25 years of OCaml

Xavier Leroy

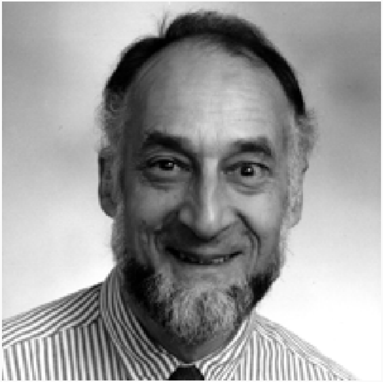
OCaml 2021

Collège de France and Inria

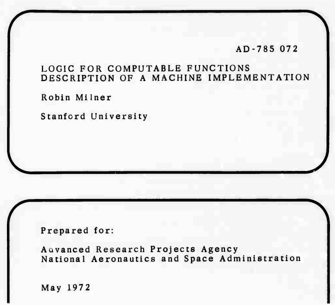
```
From: Xavier Leroy <xleroy AT pauillac.inria.fr>
To: caml-list AT pauillac.inria.fr, comp-lang-ml AT cs.cmu.edu
Subject: Objective Caml 1.00
Date: Thu, 9 May 1996 16:27:36 +0200 (MET DST)

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.
```

50 years of ML
The early years : from LCF to Core ML

Robin Milner, 1934–2010



LCF : an interactive prover for the Logic of Computable Functions


AD-785 072

LOGIC FOR COMPUTABLE FUNCTIONS
DESCRIPTION OF A MACHINE IMPLEMENTATION

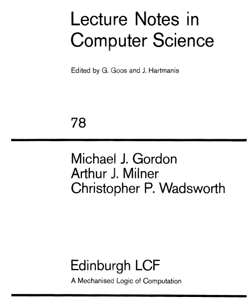
Robin Milner

Stanford University

Prepared for:

Advanced Research Projects Agency
National Aeronautics and Space Administration

May 1972


Lecture Notes in
Computer Science

Edited by G. Goos and J. Hartmanis

78

Michael J. Gordon
Arthur J. Milner
Christopher P. Wadsworth

Edinburgh LCF
A Mechanised Logic of Computation

Proofs are terms of type `thm`, built using functions such as

$$\texttt{trans}\ (t_1 : \texttt{thm})\ (t_2 : \texttt{thm}) : \texttt{thm} =$$

if t_1 is "$A = B$" and t_2 is "$B = C$" then return "$A = C$" else fail

ML : the Meta Language of the LCF prover

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.

Polymorphism and type inference in ML

JOURNAL OF COMPUTER AND SYSTEM SCIENCES 17, 348–375 (1978)

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm $\mathscr{W}$ which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot "go wrong" and a Syntactic Soundness Theorem states that if $\mathscr{W}$ accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on $\mathscr{W}$ is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system.

Principal type-schemes for functional programs

Luis Damas* and Robin Milner
Edinburgh University

1. Introduction

This paper is concerned with the polymorphic type discipline of ML, which is a general purpose functional programming language, although it was first introduced as a metalanguage (whence its name) for conducting proofs in the LCF proof system [GMW]. The type discipline was studied in [Mil], where it was shown to be semantically sound, in a sense made precise below, but where one important question was left open: does the type-checking algorithm - or more precisely, the type assignment algorithm (since types are assigned by the compiler, and need not be mentioned by the programmer) - find the most general type possible for every expression and declaration? Here we answer the question in

of successful use of th
other research and in t
it has become important
particularly because th
(due to polymorphism),
soundness) and detectio
has proved to be one of

The discipline can
small example. Let us
"map", which maps a giv
- that is,

map f [x1;...;xn]

The required declaratio

letrec map f s = if nul
else c

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 $\alpha \to \alpha$ for some fixed, unknown type α.
- Milner : give a type schema $\forall\alpha.\ \alpha \to \alpha$ denoting a polymorphic function.

User-defined data structures in LCF ML

Built-in product types $t_1 \# t_2$ 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 $*$ at leaves.

```
absrectype * 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"...

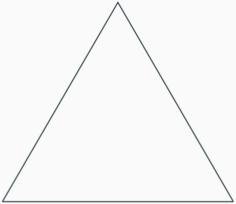
```
letrec sumtree t =
  if isleaf t then
    leafval t
  else
    sumtree (leftchild t)
    + sumtree (rightchild t)
```

. . . to Core ML

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

let rec sumtree t =
  match t with
  | Leaf n -> n
  | Node(l, r) -> sumtree l + sumtree r
```

Core ML

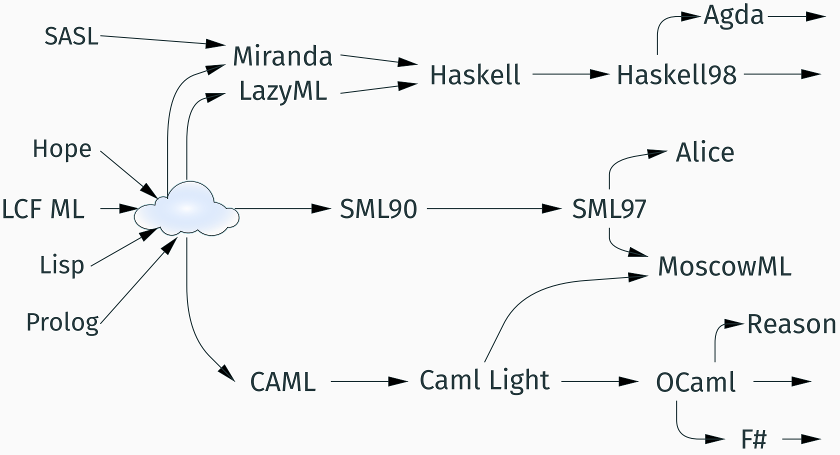
Hindley-Milner polymorphic typing
and type inference



Call-by-value
functional language

Inductive types,
pattern matching

A rich lineage



From CAML to Caml Special Light

CAML (1985–1994)

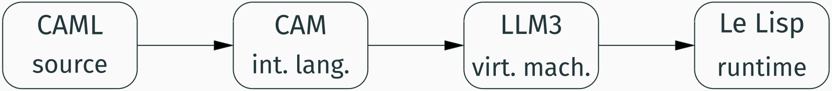
(G. Cousineau, G. Huet, M. Mauny, A. Suarez, P. Weis)

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



The Categorical Abstract Machine (G. Cousineau, P.-L. Curien, M. Mauny) : a simple evaluation model for call-by-value, inspired by cartesian closed categories.

$$[\![\, \underline{0} \,]\!] = \mathtt{snd} \qquad\qquad [\![\, \underline{n+1} \,]\!] = \mathtt{fst}; [\![\underline{n}]\!]$$

$$[\![\lambda.M]\!] = \mathtt{cur}([\![M]\!])$$

$$[\![M\ N]\!] = \mathtt{push}; [\![M]\!]; \mathtt{swap}; [\![N]\!]; \mathtt{cons}; \mathtt{app}$$

Pro : one of the first formalizations of function closures.

Cons : inefficient ; one “cons” for each binding.

A decisive encounter

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)

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39635511

Rapports Techniques

N°117

Programme 1
Calcul symbolique, Programmation et Génie logiciel

THE ZINC EXPERIMENT: AN ECONOMICAL IMPLEMENTATION OF THE ML LANGUAGE

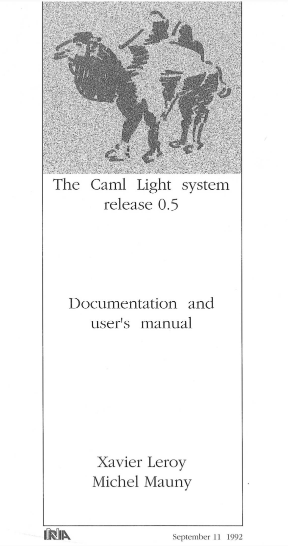
Xavier LEROY

Février 1990

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

Caml Light (1991–2000)

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

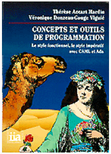


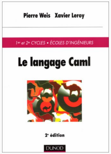
A completion of the ZINC experiment, practically usable, esp. for teaching.

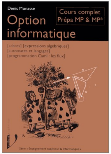
- Type checking and type inference.
- Separate compilation and linking.
- Modula-2 modules :
  implementation file (.ml)
  + interface file (.mli).
- Toplevel interactive REPL.
- Bootstrapped.
- Available for Unix, Mac OS, and MS-DOS !

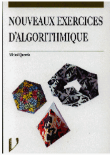
Caml Light in higher education

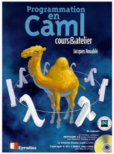
Many undergraduate CS courses used Caml Light, esp. in France.

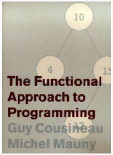












Studying the SML module language

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 ; …)

Type systems for module languages

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

Proc. 21st Symp. Principles of Programming Languages, 1994, pages 109–122.

Manifest types, modules, and separate compilation

Xavier Leroy *
Stanford University

Abstract

This paper presents a variant of the SML module system that introduces a strict distinction between abstract types and manifest types (types whose definitions are part of the module specification), while retaining most of the expressive power of the SML module system. The resulting module system provides much better support for separate compilation.

represent parameterized modules, and function applications to connect modules—all features that cannot be accounted for in the "modules as compilation units" approach.

As a consequence of this tension, SML makes no provision for separate compilation. SML is defined as "an interactive language" [17], implying that users are expected to build their programs linearly in strict bottom-up order. This requirement can be alleviated by systematic use of functors, at the cost of extra declarations (sharing constraints) and late detection of inter-compilation unit type clashes. Re-

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)

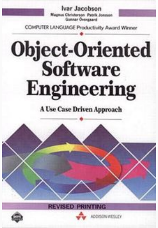
The language : the core Caml Light language + an SML-style module language using syntactic signatures and manifest types.

The implementation : the Caml Light runtime system
+ an improved ZAM2 bytecode compiler and interpreter
+ a native-code compiler.

```
From: Xavier Leroy <xleroy AT pauillac.inria.fr>
To: caml-list AT pauillac.inria.fr
Subject: Release 1.06 of Caml Special Light
Date: Tue, 12 Sep 1995 11:27:13 +0200 (MET DST)

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: [...]
```

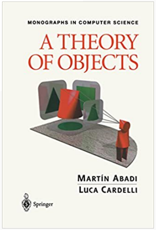
O(bjective) Caml

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.

$$
\begin{tikzcd}
 & \langle \mathtt{m} : \mathtt{int}; \rho_1 \rangle \arrow[dl] \arrow[dr] & & \langle \mathtt{k} : \mathtt{bool}; \rho_2 \rangle \arrow[dl] \\
\langle \mathtt{m} : \mathtt{int}; \emptyset \rangle & & \langle \mathtt{m} : \mathtt{int}; \mathtt{k} : \mathtt{bool}; \rho \rangle &
\end{tikzcd}
$$

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.

Objective Caml 1.00 (1996)

(J. Vouillon, D. Rémy)

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
```

From: Xavier Leroy <xleroy AT pauillac.inria.fr>
To: caml-list AT pauillac.inria.fr, comp-lang-ml AT cs.cmu.edu
Subject: Objective Caml 1.00
Date: Thu, 9 May 1996 16:27:36 +0200 (MET DST)

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)	Polymorphic recursion
3.12 (Aug 2010)	First-class modules
4.00 (Jul 2012)	Generalized Algebraic Datatypes (GADTs)

Labeled/optional arguments; extensible variants

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.

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

Polymorphic variants, to mix and match data constructors freely.

```
[‘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.

```
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)
```

Generalized Algebraic Data Types

Implemented by J. Le Normand, J. Garrigue, A. Frisch, based on ideas by many (see next slide).

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.

2007 GHC 6.8.1 : introduction of GADTs in Haskell.

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

A language that evolved gradually

> *Certainly, seen from 1996, the story [of Caml] could have been more linear.*
>
> *(Guy Cousineau, 1996)*

Seen from 2021, even more so!

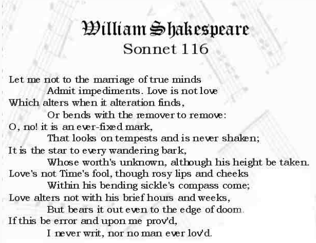
A language that is still faithful to its roots

Mostly functional (+ imperative and OO when needed).

Types as the skeleton of the language.

Devotion to type inference and existence of principal types.

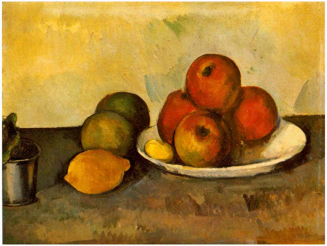
Beauty can come out of formal constraints



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.

FIGURE 4. *Opening of Fugue XXII from Part I of J.S. Bach's "The Well-Tempered Clavier."*



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!