The functional approach to programming and proving: An introduction to Caml and Coq

Xavier Leroy

INRIA Paris-Rocquencourt

DO178 ED12 working meeting, 2011-08-30

In this talk...

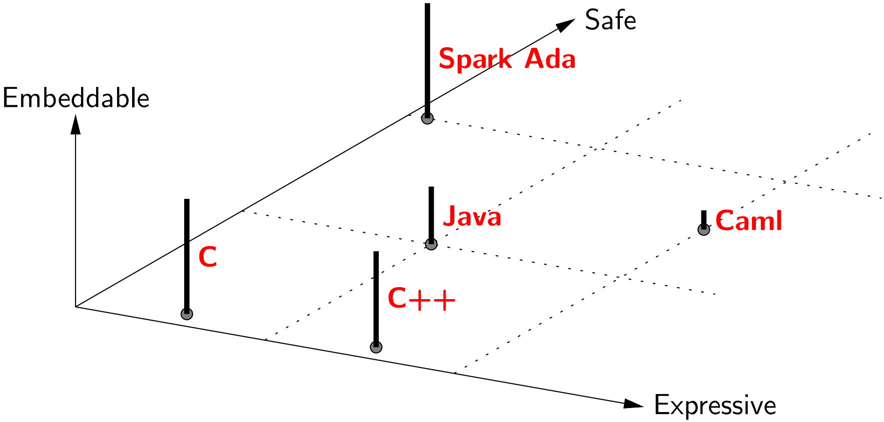
A programming language: Caml.

- Functional programming.
- Emphasis on safety and expressiveness.
- Especially good at symbolic computation.

Established uses in the aircraft industry:
implementation language for development and verification tools.

One of a family of typed functional languages: Caml, SML, Haskell, F#.

A landscape of some programming languages



In this talk. . .

A programming language: Caml.

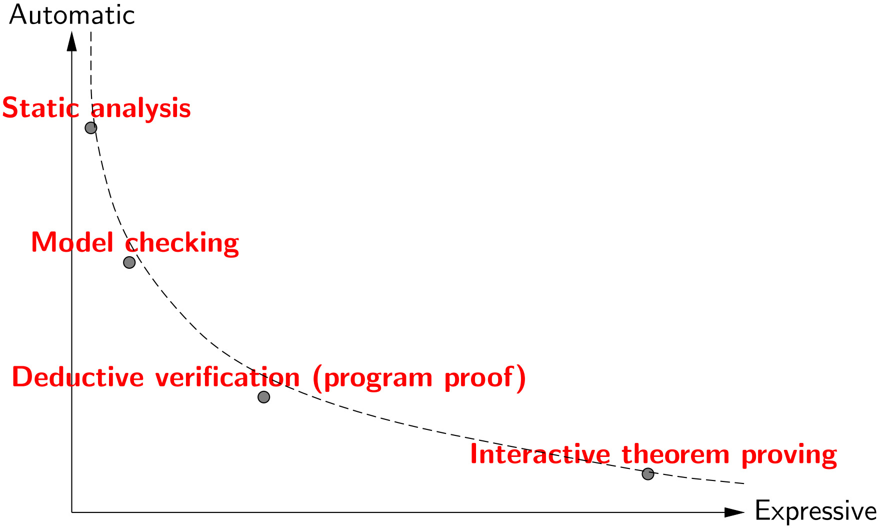
A proof assistant: Coq.

- Powerful specification language in mathematical logic.
- Interactive proof development & automatic proof checking.

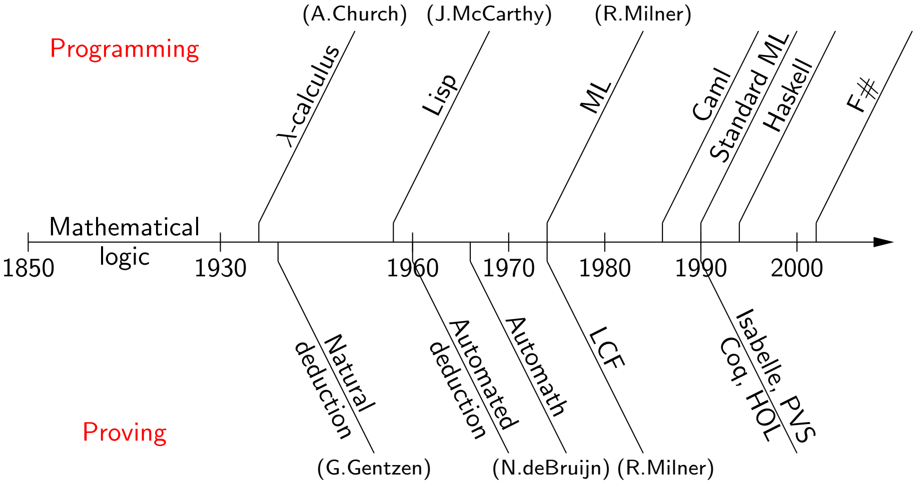
Potential uses in the aircraft industry:
formal verification of algorithms & difficult code fragments;
formal verification of code generation & verification tools.

One of a family of proof assistants: ACL2, Coq, HOL, Isabelle, PVS.

A landscape of some formal verification tools



A bit of history

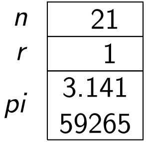


Part I

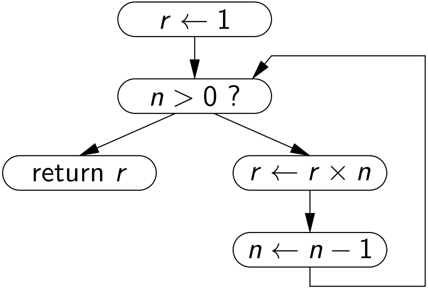
The Caml functional programming language

The imperative approach to programming

Program ≈ sequence of modifications to the machine state.



Memory



The imperative approach to programming

Program $\approx$ sequence of modifications to the machine state.

A view that is reflected by most existing programming languages:

```
int fact(int n)
{
    int r = 1;
    while (n > 0) { r = r * n; n = n - 1; }
    return r;
}
```

Plus: some encapsulation of these effects in functions or objects.

The functional approach to programming

(Peter J. Landin, *The next 700 programming languages*, Comm.ACM, 1966.)

Less emphasis on the **how?**
(how to chain computations? how to change memory state?)

Much more emphasis on the **what?**
(what is the expected final result? stay close to its mathematical definition!)

My first functional program

In mathematics:

$$0! = 1 \qquad\qquad n! = n \times (n-1)!$$

In Caml:

```
let rec fact n =
    if n = 0
    then 1
    else n * fact (n - 1)
```

Higher-order functions

Alternate mathematical definition:

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

i.e. reduce the sequence $1 \ldots n$ using $\times$.

In Caml:

```
let fact n = reduce (fun x y -> x * y) 1 (sequence 1 n)
```

Note: 1st parameter to `reduce` is a function (`fun x y -> x * y`), so `reduce` is a "higher-order" function.

Higher-order functions

The `reduce` h-o function is defined as:

```
let rec reduce fn neutral list =
    match list with
    | [] -> neutral
    | head :: tail -> fn head (reduce fn neutral tail)
```

Note: pattern-matching over a list, with two branches empty / nonempty.

`sequence` l h is the list of integers $l; l+1; \ldots; h$.

```
let rec sequence low high =
    if low > high
    then []
    else low :: sequence (low + 1) high
```

Composing & reusing functions

What if we need power series as well?

$$1^k + 2^k + \cdots + (n-1)^k + n^k$$

```
let powerseries n k =
    reduce (fun x y -> power x k + y) 0 (sequence 1 n)
```

Note free variable `k` in `fun x y -> power x k + y`
which is bound by the 2nd parameter of `powerseries`.

(Functions as first-class values, also known as *closures*.)

Symbolic computation

Computing over complex tree-shaped data structures,
with little, if any, arithmetic.

Typical examples:

- Formulas in spreadsheets, computer algebra systems, theorem provers.
- Abstract syntax trees in interpreters, compilers, static analyzers, ...
- Semi-structured data (XML, HTML) on the Web, in structured documents, ...

A killer combination:
inductive data types + pattern-matching + recursion.

An example of symbolic computation: formal derivatives

```
type expression =
   | Const of float
   | Var of string
   | Sum of expression * expression    (* e1 + e2 *)
   | Diff of expression * expression   (* e1 - e2 *)
   | Prod of expression * expression   (* e1 * e2 *)
   | Quot of expression * expression   (* e1 / e2 *)

let rec deriv exp dv =
  match exp with
   | Const c -> Const 0.0
   | Var v -> if v = dv then Const 1.0 else Const 0.0
   | Sum(f, g)  -> Sum(deriv f dv, deriv g dv)
   | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
   | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
   | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g),
                             Prod(f, deriv g dv)),
                        Prod(g, g))
```

An example of symbolic computation: formal derivatives

The same computation in a conventional imperative language (C):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct expr {
  enum { CONST, VAR, SUM, DIFF, PROD, QUOT } kind;
  union {
    double cst;
    char * varname;
    struct expr * t[2];
  } u;
} * expr;

expr alloc_expr(void) {
  expr res = malloc(sizeof(struct expr));
  if (res == NULL) { fprintf(stderr, "Out of memory.\n"); exit(2); }
  return res;
}
```

An example of symbolic computation: formal derivatives

```
expr deriv(expr t, char * dx)
{
  expr dt = alloc_expr();
  switch (t->kind) {
  case CONST:
    dt->kind = CONST; dt->u.cst = 0; break;
  case VAR:
    dt->kind = CONST;
    dt->u.cst = strcmp(dt->u.varname, dx) == 0 ? 1 : 0;
    break;
  case SUM:
  case DIFF:
    dt->kind = t->kind;
    dt->u.t[0] = deriv(t->u.t[0], dx);
    dt->u.t[1] = deriv(t->u.t[1], dx);
    break;
```

An example of symbolic computation: formal derivatives

```
case PROD: {
  expr dt1 = alloc_expr();
  expr dt2 = alloc_expr();
  dt1->kind = PROD;
  dt1->u.t[0] = t->u.t[0];
  dt1->u.t[1] = deriv(t->u.t[1], dx);
  dt2->kind = PROD;
  dt2->u.t[0] = deriv(t->u.t[0], dx);
  dt2->u.t[1] = t->u.t[1];
  dt->kind = SUM;
  dt->u.t[0] = dt1; dt->u.t[1] = dt2;
  break; }
```

An example of symbolic computation: formal derivatives

```
  case QUOT: {
    expr dt1 = alloc_expr();
    expr dt2 = alloc_expr();
    expr dt3 = alloc_expr();
    expr dt4 = alloc_expr();
    dt1->kind = PROD;
    dt1->u.t[0] = deriv(t->u.t[0], dx);
    dt1->u.t[1] = t->u.t[1];
    dt2->kind = PROD;
    dt2->u.t[0] = t->u.t[0];
    dt2->u.t[1] = deriv(t->u.t[1], dx);
    dt3->kind = DIFF; dt3->u.t[0] = dt1; dt3->u.t[1] = dt2;
    dt4->kind = PROD;
    dt4->u.t[0] = t->u.t[1]; dt4->u.t[1] = t->u.t[1];
    dt->kind = QUOT; dt->u.t[0] = dt3; dt->u.t[1] = dt4;
    break; }
  }
  return dt;
}
```

Strong static typing

Strong typing prevents the program from applying operations to data on which they are not defined.

`sin(3.14)`	ok
`sin("hello")`	error
`sin(3.14, 2.718)`	error
`"hello"(3.14)`	error

Implies: no casts between integers and pointers; array bound checks; automatic memory management; ...

- **Dynamic typing**: checked at run-time (Lisp, Perl, Python, ...)
- **Static typing**: checked as much as possible at compile-time. Finds errors earlier; saves on testing. (Java, ML, Caml, Haskell, ...)

Strong static typing

Enables early, automatic detection of common programming errors.

```
let rec deriv exp dv =
   match exp with
   | Const c    -> 0.0
   | Var v      -> if v = dv then Const 1.0 else Const 0.0
   | Sum(f, g)  -> Sum(deriv f dv, deriv dv g)
   | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
   | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
(* no case for Quot *)
```

Applicable to most programming paradigms, but especially effective in functional programming.

Type inference

(or: strong static typing without cluttering the source with type declarations)

Most types are automatically inferred by the compiler, without requiring the programmer to declare types.

```
let rec fact n =
    if n = 0 then 1 else n * fact (n - 1)
```

Inferred type: `int` $\rightarrow$ `int` (because 0, 1 are of type `int`).

```
let rec deriv exp dv = ...
```

Inferred type: `expression` $\rightarrow$ `string` $\rightarrow$ `expression`

Parametric polymorphism

What if the context doesn't determine types uniquely?

```
let rec reduce fn neutral list =
    match list with
    | [] -> neutral
    | head :: tail -> fn head (reduce fn neutral tail)
```

The compiler infers a polymorphic (generic) type:

$$\texttt{reduce} : \forall \alpha, \beta.\ \ (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha\ \texttt{list} \rightarrow \beta$$

The `reduce` function can, then, safely be used at any instance of its type:

```
let fact n = reduce (fun x y -> x * y) 1 (sequence 1 n)
```

$\alpha = \beta = \mathit{int}$

```
let concat l = reduce (fun x y -> x ^ " " ^ y) "" l
```

$\alpha = \beta = \mathit{string}$

Wrapping up

Some other notable features of Caml:

- Full support for imperative programming
  (exceptions, input/output, mutable arrays, mutable references, ...)
- A powerful module system
  (supporting type abstraction and parameterized modules)
- A rich (but temperamental) object and class layer
  (with multiple inheritance and independent subtyping).

Some uses of Caml

In the avionics industry:

- The Scade KCG 6 code generator (qualified DO178-A).
- The Astrée static analyzer.
- The Frama-C static analyzer and deductive program verifier.
- The Alt-Ergo automated theorem prover.

Some uses of Caml

Main application areas:

- Languages: compilers, domain-specific languages
  (e.g. CellControl, part of Dassault Systèmes's Delmia)
- Verification: theorem provers, static analyzers
  (e.g. Microsoft's Static Driver Verifier)
- Finance: pricing, trading
  (e.g. Jane Street Capital)
- Systems administration
  (e.g. Citrix's administration tools for the Xen hypervisor)
- Network programming
  (e.g. the Unison file synchronization tool)
- Web programming
  (e.g. the Wink.com search engine)

The OCaml system and community

Free software.

Windows, MacOS X, all Linux distributions

Developed at INRIA since 1995. 37 releases since.

300 user contributions: libraries, tutorials, packaging for distros, ...

18 books
Widely taught in France, but also in the US and Japan.

The Caml Consortium: 12 partners supporting the development
(incl. Microsoft, Dassault Aviation, Dassault Systèmes, CEA, and Esterel Technologies)

For more information…

`http://caml.inria.fr/`

Part II

The Coq proof assistant

The Coq proof assistant

A tool to write specifications and conduct proofs in the Calculus of Inductive Constructions (a typed, constructive logic).

- A powerful specification language (Gallina).
- Commands ("tactics") to develop proofs in interaction with the tool.
- Produces proof terms that are re-checked by a small, trustworthy kernel.

A glance at Gallina

1. Functions defined by recursion & pattern-matching.

```
Fixpoint fact (n: nat) : nat :=
    match n with
    | O => 1
    | S p => n * fact p
    end.
```

Unlike in Caml, functions must be pure and terminating.

A glance at Gallina

1. **Functions**

2. **Predicate logic** with the usual connectives & quantifiers.

```
Theorem fact_multiple_6:
    forall n, n >= 3 -> exists m, fact n = 6 * m.
```

A glance at Gallina

1. **Functions**

2. **Predicate logic**

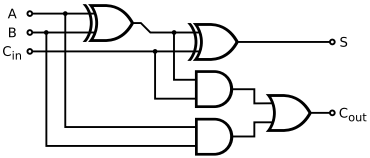
3. **Inductive predicates** (≈ inference rules in C.S., ≈ Prolog programs.)

```
Inductive even: nat -> Prop :=
  | even_zero:
      even 0
  | even_plus2:
      forall n, even n -> even (n + 2).
```

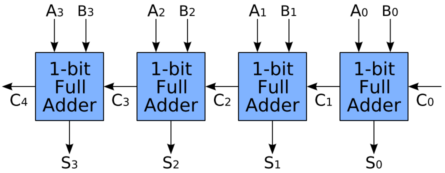
Defines `even` as the minimal predicate such that
`even` n iff `n` $= 0$ or ($n = m + 2$ and `even` m).

Example 1: boolean circuits

Full adder



4-bit ripple carry adder



Do these circuits really compute the + operation?

→ Coq development.

(`http://gallium.inria.fr/~xleroy/talks/caml+coq/Adders.html`)

Example 2: compilation to Reverse Polish Notation

Algebraic notation: $2 \times (3 + 4)$

$\downarrow$

R.P. notation: $2 \;\; 3 \;\; 4 \;\; + \;\; \times$

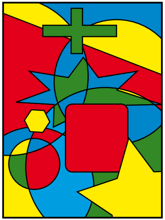
Why is it that both notations compute the same results?

$\rightarrow$ Coq development.

(`http://gallium.inria.fr/~xleroy/talks/caml+coq/RPN.html`)

Some major Coq projects — Higher mathematics

First fully formal proof of the four-color theorem.
(Gonthier & Werner, 2005)



Using a related proof assistant (HOL-light):

Formal verification of Hales's proof of Kepler's conjecture.
(Flyspeck project.)

Some major Coq projects — Software verification

Common Criteria EAL7 certification of a
Java Card implementation:
virtual machine, firewall, security API.
(B. Chetali et al, Gemalto, 2007.)

CompCert: a formally verified, realistic compiler
from C to PowerPC, ARM and x86.
(X. Leroy, S. Blazy, 2006–2011)
(`http://compcert.inria.fr/`)

Using a related proof assistant (Isabelle/HOL):

seL4, a formally-verified secure micro-kernel.
(G. Klein et al, NICTA, 2009; Open Kernel Labs & Galois)

The Coq system and community

Free software.

Windows, MacOS X, all Linux distributions

Developed at INRIA since 1989. 18 major releases since.

3 textbooks.

Increasingly taught in the US (U. Penn, Harvard, Princeton, UCSD, ...)

For more information…

`http://coq.inria.fr/`

`http://caml.inria.fr/`