From functional programming to program proof

or: The continuation of ML by other means

Xavier Leroy

INRIA Paris-Rocquencourt

Journées d'Informatique Fondamentale de Paris Diderot,
2013-04-26

Part I

The legacy of ML

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,

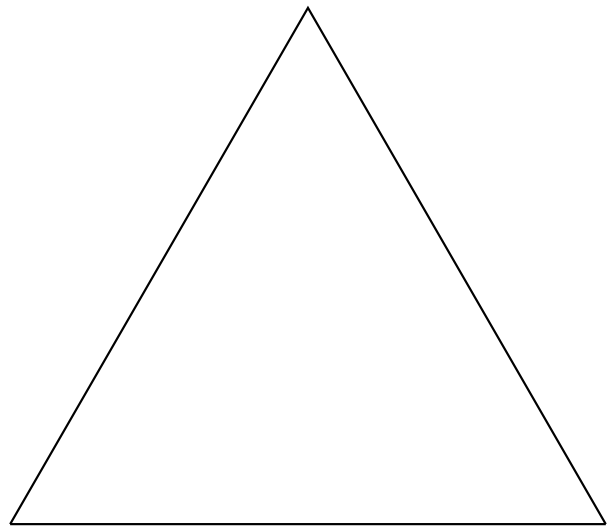
map f [x1;...;xn]

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.

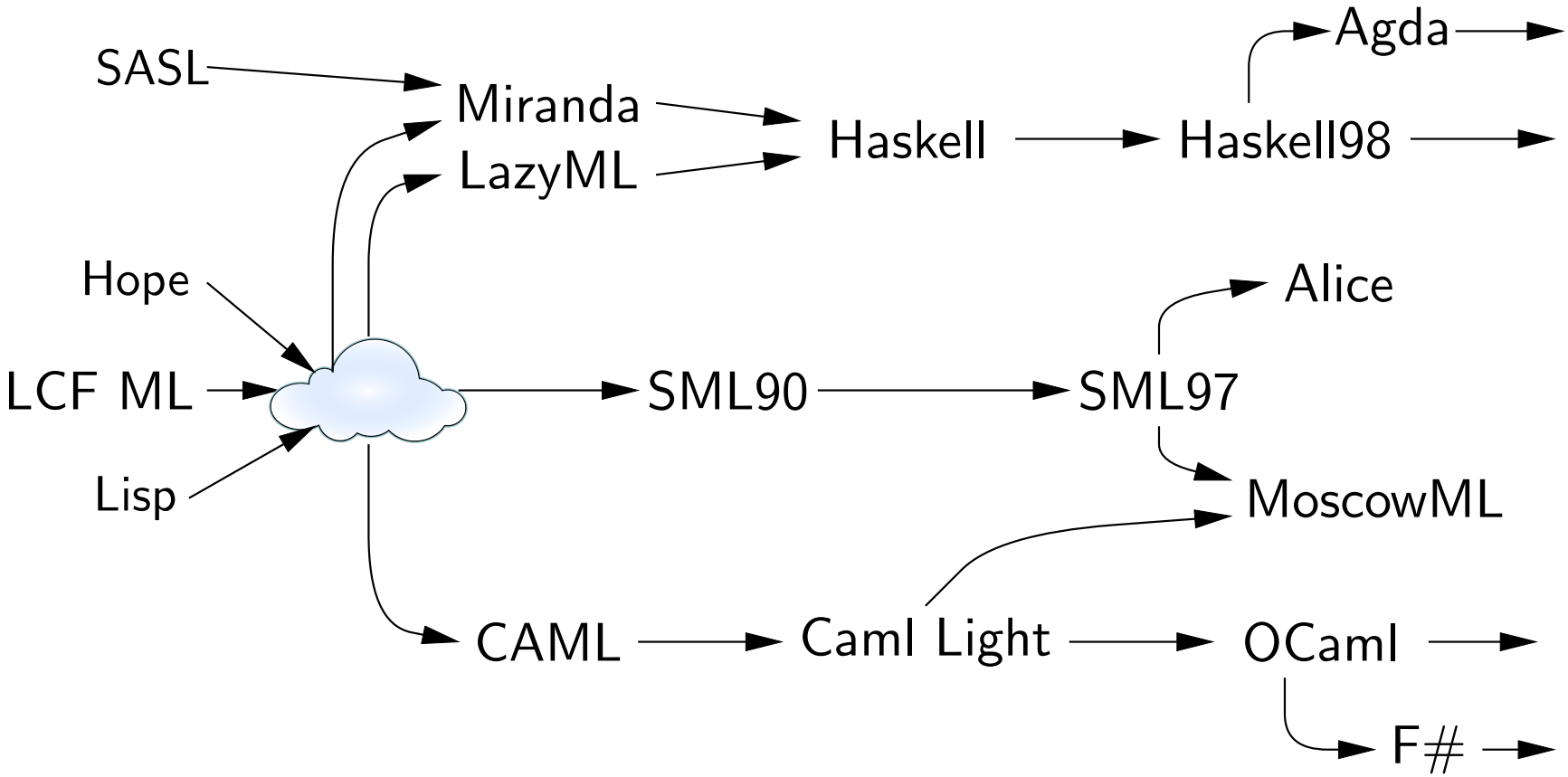
Explicit types as documentation (datatypes, interfaces).

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

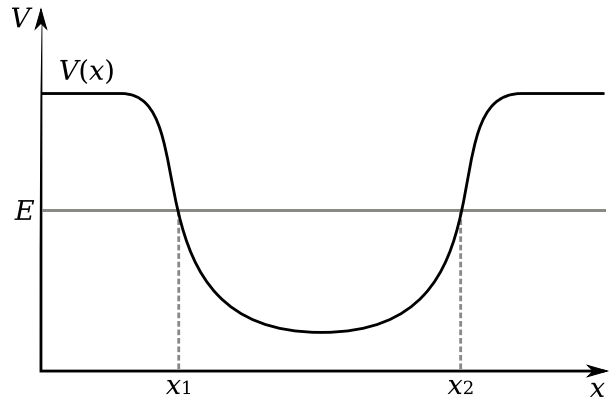


1980–2000: a flurry of type systems

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?

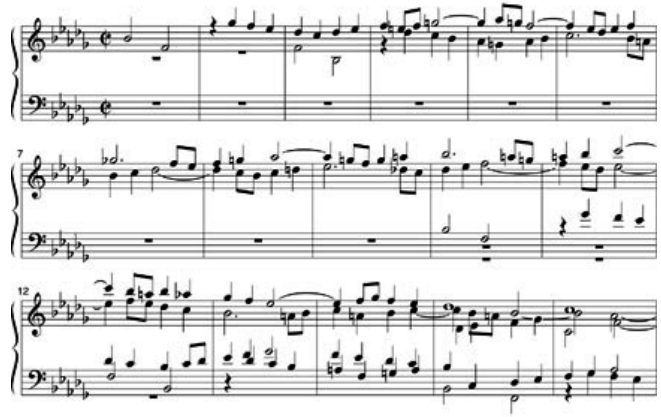
Beauty can also arise from formal constraints. . .

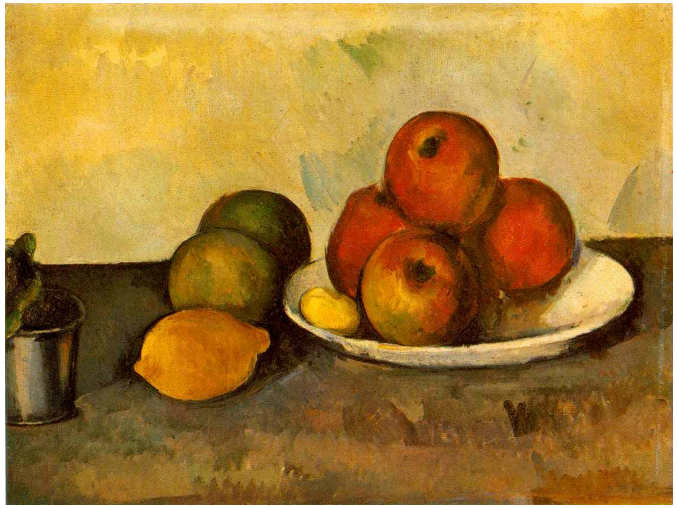
William Shakespeare

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.

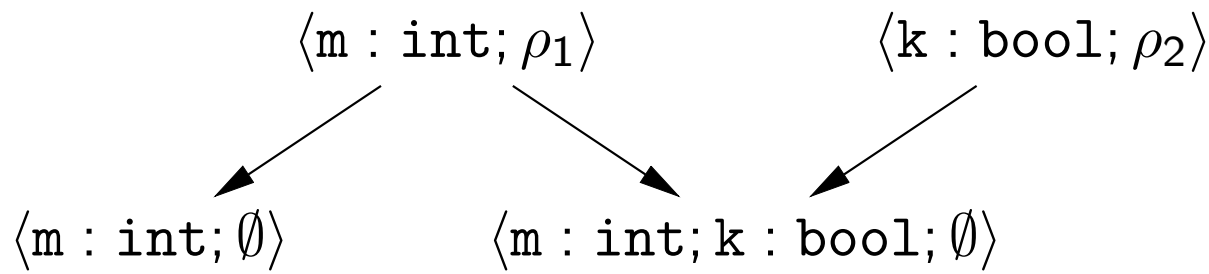
ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

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

Independently-developed advanced type systems that don't combine well in one language.

Multiple sub-languages:

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

Intractable type expressions.

Diminishing returns.

Complicated encodings:

phantom types, GADTs, ...

1990–2000: making pure functional programming work

> *[Standard ML] grew in response to a particular programming task, for which it was equipped also with full imperastive power, and a sophisticated exception mechanism.*
>
> *(R. Milner et al, The Definition of SML)*

Pragmatic reasons for "impure" functional programming:

- Efficient algorithms using in-place modification (e.g. unification)
- Basic software engineering:
  - error reporting
  - logging, I/O, ...
  - configuration variables, "gensym", ...

Paradise lost?

With imperative features come difficult metatheory:

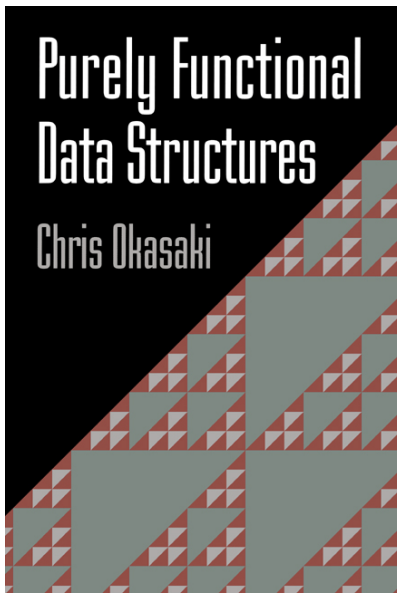
- STLC + control operators $\approx$ classical logic
- Hindley-Milner + references $\approx$ communication channels

and great responsibilities on the programmers:

- equational reasoning is generally false;
- missing cases ("uncaught exception");
- sharing of data structures is no longer always safe.

Paradise regained, I: functional data structures

C. Okasaki, R. Bird, R. Hinze, J.-C. Filliâtre, ...



Building a catalogue of pure, persistent data structures:

- with only $O(\log n)$ slowdowns
  (balanced binary trees, Patricia trees, ...)
- or even $O(1)$ amortized slowdowns
  (by clever use of lazy evaluation)

Paradise regained, II: monads

The essence of functional programming
(Invited talk)

Philip Wadler, University of Glasgow*

Abstract

This paper explores the use monads to structure functional programs. No prior knowledge of monads or category theory is required.

A Modular Approach to Denotational Semantics

Eugenio Moggi
Dipartimento di Matematica
Universitá di Genova
via L.B. Alberti 4
16132 Genova, ITALY
moggi@igecuniv.bitnet

We propose an incremental approach to the denotational semantics of complex programming languages based on the idea of *monad transformer*.

A monad = a parameterized type α `mon` and operations

$$\begin{aligned} \texttt{ret} &: \forall\alpha,\ \alpha \to \alpha\ \texttt{mon} \\ \texttt{bind} &: \forall\alpha\beta, \alpha\ \texttt{mon} \to (\alpha \to \beta\ \texttt{mon}) \to \beta\ \texttt{mon} \end{aligned}$$

such that `bind`(`ret` x) $f \approx f\ x$ and `bind` $x\ f \approx$ `bind` $y\ f$ if $x \approx y$.

Paradise regained, II: monads

All classic imperative features can be uniformly presented as monadic programming: state, exceptions, continuations.

Not a silver bullet: if all your code is in the state-and-exception monads, it has the same problems as ML code.

A way to segregate (by typing) pure computations from effectful computations.

A way to restrict effects to just what is needed by the program:

- just logging or just configuration passing or just "gensym" instead of full state;
- just backtracking or just coroutining instead of full control operators.

Limitations

The check is in the mail.

Next year unemployment will go down.

If it typechecks, it is correct.

Reality check: even after leveraging static typing and exploiting purity as much as possible, functional programs still have serious bugs, esp.

- regarding numerical computations
- and data structure invariants (beyond basic integrity).

Part II

Towards software verification

2000–now: towards program verification

(for security and safety)

A big chunk of the P.L. community shifts to verification:

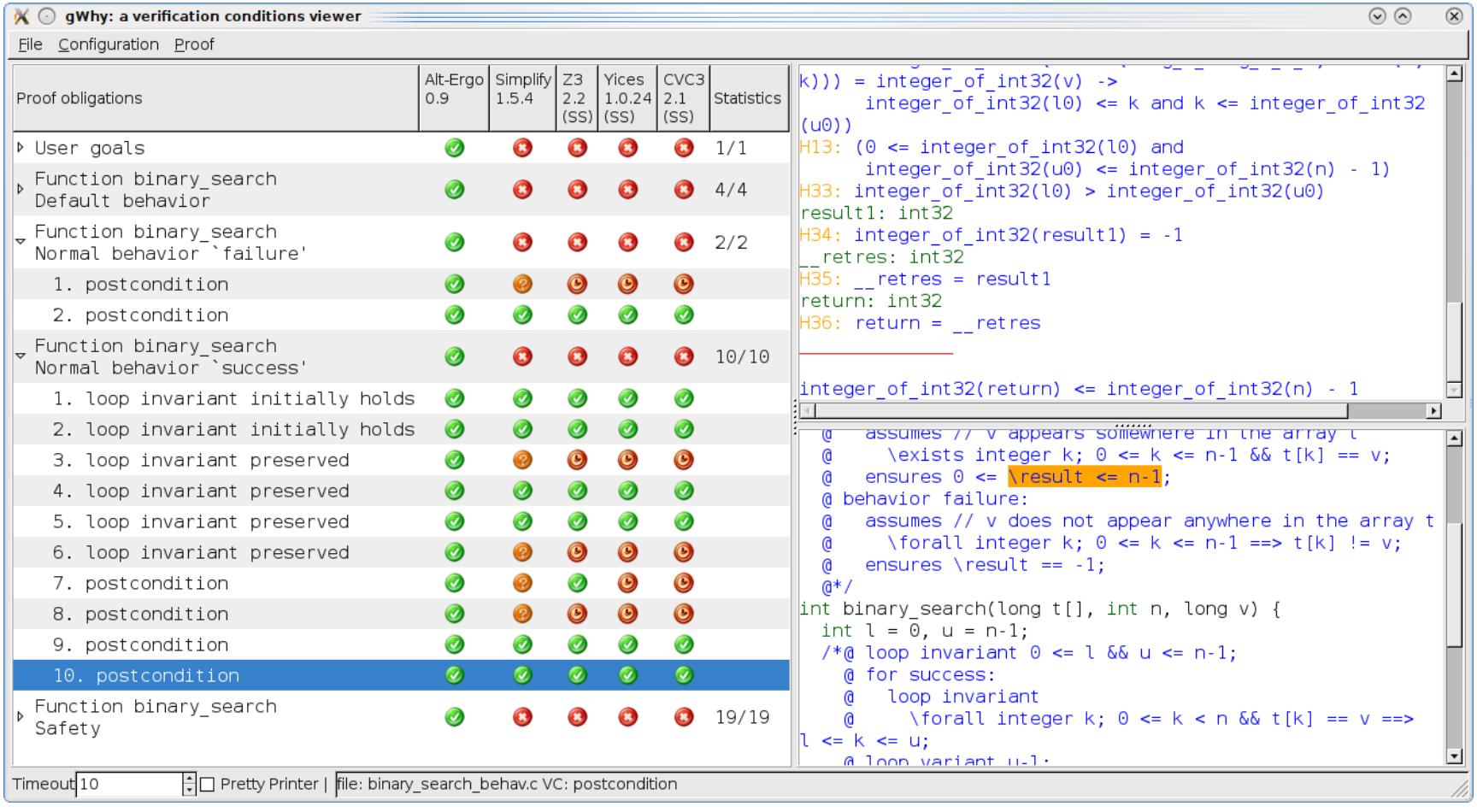
- static analysis,
- model checking,
- program proof,
- and combinations thereof.

Another chunk re-discovers LCF-style interactive proof assistants:

- mechanized metatheory of typed languages
  (the POPLmark challenge)
- verification of fundamental algorithms
  (Java bytecode verification, H-M type inference, SSA conversion)

A modern verification tool

(Frama-C Jessie)



. . . with a rich specification language

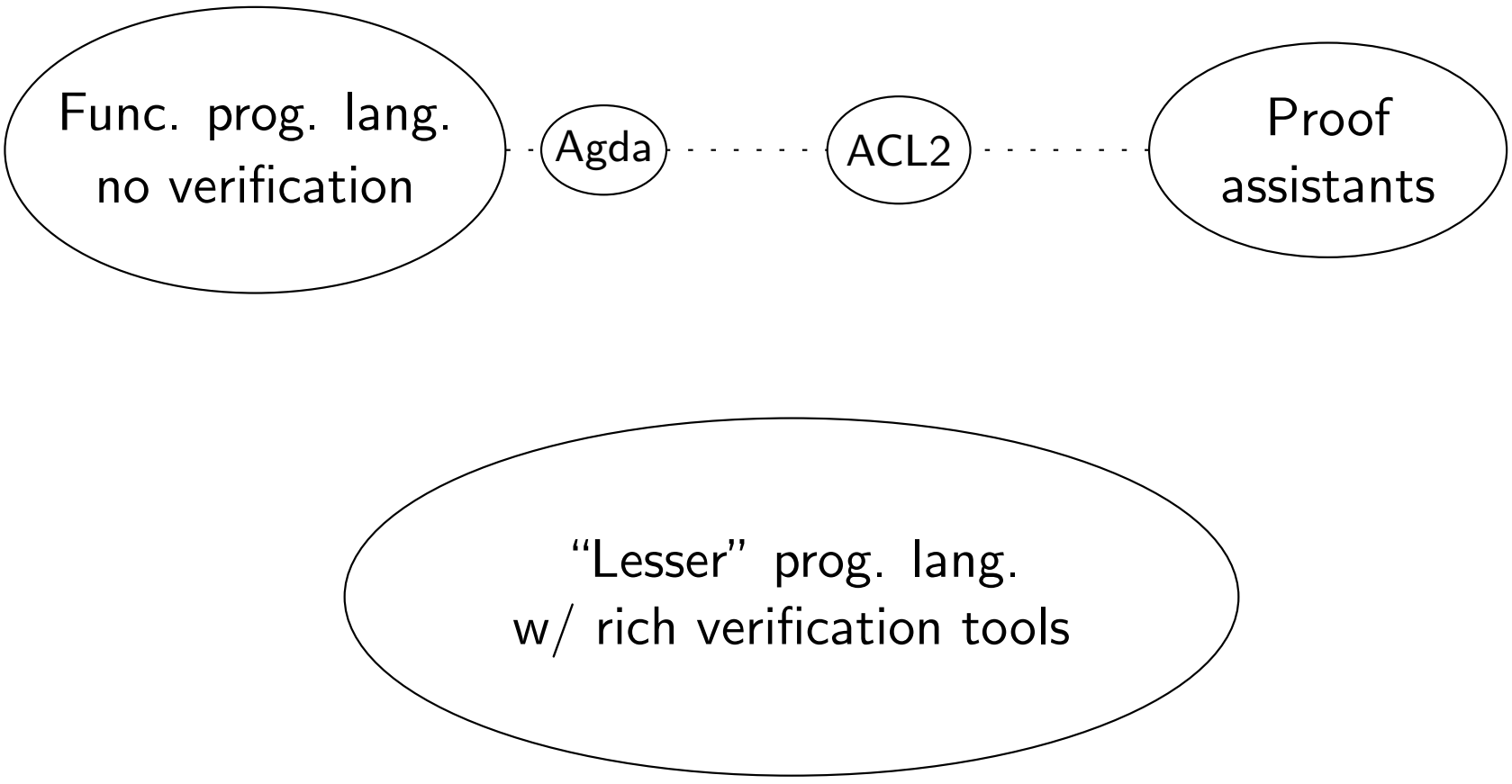
```
/*@ requires n >= 0 && \valid_range(t,0,n-1);
  @ behavior success:
  @   assumes // array t is sorted in increasing order
  @     \forall integer k1, k2;
  @             0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
  @   assumes // v appears somewhere in the array t
  @     \exists integer k; 0 <= k <= n-1 && t[k] == v;
  @   ensures 0 <= \result <= n-1 && t[\result] == v;
  @ behavior failure:
  @   assumes // v does not appear anywhere in the array t
  @     \forall integer k; 0 <= k <= n-1 ==> t[k] != v;
  @   ensures \result == -1;
  @*/

int binary_search(long t[], int n, long v) {
```

A flurry of powerful verification tools

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*

A fragmented landscape



A dilemma

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

1. Use a "lesser" programming language that has good verification tools?
2. Develop verification tools for a proper F.P. language?
3. Use an interactive proof assistant and program within it?

Part III

Programming and proving a compiler in Coq

Early compiler verification in Stanford LCF

(Milner and Weyrauch, *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.

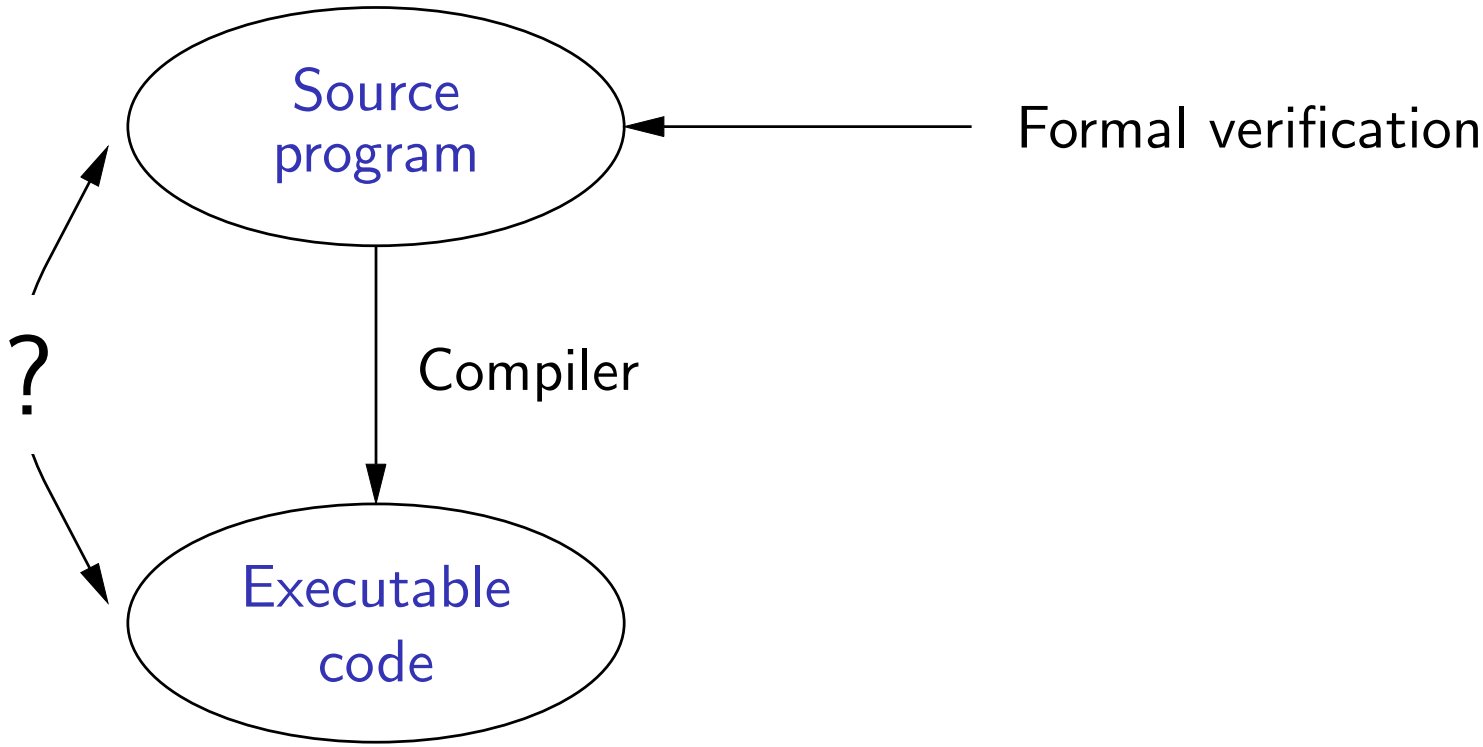
Early compiler verification in Stanford LCF

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp.|swfse e::MT(compe e,sp)≡svof(sp)|((MSE(e,svof sp))&pdof(sp)),
     ∀e.|swfse e::|swft(compe e)≡TT,
     ∀e.|swfse e::(count(compe e)=0)≡TT;

TRY 1 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;
```

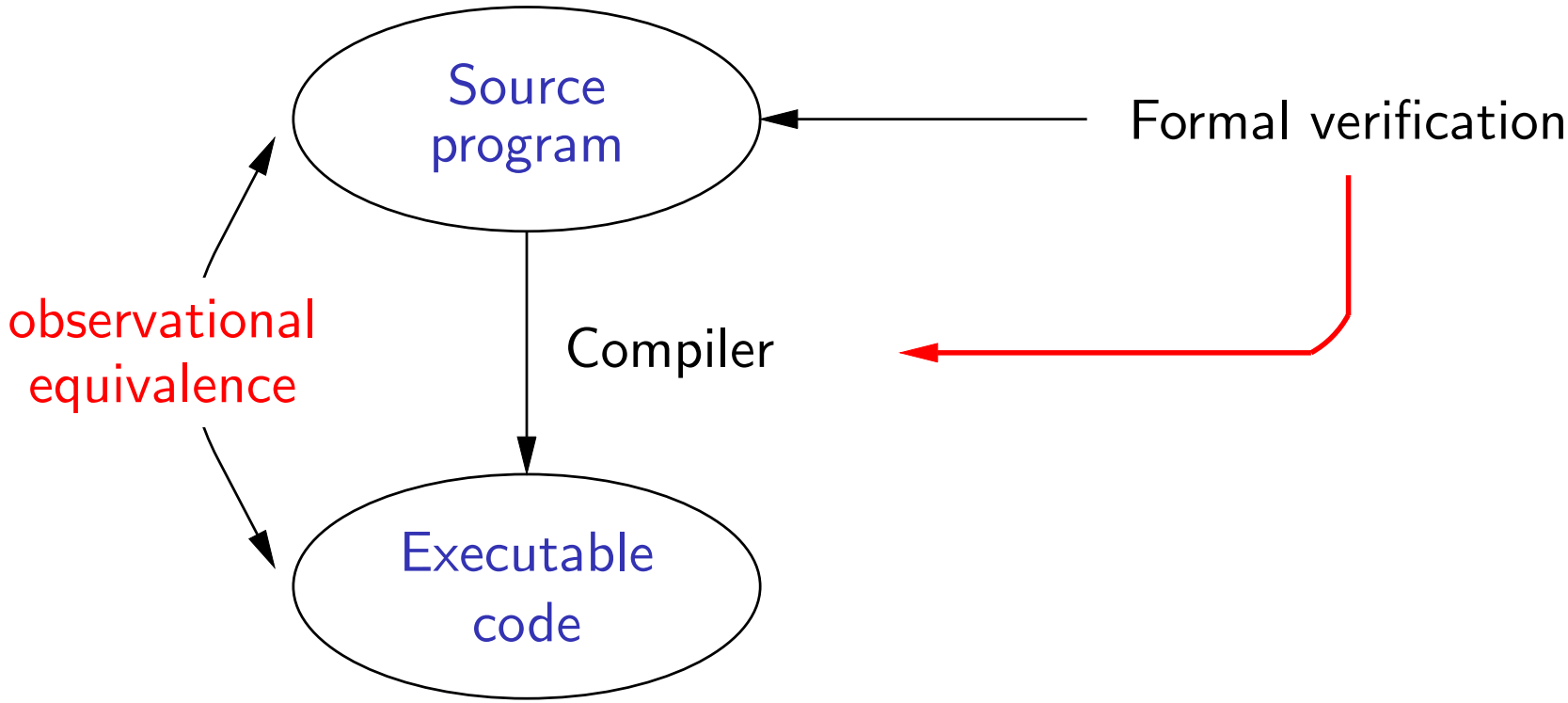
Trust in compilers



Critical software with source-level verification:

Bugs in compilers (miscompilation) can invalidate the guarantees obtained by verification.

Trust in compilers



Formally-verified compiler:

Guarantees that the generated executable code behaves as prescribed by the semantics of the source program.

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.

The CompCert project

(X.Leroy, S.Blazy, et al)

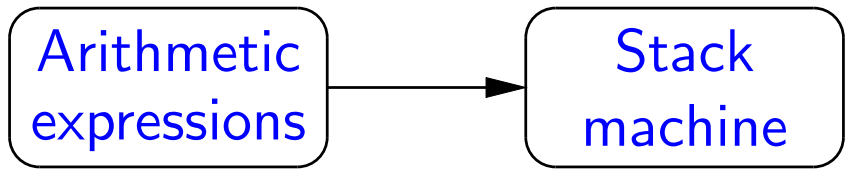
Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a very large subset of C.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  $\Rightarrow$ careful code generation; some optimizations.

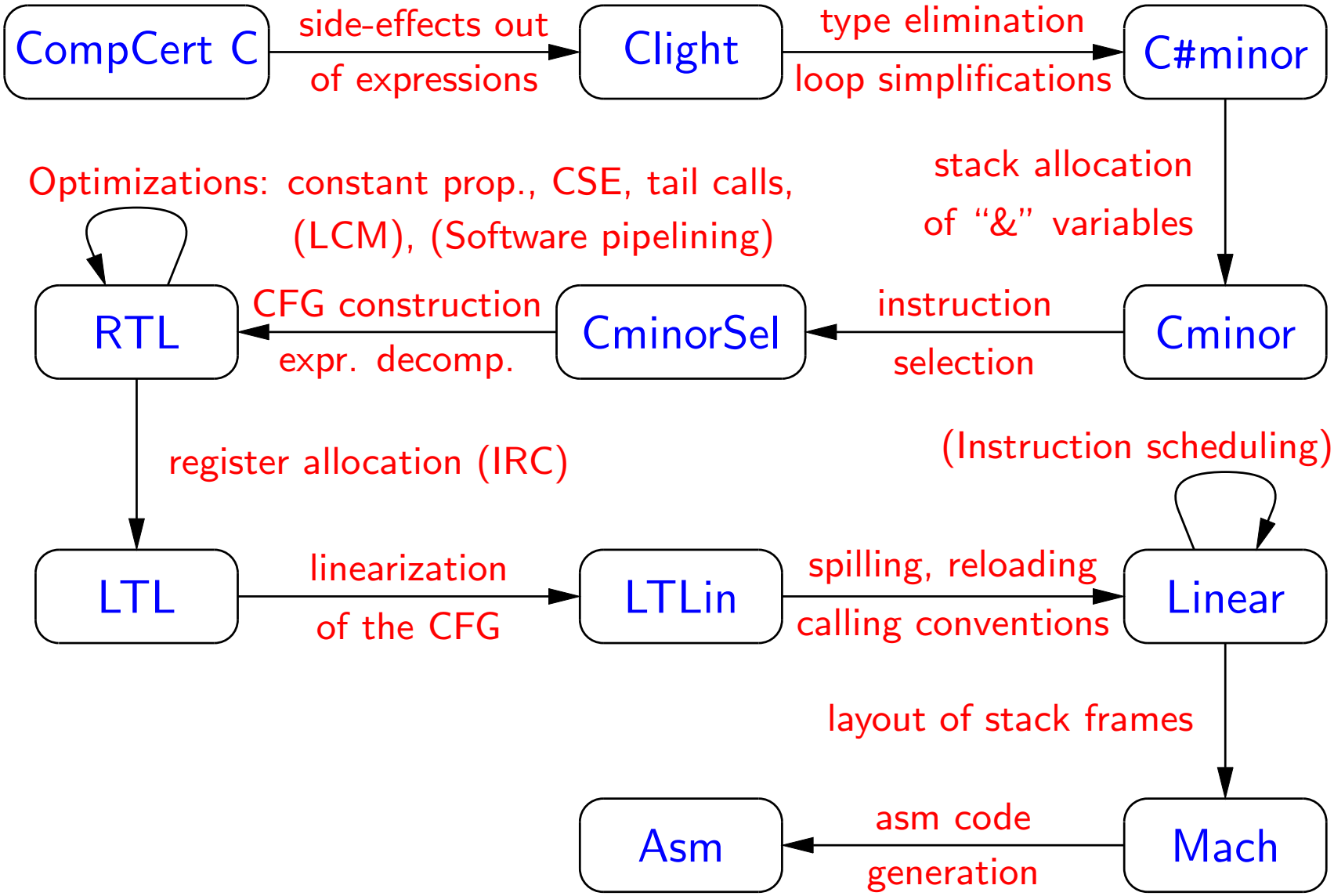
Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler.

Note 2: wish to prove correct the actual implementation, not just the algorithms.

From Milner & Weyrauch...



. . . to the CompCert C verified compiler



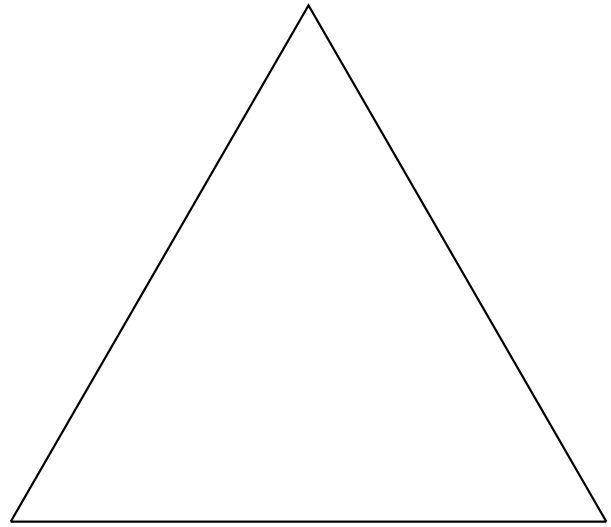
Using the Coq proof assistant

1. To write the specification and conduct the proof.
   (50 000 lines; 5 person.years)

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

Coq in a nutshell

The Calculus of Constructions
(HO constructive logic based on dependent type theory)



Structurally-recursive
functions & pattern-matching

Inductive and coinductive
datatypes and predicates

Programming a compiler pass in Coq

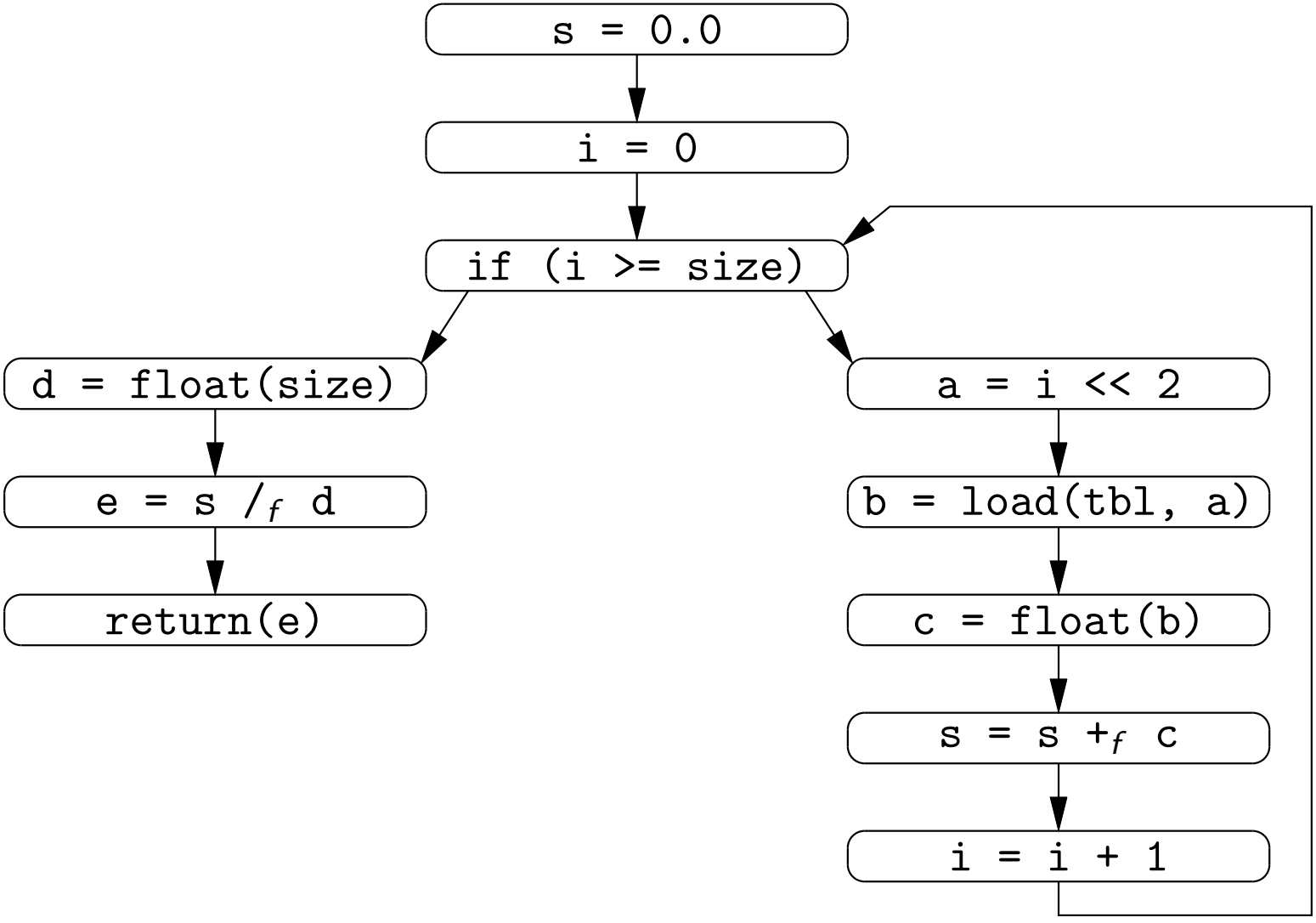
Example: translation from structured control (if/then/else, loops) to a control-flow graph:

- Nodes = instructions.
- Edge from I to J = J is a successor of I (J can execute just after I).

Example: some structured code

```
double average(int * tbl, int size)
{
    double s = 0.0;
    int i;
    for (i = 0; i < size; i++)
        s = s + tbl[i];
    return s / size;
}
```

Example: the corresponding CFG



Programming the translation

The graph is built incrementally by recursive functions such as

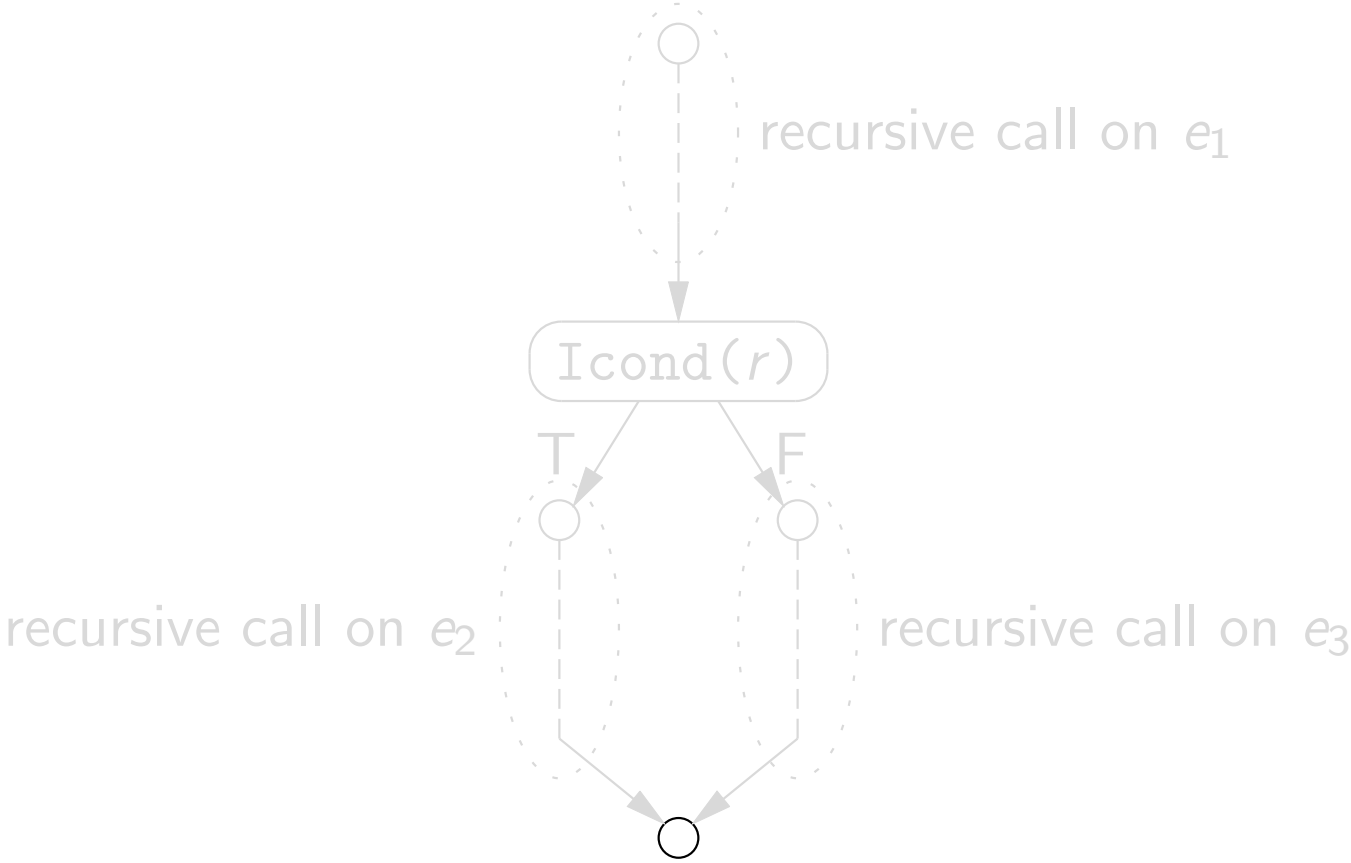
$$\texttt{transl\_expr}\ \Gamma\ e\ r\ n'$$

Effect: add to the graph the instructions that evaluate expression e, deposit its result in temporary r, and branch to node n'. Return first node n of this sequence of instructions.

If we were to write this function in ML, we would use mutable state to represent the current state of the CFG, and the generators for fresh CFG nodes and fresh temporaries.

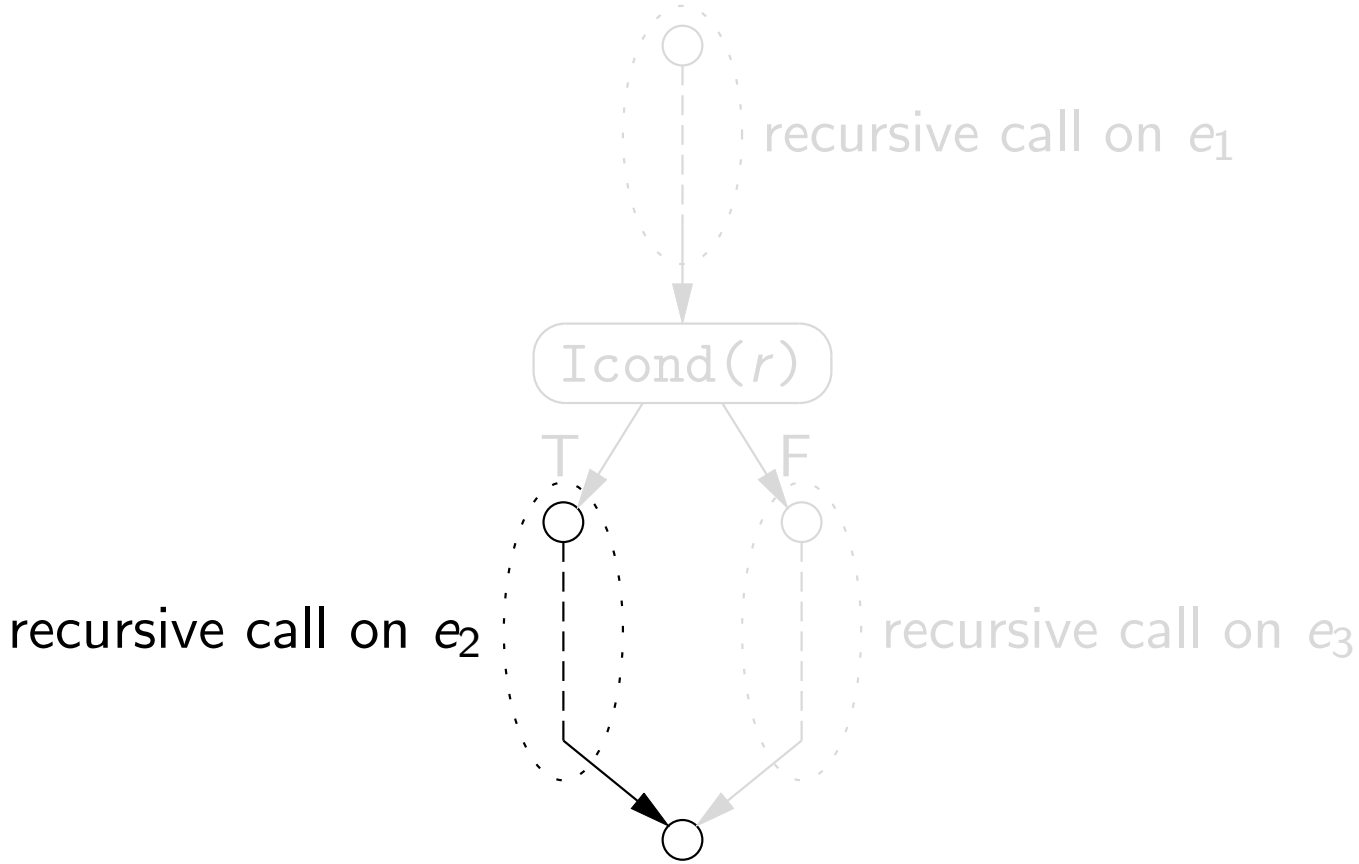
Example of translation

For a conditional expression e_1 ? e_2 : e_3



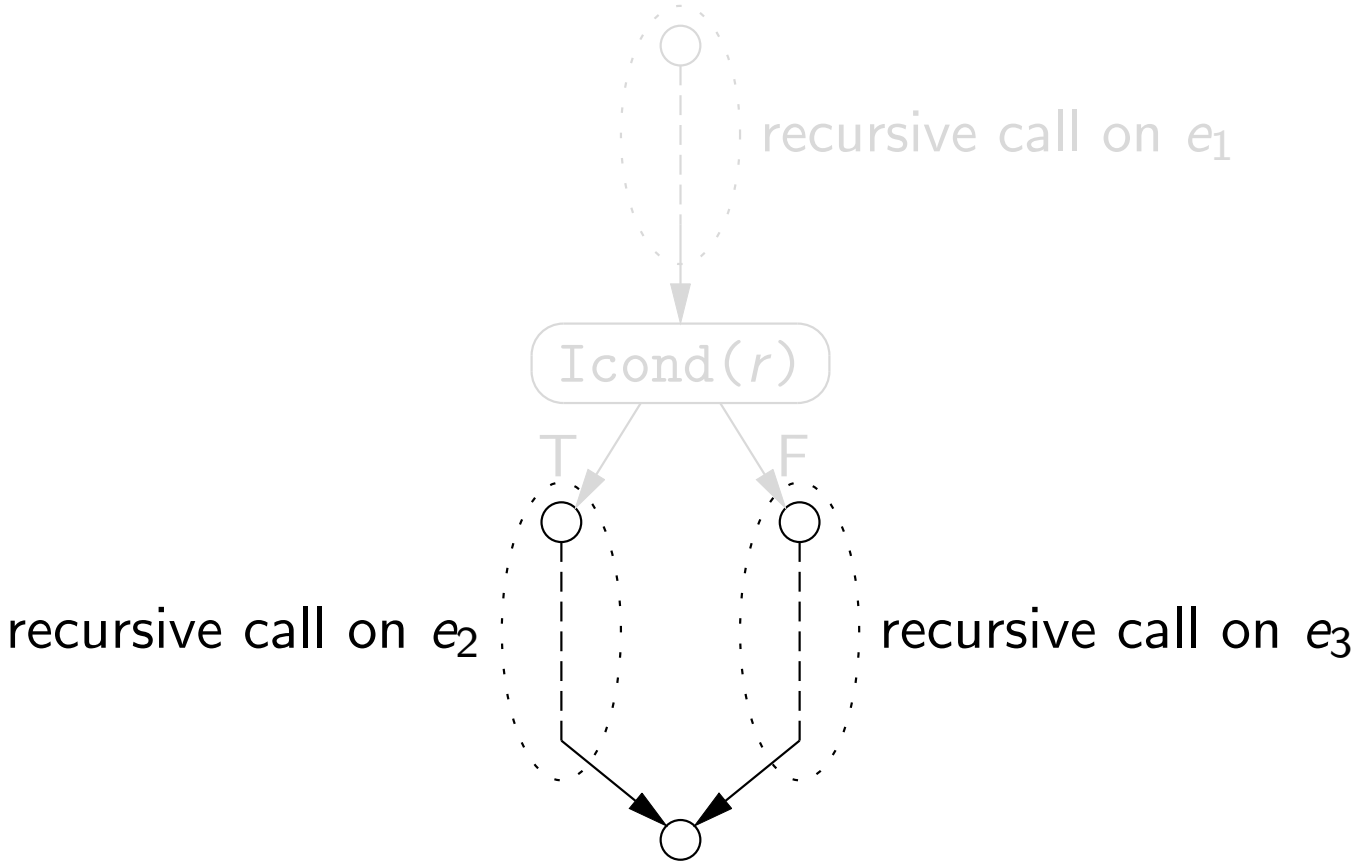
Example of translation

For a conditional expression e_1 ? e_2 : e_3



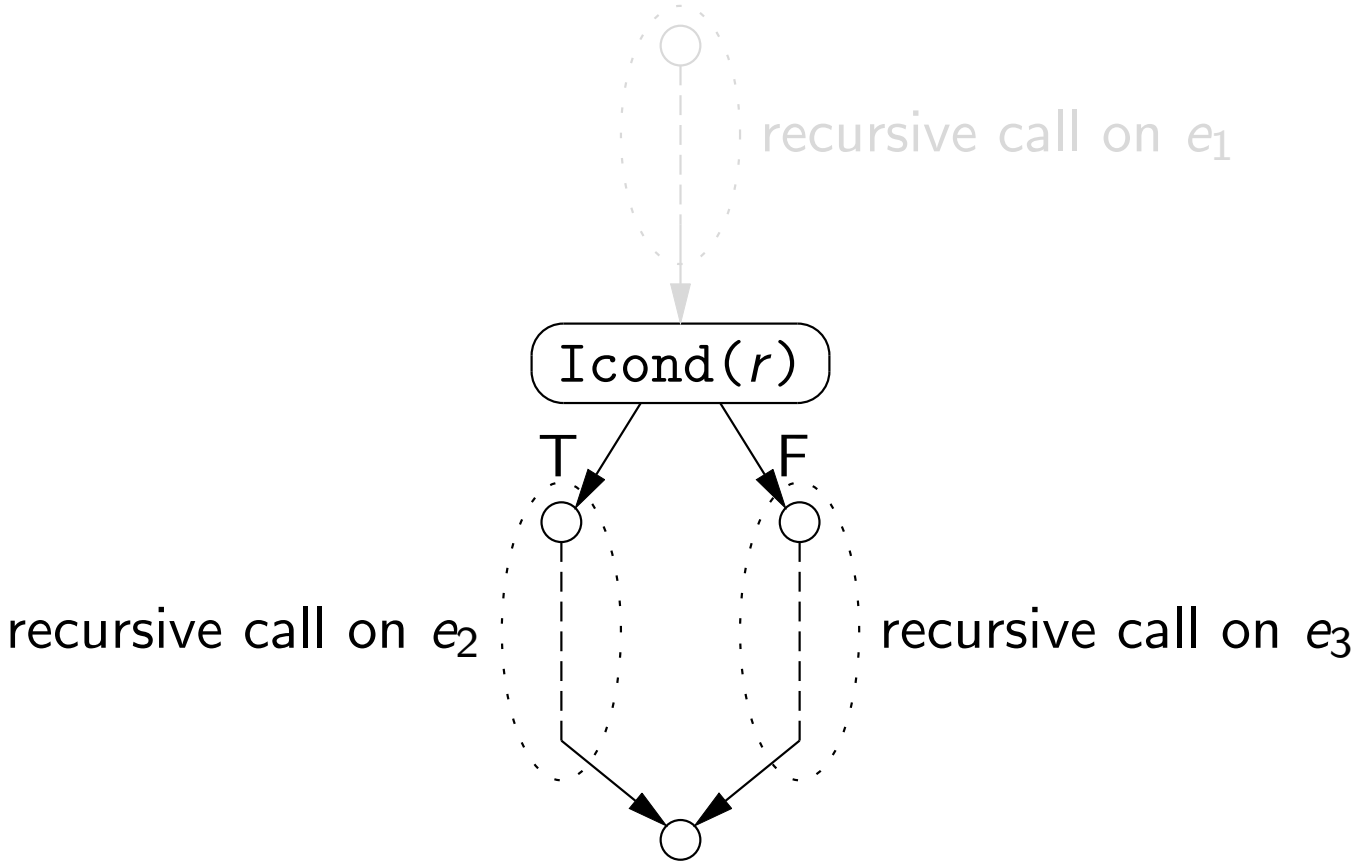
Example of translation

For a conditional expression e_1 ? e_2 : e_3



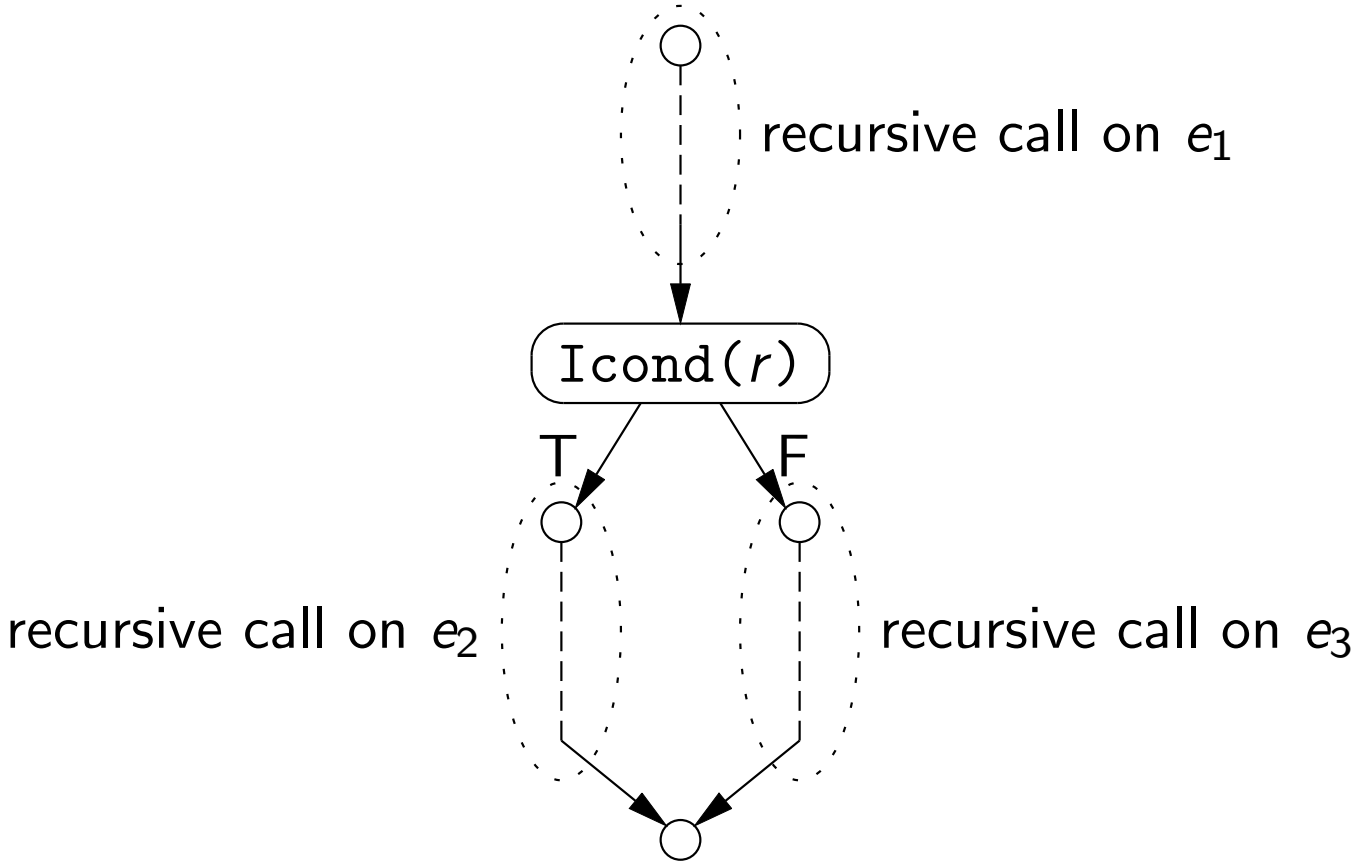
Example of translation

For a conditional expression e_1 ? e_2 : e_3



Example of translation

For a conditional expression e_1 ? e_2 : e_3



Making the state explicit

CFG construction functions as state transformers

$$state \rightarrow result \times state$$

The state is:

```
Record state: Type := {
  st_nextreg: positive;
  st_nextnode: positive;
  st_code: PTree.t instruction
                (* finite map positive -> instr *)
}.
```

The state monad to the rescue

Encapsulate threading of the state in a monad. An imperative computation returning type A is a Coq term of type `mon` A.

```
Definition mon (A: Type) : Type := state -> A * state.

Definition ret (A: Type) (x: A) : mon A :=
  fun (s: state) => (x, s).

Definition bind (A B: Type) (f: mon A) (g: A -> mon B) : mon B :
  fun (s: state) => match f s with (a, s') => g a s' end.

Notation "'do' X <- A ; B" := (bind A (fun X => B))
   (at level 200, X ident, A at level 100, B at level 200).
```

Excerpt from the translation

```
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
  | ...
```

Monotone evolution of the state

A crucial property of the CFG construction functions is that the state evolves in a monotone way:

- New nodes are added to the CFG, but preexisting nodes are never changed.
- Temporaries are not reused.

We must maintain an invariant over states:

```
state_wf s :=
  forall pc, pc >= s.(st_nextnode) -> s.(st_code) pc = None
```

and guarantee that if the state evolves from `s1` to `s2`, we have

```
state_incr s1 s2: =
       s1.(st_nextnode) <= s2.(st_nextnode)
  /\   s1.(st_nextreg)  <= s2.(st_nextreg)
  /\   forall pc,
       pc < s1.(st_nextnode) -> s2.(st_code) pc = s1.(st_code) pc
```

Dependent types to the rescue

The properties on the state are guaranteed by the basic state operations (add a new node, etc), but we still have to show that all CFG construction operations, built on these basic operations, also guarantee these properties.

Dependent types can help!

The `state_wf` property of a single state can easily be attached to all states we manipulate:

```
Record state: Type := {
  st_nextreg: positive;
  st_nextnode: positive;
  st_code: positive -> option instr;
  st_wf: forall pc,
         pc >= st_nextnode -> st_code pc = None
}.
```

Dependent types to the rescue

Less trivially, the `state_incr` property of two states can be neatly hidden in the monad.

```
Definition mon (A: Type) : Type :=
  forall (s: state), A * { s': state | state_incr s s' }

Definition ret (A: Type) (x: A) : mon A :=
  fun (s: state) => (x, exist s (state_incr_refl s)).

Definition bind (A B: Type) (f: mon A) (g: A -> mon B) : mon B :=
  fun (s: state) =>
    match f s with (a, exist s' i) =>
      match g a s' with (b, exist s'' i') =>
        (b, exist s'' (state_incr_trans i i'))
      end
    end.
```

Dependently typed monads: a winning combination?

While dependent function types $\forall x : A,\ \{y : B \mid P\ x\ y\}$ are hard to use in general, a monadic encapsulation hides them entirely from the user of the monad.

The client code using the monad is unchanged:

```
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;
      ...
```

Programming limitations

Is it effective to program directly in Coq?

In exchange for powerful proof support, we must deal with

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

Is there a better way?

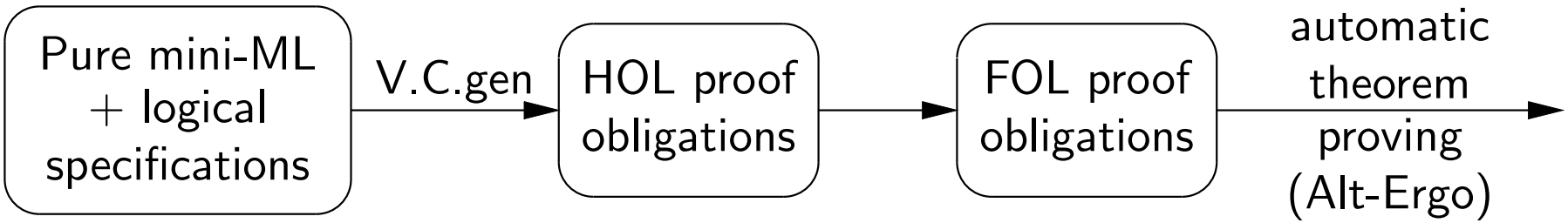
Part IV

Alternative approaches

The Pangolin system

(Y. Régis-Gianas, F. Pottier)

Hoare logic for a pure functional language.



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))
=
   ...
```

HALO

(D. Vytiniotis, S. Peyton Jones, D. Rosen, K. Claessen)

Static verification of contracts for Haskell.

$$C ::= \{x \mid e\} \mid (x : C_1) \to C_2 \mid C_1 \& C_2$$

(Contracts were initially introduced as dynamically-checked assertions.)

HALO generates and solves first-order proof obligations much like Pangolin does.

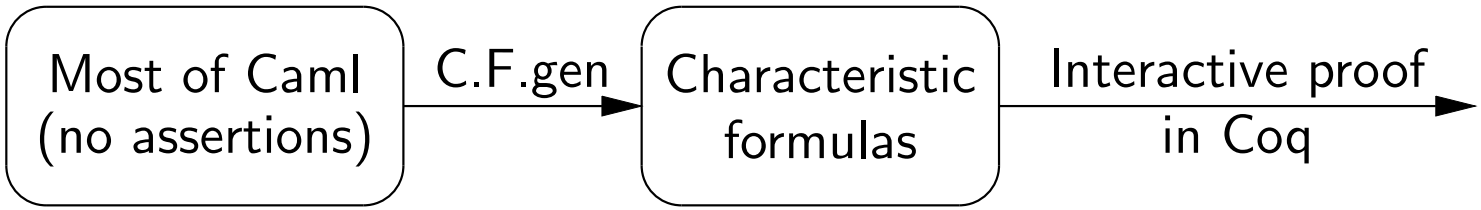
However, specifications are given as boolean-valued Haskell expressions (e above), not HO logic formulas, hence

- No infinite quantification.
- Delicate semantic issues when e does not terminate.

The CFML system

(A. Charguéraud)

An alternative approach based on characteristic formulas:



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

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

It can be viewed simultaneously as

- a weakest precondition for t
- a denotational semantics for t
- a deep embedding of t in HO logic.

The CFML system

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

Pre-/post-conditions and invariants are not written in the source code, but provided as needed during the proof.

Can also be extended with exceptions and mutable state.

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 future F.P. languages with verifiability in mind?