From ML to program proof

or: The continuation of functional programming by other means

Xavier Leroy

INRIA Paris-Rocquencourt

Milner symposium, 2012-04-17

In the beginning...

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.

(POPL 1982)

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,

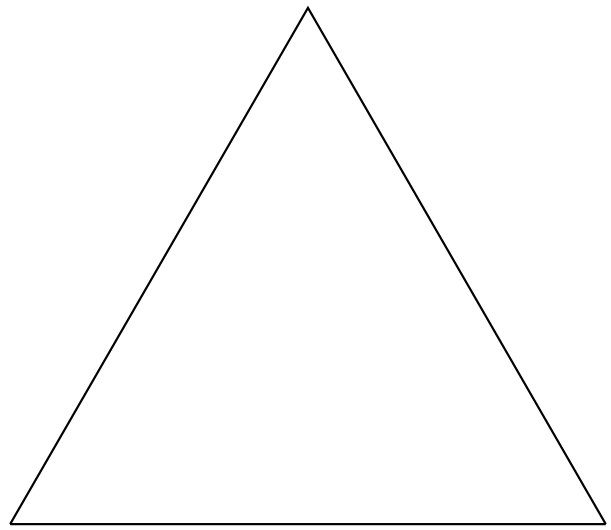
map f [x1;...;xn]

The required declaratio

letrec map f s = if nul
else c

Core ML

Hindley-Milner polymorphic types
Damas-Milner type inference



First-class
functions

Datatypes and
pattern-matching

Things we learned from Core ML

Strong static typing is the programmer's friend.

Types need not be verbose.

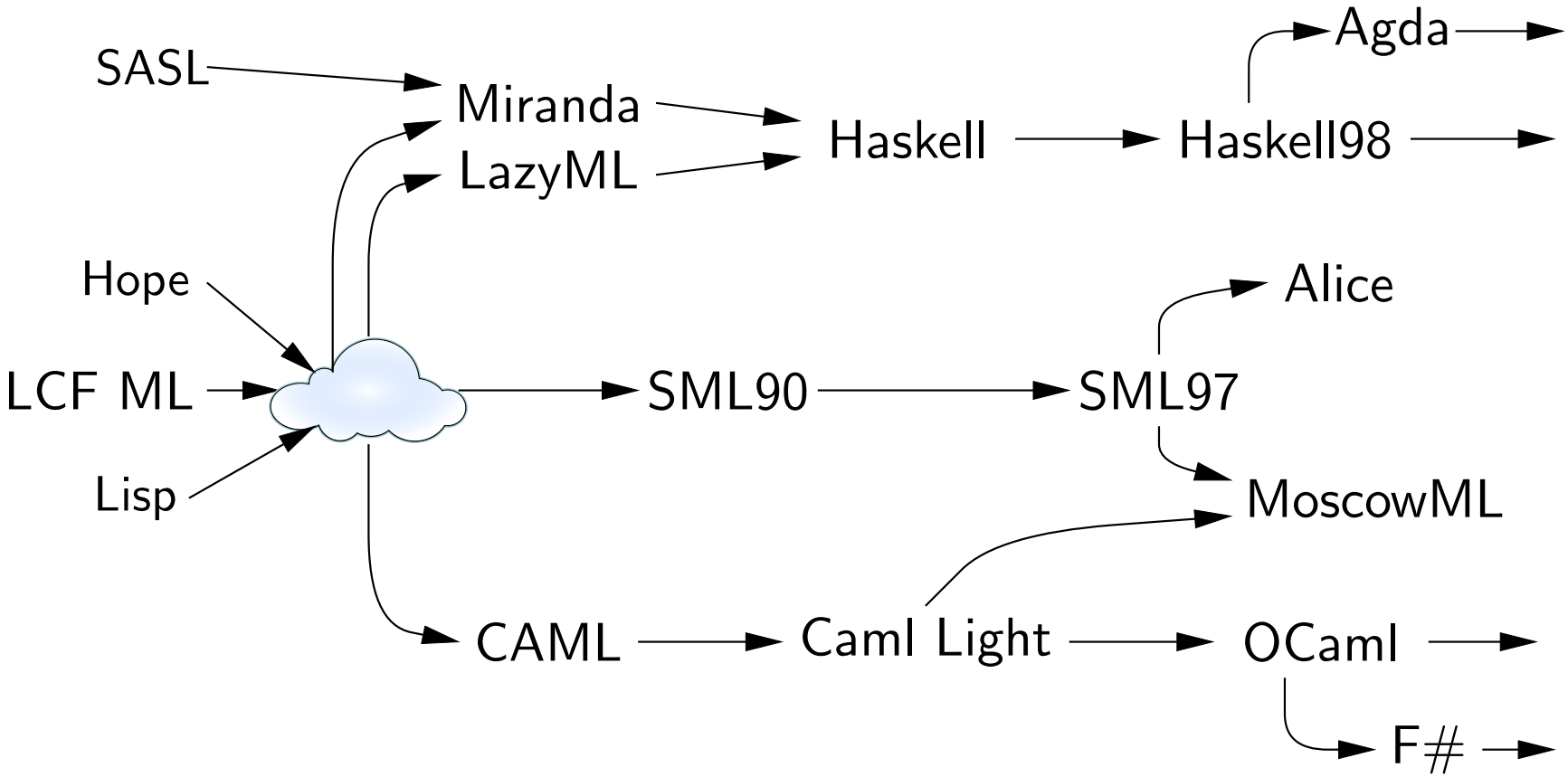
Explicit types as documentation (datatypes, interfaces).

Types are compatible with code reuse.

Opportunities for reuse can be discovered rather than planned.

Not just type safety, but also exhaustiveness checks.

A rich lineage

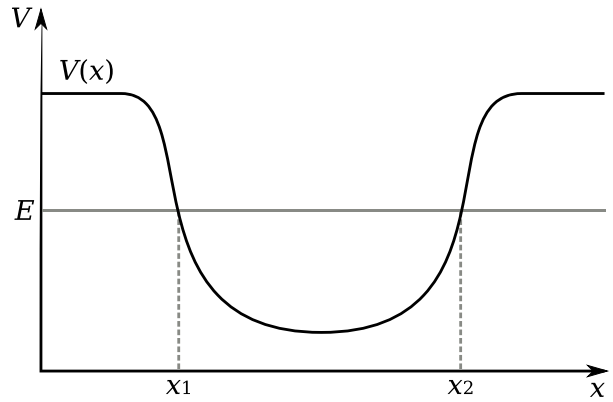


1985–2000: a flurry of type systems

Type more features; type them more precisely.

Type other programming paradigms (OO, distributed).

Typed intermediate & assembly languages.



(Core ML as the bottom of a potential well.)

Principal types or not?

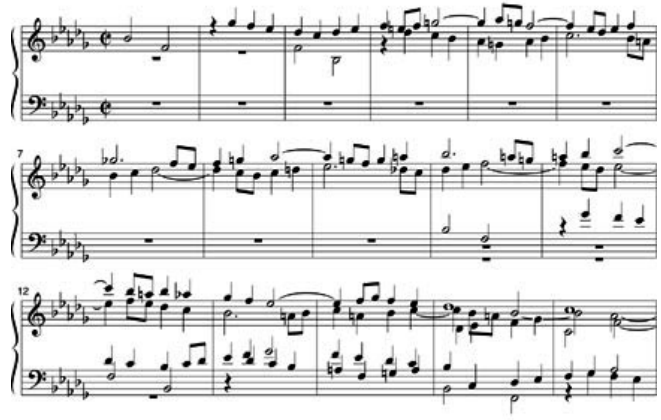
Beauty can also arise from formal constraints. . .

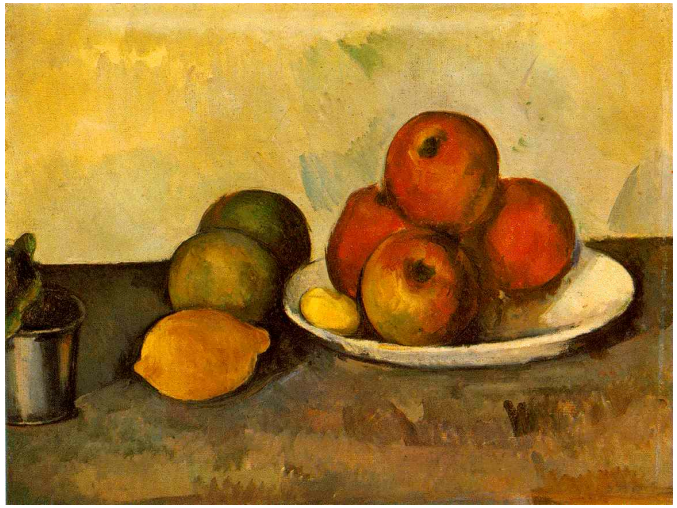
William Shakespeare

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."*





Novel approaches driven by principality

Exhibit A: Haskell's type classes.

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them formally by means of type inference rules.

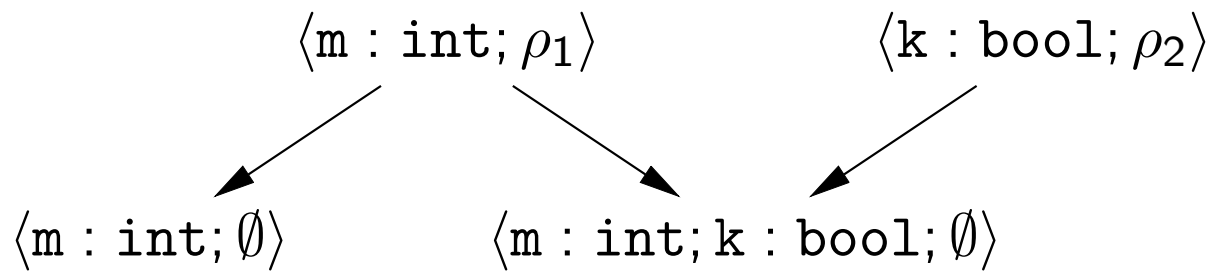
ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

The type system presented here is a generalisation of the Hindley/Milner type system. As in that system, type declarations can be inferred, so explicit type declarations for functions are not required. During the inference process, it is possible to translate a

Novel approaches driven by principality

Exhibit B: row polymorphism (Wand, Rémy, Garrigue)



Pitfalls

Multiple sub-languages:

core + modules + OO classes + type classes + ...

Intractable type expressions.

Diminishing returns.

Complicated encodings:

phantom types, GADTs, ...

2000–now: towards program verification

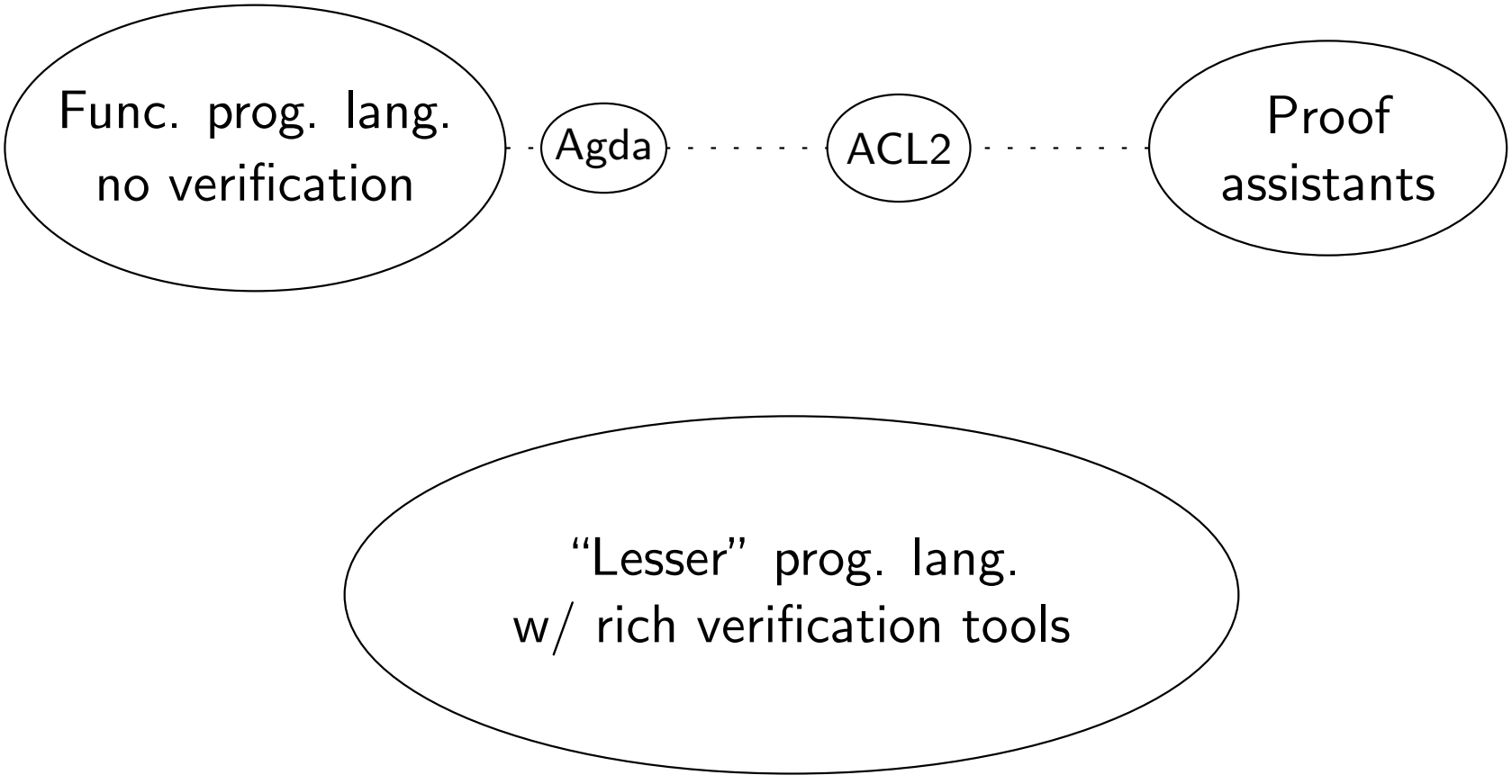
(for security and safety)

A big chunk of the P.L. community shifts to verification
(static analysis, model checking, program proof, combinations thereof).

C	Astree, BLAST, CBMC, Coverity, Fluctuat, Frama-C, Polyspace, SLAM, VCC, ...
Java, C#	Bandera, CodeContracts, Coverity, ESC/Java2, Java Pathfinder, KeY, Klocwork, Spec#, ...
F.P.	F*

Another chunk (re-)discovers LCF-style interactive proof assistants
(the POPLmark challenge, etc).

A fragmented landscape



How to develop and formally verify a program well-suited to F.P.?
(running example: a compiler)

Early compiler verification in Stanford LCF

(*Machine Intelligence*, 1972)

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch

Computer Science Department
Stanford University

Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

Early compiler verification in Stanford LCF

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp.iswfse e⊃MT(compe e,sp)≡svof(sp)∣((MSE(e,svof sp))&pdof(sp)),
     ∀e.iswfse e⊃iswft(compe e)≡TT,
     ∀e.iswfse e⊃(count(compe e)=0)≡TT;

TRY 2 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfsefun(f,e);
  LABEL TT;
   TRY 1 CASES type e=_N;
    TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES type e=_E;
     TRY 1 SUBST ,FCOMPE;
     SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
     INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
      TRY 1 CONJ;
       TRY 1 SIMPL;
        TRY 1 USE COUNT1;
         TRY 1;
         APPL ,INDHYP+2,arg1of e;
         LABEL CARG1;
         SIMPL-;QED;
         TRY 2 USE COUNT1;
          TRY 1;
```

Compiler verification

Theorem (Semantic preservation)

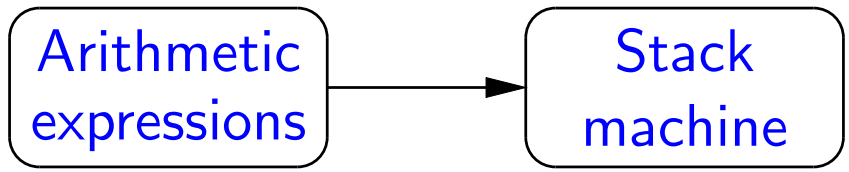
Let S be a source program and E an executable.
Assume the compiler produces E from S,
without reporting compile-time errors.
Then, for all observable behaviors b, $E \Downarrow b \Longrightarrow S \Downarrow b$

A challenge for standard deductive program provers:

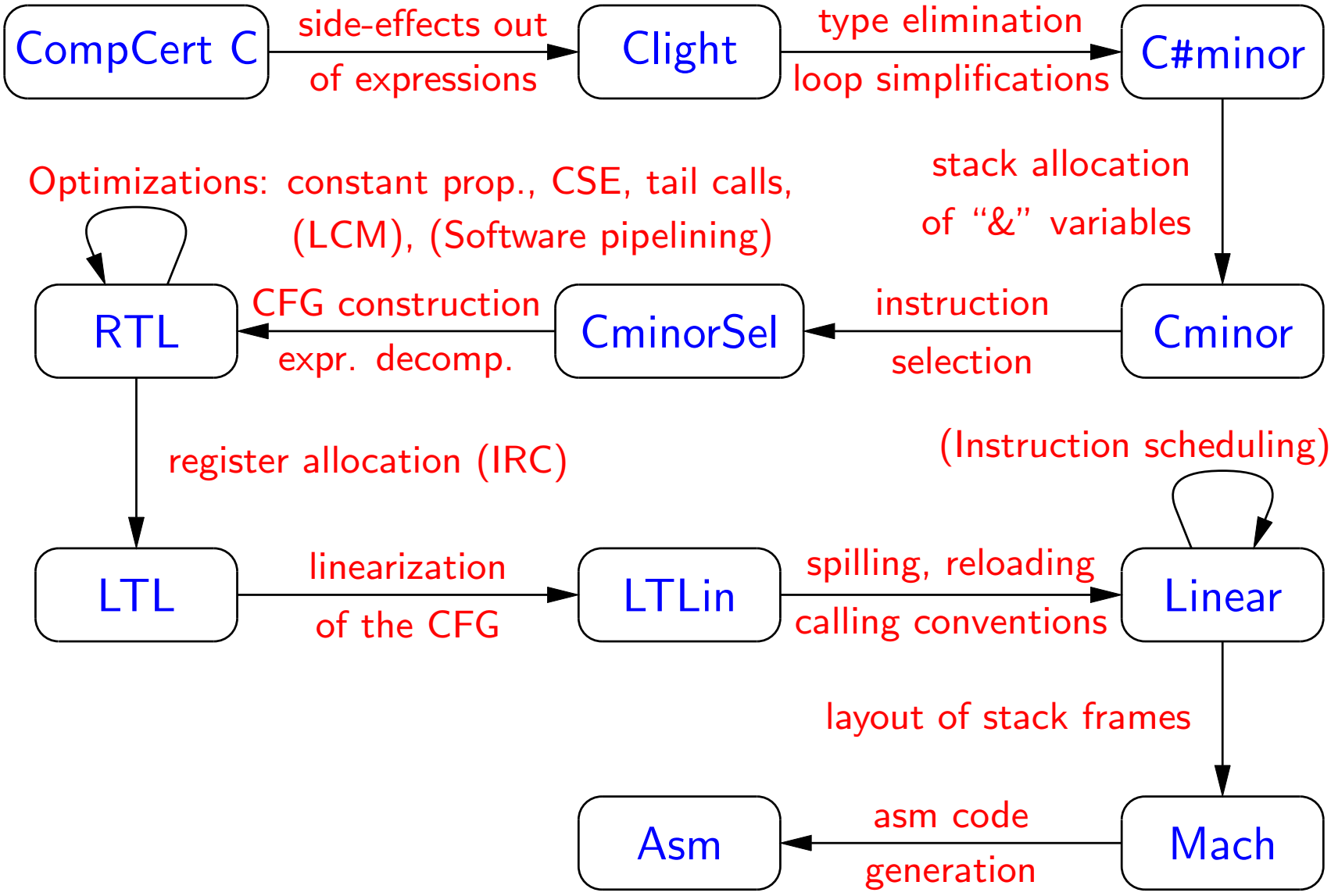
- The specification involves complex logical definitions (two operational semantics).
- The program involves recursion and tree/graph-shaped data.

A good match for interactive theorem provers.

From Milner & Weyrauch...



. . . to the CompCert C verified compiler



Using the Coq proof assistant

1. To write the specification and conduct the proof.
   (50 000 lines; 4 person.years)

2. To program the compiler.
   In pure, strict, terminating functional style.
   Executability via Coq $\rightarrow$ OCaml extraction.

Programming a compiler pass in Coq

```
Fixpoint transl_expr (map: mapping) (a: expr) (rd: reg) (nd: node)
                     {struct a}: mon node :=
  match a with
  | Evar v =>
      do r <- find_var map v; add_move r rd nd
  | Eop op al =>
      do rl <- alloc_regs map al;
      do no <- add_instr (Iop op rl rd nd);
      transl_exprlist map al rl no
  | Eload chunk addr al =>
      do rl <- alloc_regs map al;
      do no <- add_instr (Iload chunk addr rl rd nd);
         transl_exprlist map al rl no
  | ...
```

Programming limitations

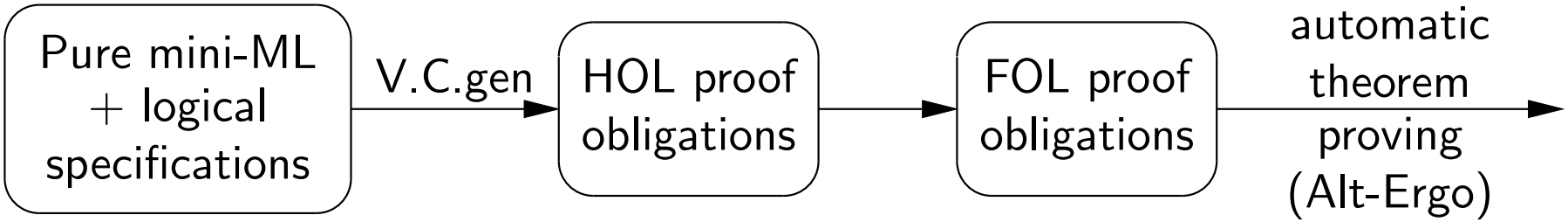
In exchange for powerful proof support, we must deal with

- Strictness: no problem
- Purity: use monads for state and errors.
- Termination: often a problem.
  Use "fuel" to bound recursions.
  Irritating issue: we're only interested in partial correctness.

Is there a better way?

The Pangolin system

(Y. Régis-Gianas, F. Pottier)



How to specify an argument $f : A \to B$ to a higher-order function?

- Not by a function of the logic (non-termination, ...)
- But by a pair of predicates:

$$\begin{array}{lcl} \texttt{Pre}(f) & : & A \to \texttt{Prop} \\ \texttt{Post}(f) & : & A \times B \to \texttt{Prop} \end{array}$$

Example of specification

```
let rec add (x, a) where (bst(a) and avl(a))
returns b where
          (bst(b) and avl(b) and
           elements (b) === singleton (x) ∪ elements (a)) =
  match a with
  | Empty ->
      Node (Empty, x, Empty)
  | Node (l, y, r) ->
      if x = y then a
      else if x < y then bal (add (x, l), y, r)
      else bal (l, y, add (x, r))
  end
```

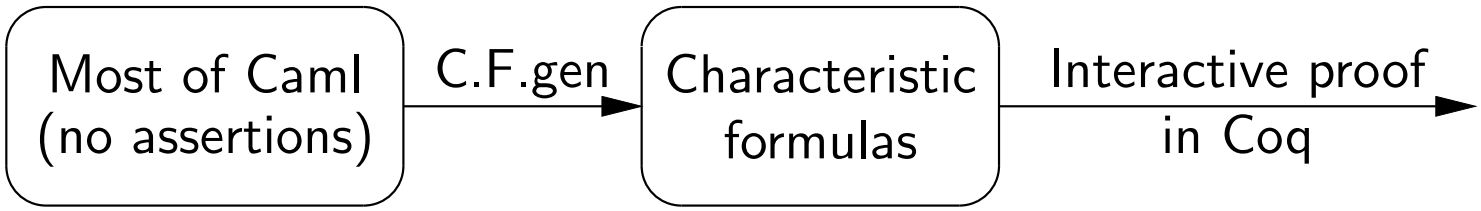
Proof obligations automatically discharged by Alt-Ergo.

Example of higher-order specification

```
let rec map (f, a) where
   (bst(a) and avl(a)
    and ∀ x ∈ elements(a), Pre(f)(x))
returns b where
   (bst(b) and avl(b)
    and ∀ x ∈ elements(a), ∃ y ∈ b, Post(f)(x,y)
    and ∀ y ∈ elements(b), ∃ x ∈ a, Post(f)(x,y))
=
   ...
```

The CFML system

(A. Charguéraud)



The characteristic formula $[[t]]$ of a term t is the HO predicate s.t.

$$\forall P, Q, \quad [[t]]\ P\ Q \Longleftrightarrow \{P\}\ t\ \{Q\} \text{ (in Hoare logic)}$$

($\approx$ weakest precondition / denotational semantics / deep embedding.)

The characteristic formula $[[t]]$ follows exactly the structure of t
$\rightarrow$ lends itself well to interactive proof.

In closing...

Pure, strict functional programming is a very short path from a program to its correctness proof.

Contemporary F.P. languages do not realize this potential.

Axiomatic semantics is not just for imperative languages!

Shall we just embrace and improve proof assistants as P.L.?

Or design F.P.L. with verifiability in mind?