reconstruction.

Objective Caml is a class-based OO language, and offers pretty much all standard features of these languages, including "self", single and multiple inheritance, "super", and binary methods, plus a number of less common features such as parametric classes. [...]

Objective Caml is based on (and supersedes) the Caml Special Light system. It inherits from Caml Special Light a powerful module calculus, Modula-style separate compilation, a fast-turnaround bytecode compiler, and a high-performance native-code compiler. Upward compatibility with Caml Special Light is very high.
Reactions to Objective Caml

The FOOL community: polite lack of interest.
(“Nice, but not enough like Java.”)

Early adopters of ML: slight concern.
(“You’re not giving up on functional programming, right?”)

Many newcomers, reassured by familiar objects, quickly learned to use functions and datatypes instead.

**OCaml: the rehabilitation clinic for OO programmers.**

*(Erik Meijer)*
Two influential early uses

**Active VRML** (Todd Knoblock et al, Microsoft Research)
A domain-specific language for animated 3D scenes.

**Horus/ML then Ensemble** (Robert van Renesse et al, Cornell)
A toolkit for building distributed applications.

An unexpected affinity between OCaml and systems programming.
Major evolutions of O(bjective) Caml

- 2.00 (Aug 1998) Revised class language
- 3.00 (Apr 2000) Labeled/optional arguments; polymorphic variants
- 3.05 (Jul 2002) Polymorphic record fields and methods
- 3.07 (Sep 2003) Recursive module definitions
- 3.08 (Jul 2004) Immediate objects
- 3.12 (Aug 2010) First-class modules
- 4.00 (Jul 2012) Generalized Algebraic Datatypes (GADTs)
Two extensions prototyped by J. Garrigue in OLabl, then merged in OCaml 3.00:

**Labels on function arguments**, to make functions more self-documenting and to support optional arguments.

``` OCaml
StringLabels.sub ~pos: 5 ~len: 2 txt
```

**Polymorphic variants**, to mix and match data constructors freely.

``` OCaml
['On; 'Off] : [> 'Off | 'On ] list

function 'On -> 1 | 'Off -> 0 | 'Number n -> n
  : [< 'Number of int | 'Off | 'On ] -> int
```

Both extensions were motivated by GUI toolkits (LablTk, LablGTK).
Modules as first-class values

Implemented by A. Frisch based on a design by Cl. Russo for Moscow ML. Enable modules to be encapsulated as first-class values and manipulated by the core language.

```ocaml
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)
```
A natural idea: constructors of parameterized datatypes (‘a ty) may not all produce ‘a ty results, just instances τ ty.

type ‘a compact_array =
  | Array: ‘a array -> ‘a compact_array (* default case *)
  | Bytes: bytes -> char compact_array (* special case *)
  | Bools: bitvect -> bool compact_array (* special case *)

The devil is in the details of type inference for pattern-matchings over GADTs...
History of GADTs

1992  Läufer : *Polymorphic Type Inference and Abstract Data Types*. “Existential types”, a special case of GADT.

1994  Augustsson, Petersson : *Silly type families* (draft). Let’s remove the regularity condition over constructor types. Problems to infer the types of match.

2003  Xi, Chen, Chen : *Guarded Recursive Datatype Constructors*. Rediscovery of the same ideas.

2006  Peyton-Jones et al + Pottier and Régis-Gianas. First algorithms for partial type inference for GADTs pattern matching.


2012  OCaml 4.00 : introduction of GADTs in Caml.
Recent and planned evolutions

Since 4.00: many small additions to the language, e.g.

4.02 (Aug 2014)  match ... with exception ...
   Extensible datatypes
4.03 (Apr 2016)  Inline records
4.12 (Feb 2021)  Injectivity annotations on type constructors

In progress:

5.00  Multicore OCaml (shared-memory parallelism)
5.??  Some forms of algebraic effects
5.??  Modular implicits
In closing
Certainly, seen from 1996, the story [of Caml] could have been more linear.

(Guy Cousineau, 1996)

Seen from 2021, even more so!
A language that is still faithful to its roots

Mostly functional (+ imperative and OO when needed).

Types as the skeleton of the language.

Devotion to type inference and existence of principal types.
William Shakespeare
Sonnet 116

Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds;
Or bends with the remover to remove:
O, no! it is an ever-fixed mark,
That looks on tempests and is never shaken;
It is the star to every wandering bark,
Whose worth’s unknown, although his height be taken.
Love’s not Time’s fool, though rosy lips and cheeks
Within his bending sickle’s compass come;
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
If this be error and upon me prov’d,
I never writ, nor no man ever lov’d.
An active community that it still organizing

Much collective effort, as exemplified in this OCaml workshop.

Thanks to all for the many contributions.

Keep up the good work!