Verified compilation
An introduction to CompCert

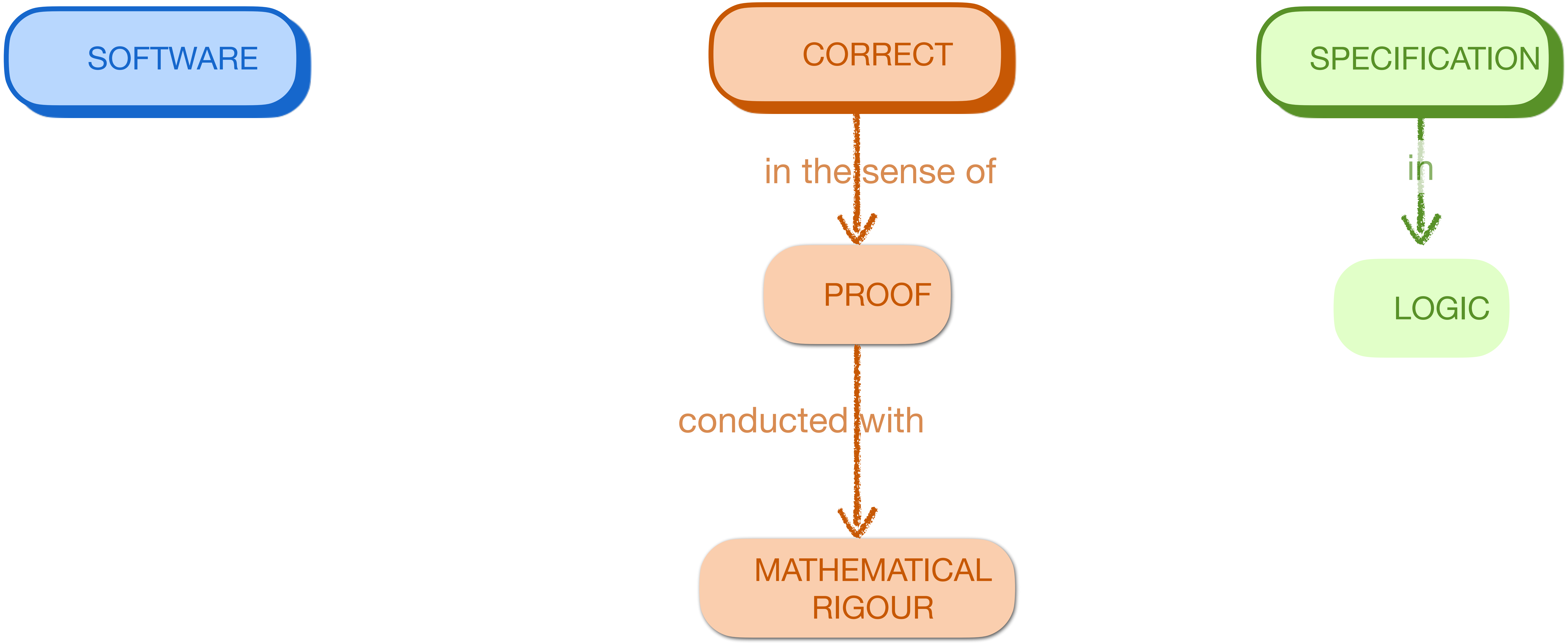
Sandrine Blazy

Université de Rennes

UMR IRISA

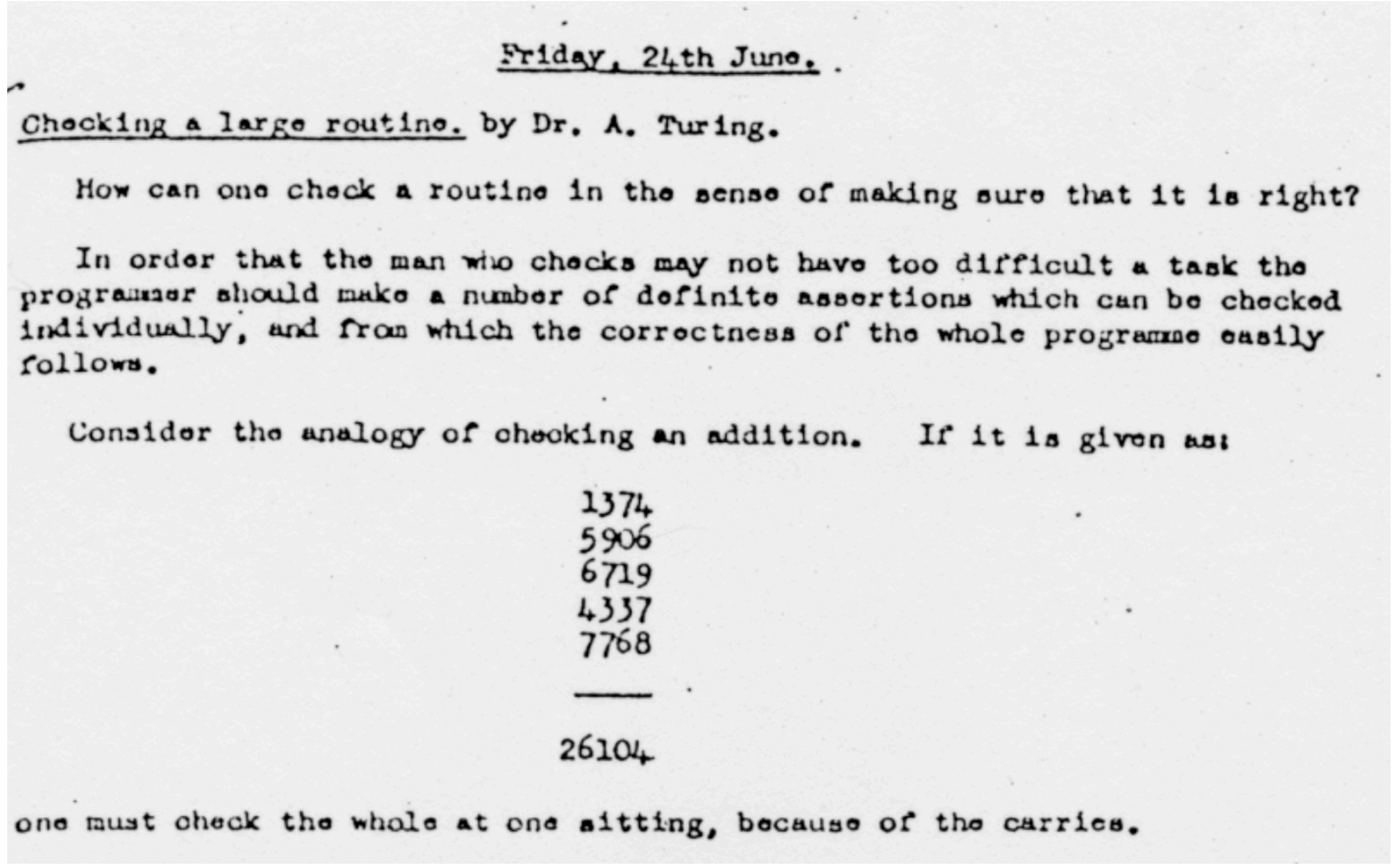
OPLSS, Eugene, 2023-07-05

Deductive verification



From early intuitions …

A. M. Turing.
Checking a large routine. 1949.



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

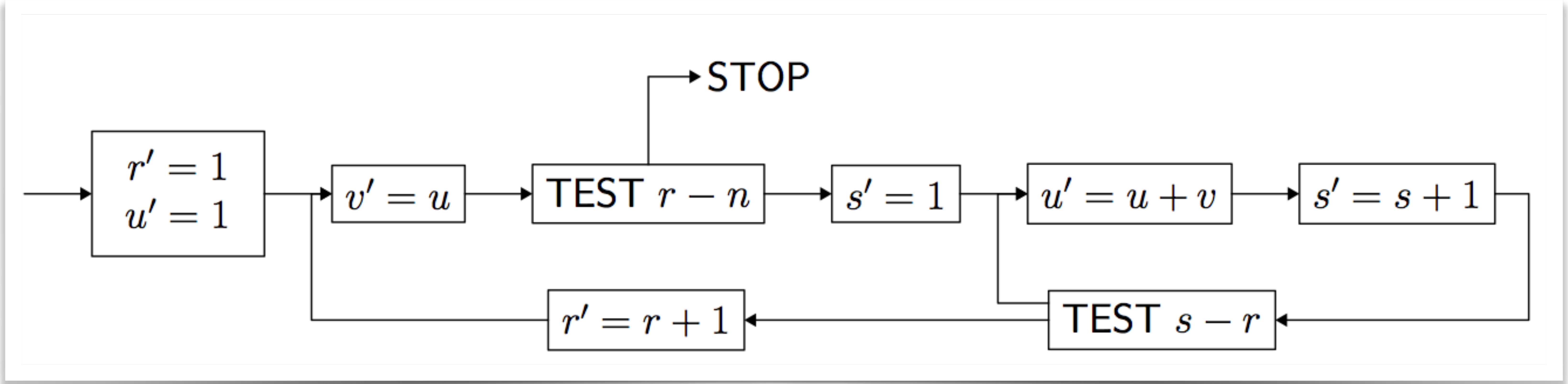
How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

1374
5906
6719
4337
7768
―――
26104

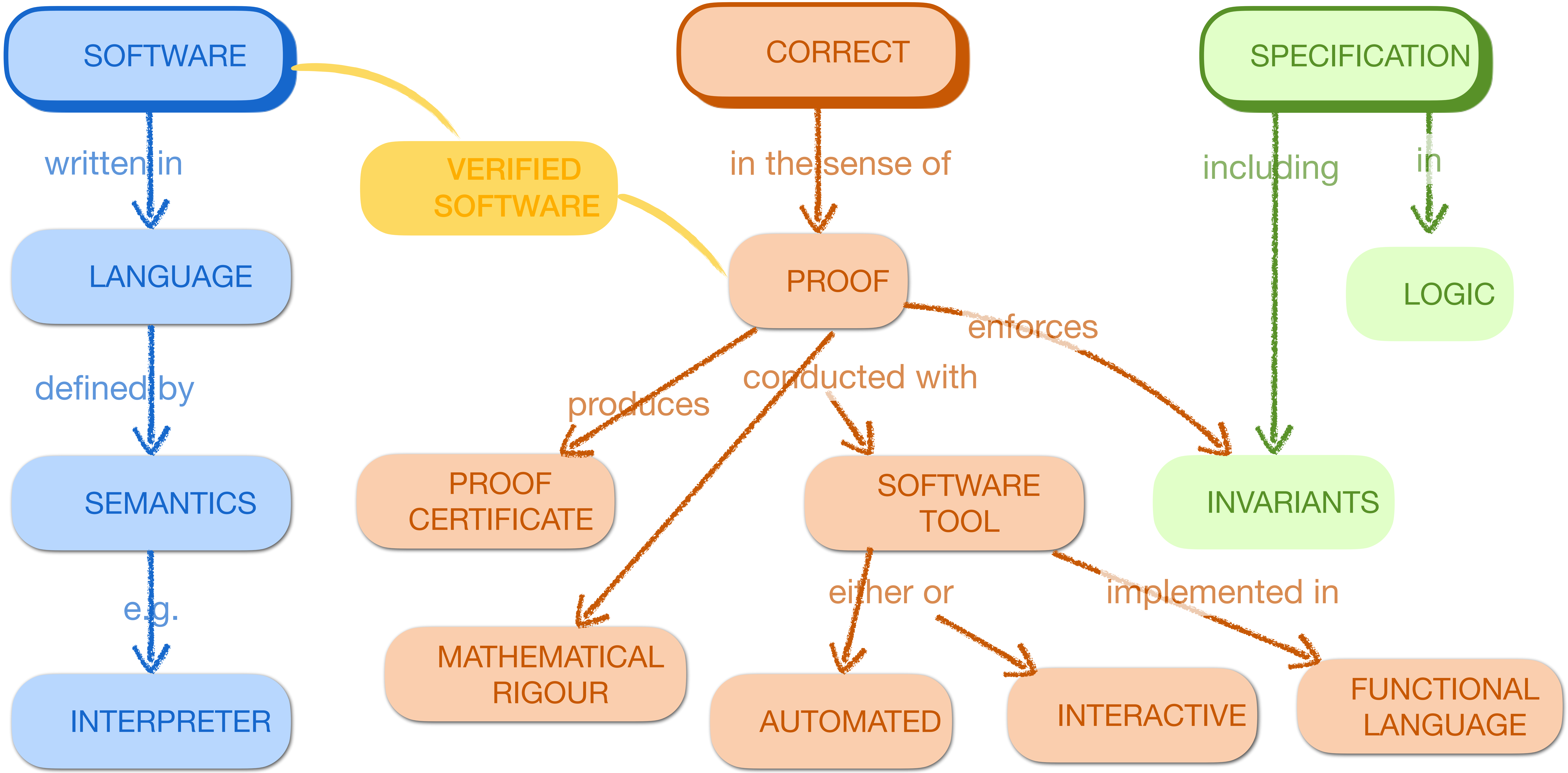
one must check the whole at one sitting, because of the carries.




```
u ← 1
for r = 0 to n − 1 do
  v ← u
  for s = 1 to r do
    u ← u + v
```

… to deductive-verification and automated tools

Floyd 1967, Hoare 1969



Another historical example

Boyer-Moore’s majority. 1980

Given N votes, determine the majority if any

majority = A

cpt_delta = 3

MJRTY—A Fast Majority Vote Algorithm[1]

Robert S. Boyer and J Strother Moore

Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

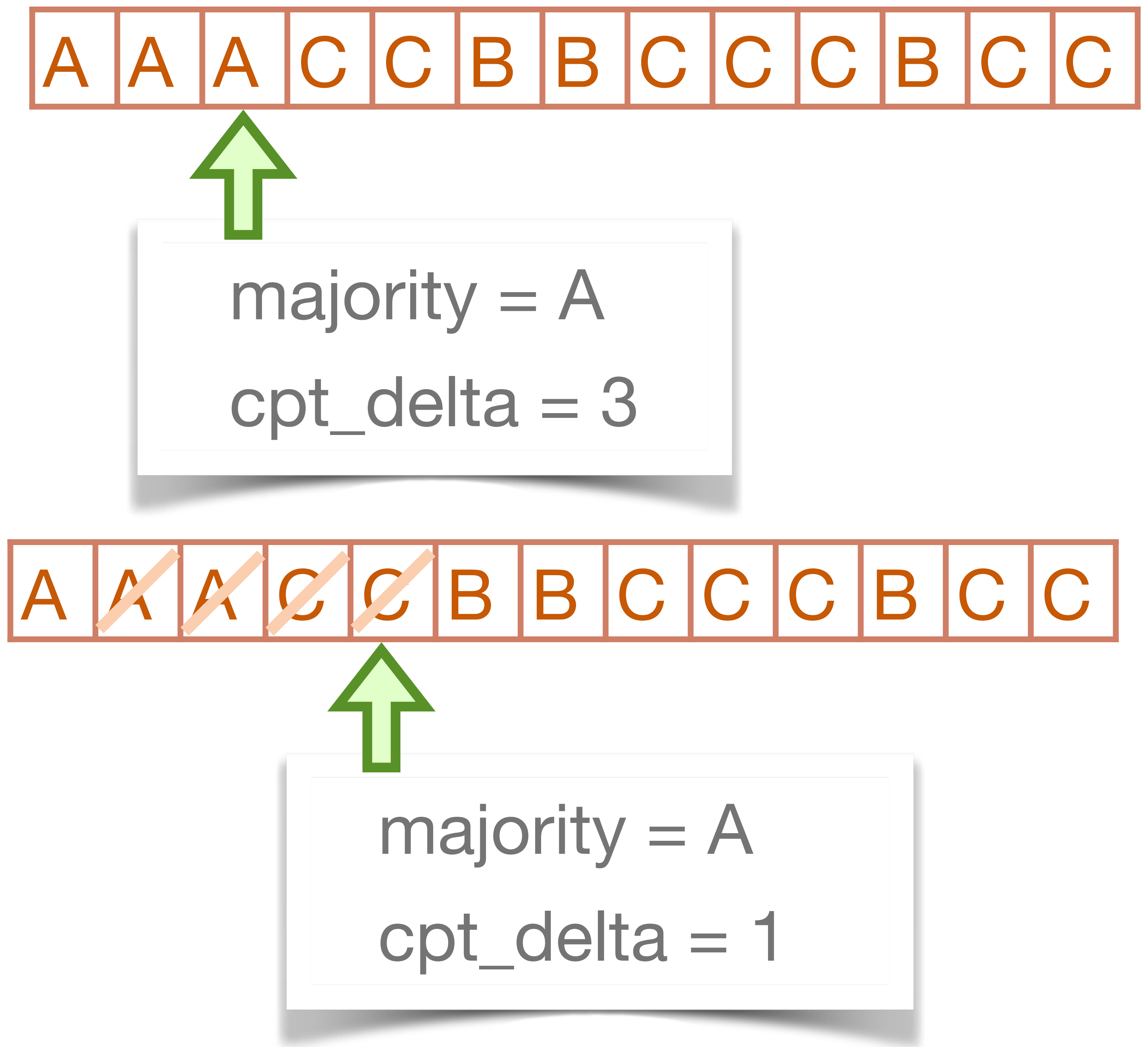
Abstract

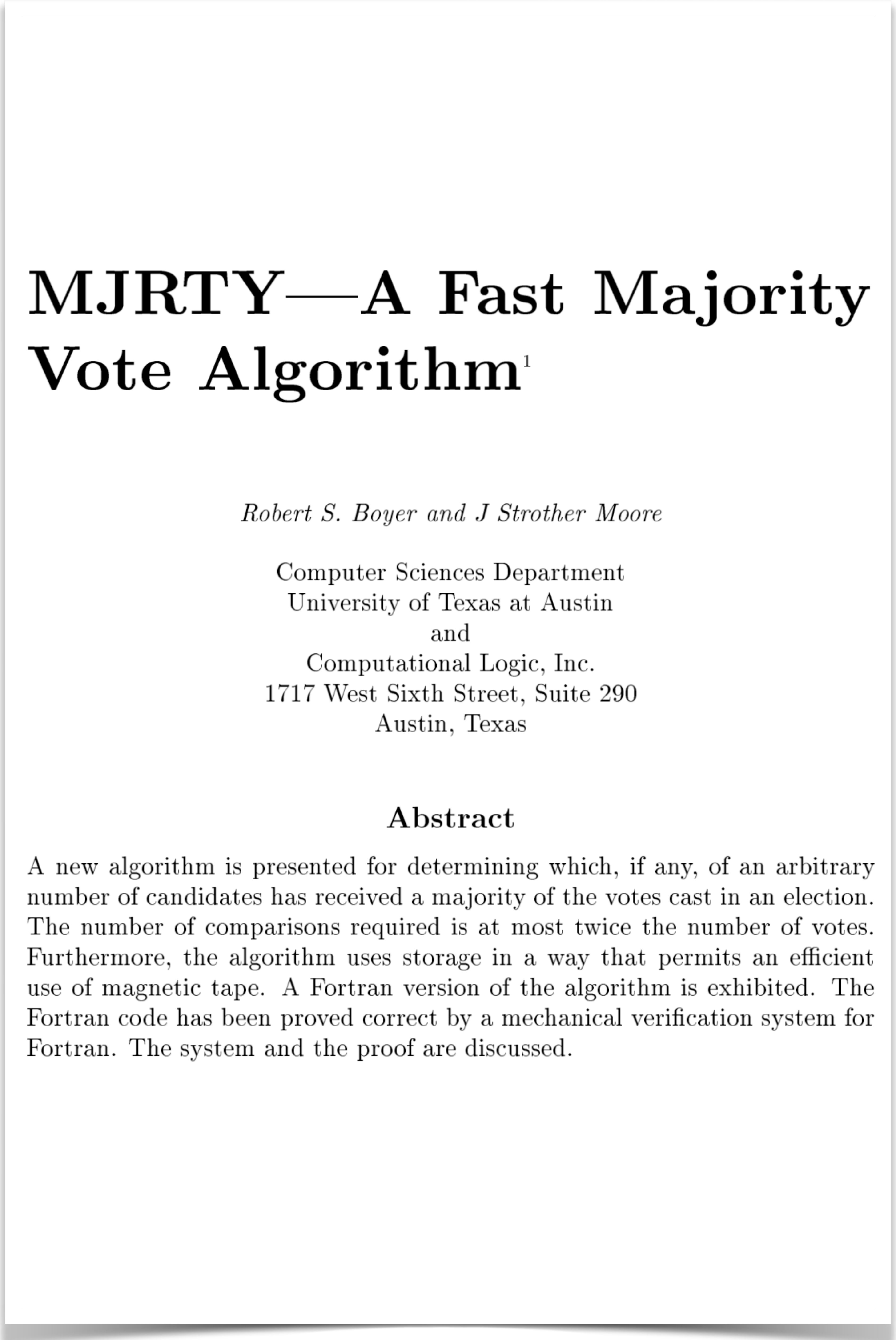
A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient use of magnetic tape. A Fortran version of the algorithm is exhibited. The Fortran code has been proved correct by a mechanical verification system for Fortran. The system and the proof are discussed.

Another historical example

Boyer-Moore's majority. 1980

Given N votes, determine the majority if any





MJRTY—A Fast Majority Vote Algorithm[1]

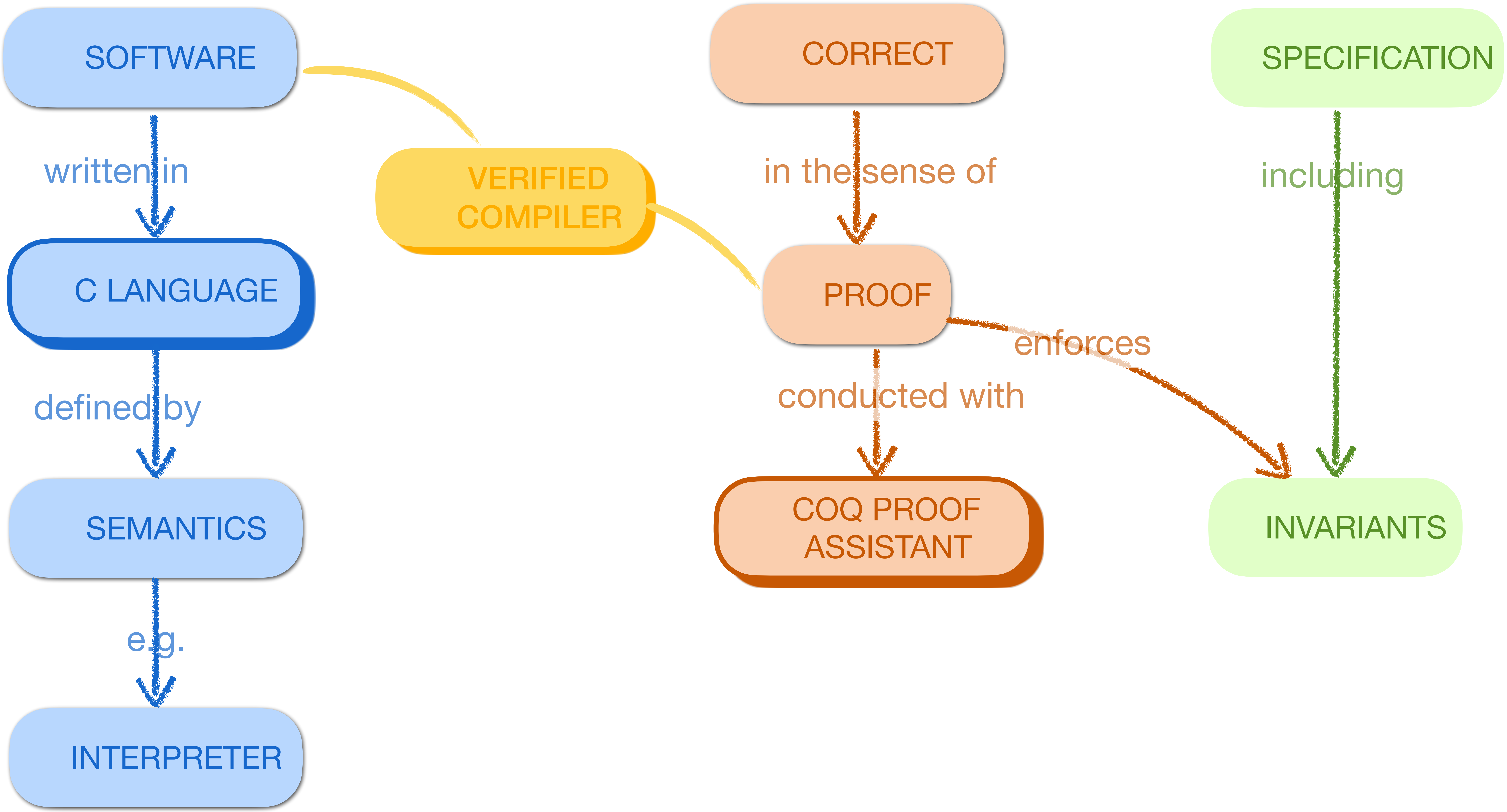
Robert S. Boyer and J Strother Moore

Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient use of magnetic tape. A Fortran version of the algorithm is exhibited. The Fortran code has been proved correct by a mechanical verification system for Fortran. The system and the proof are discussed.

Part 1: summary



Lecture material

https://people.irisa.fr/Sandrine.Blazy/2023-OPLSS

These slides
(including some slides borrowed from by Xavier Leroy)

COLLÈGE DE FRANCE 1530

Mechanized semantics, second lecture

***Traduttore, traditore*:**
formal verification of a compiler

Xavier Leroy
2019-12-12
Collège de France, chair of software sciences

Reused Coq developments

SOFTWARE FOUNDATIONS
VOLUME 1: LOGICAL FOUNDATIONS

TABLE OF CONTENTS | INDEX | ROADMAP

IMP
SIMPLE IMPERATIVE PROGRAMS

In this chapter, we take a more serious look at how to use Coq as a tool to study other things. Our case study is a *simple imperative programming language* called Imp, embodying a tiny core fragment of conventional mainstream languages such as C and Java.

Mechanized semantics: the Coq development

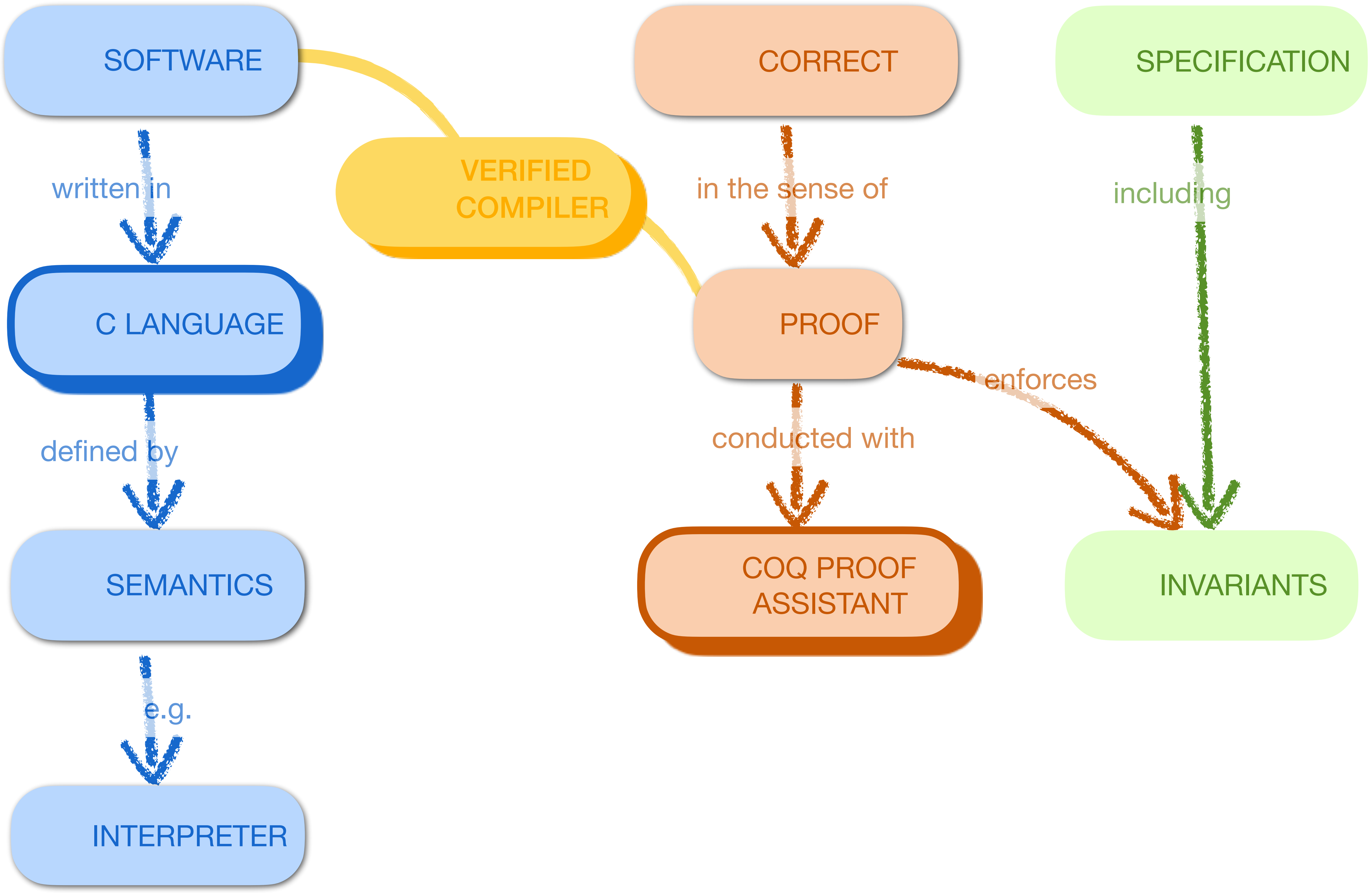
This repository contains the Coq sources for the course "Mechanized semantics" given by Xavier Leroy at Collège de France in 2019-2020.

This is the English version of the Coq sources. La version commentée en français est disponible ici.

An HTML pretty-printing of the commented sources is also available:

1. The semantics of an imperative language
   - Module IMP: the imperative language IMP and its various semantics.
   - Library Sequences: definitions and properties of reduction sequences.
2. Formal verification of a compiler
   - Module Compil: compiling IMP to a virtual machine.
   - Library Simulation: simulation diagrams between two transition systems.

Part 2:
early intuitions



The miscompilation risk

Compilers still contain bugs!

> We found and reported **hundreds** of previously **unknown** bugs [...]. Many of the bugs we found cause a compiler to emit incorrect code **without any warning**. 25 of the bugs we reported against GCC were classified as **release-blocking**.

[Yang, Chen, Eide, Regehr. Finding and understanding bugs in C compilers. PLDI'11]

Verified compilation

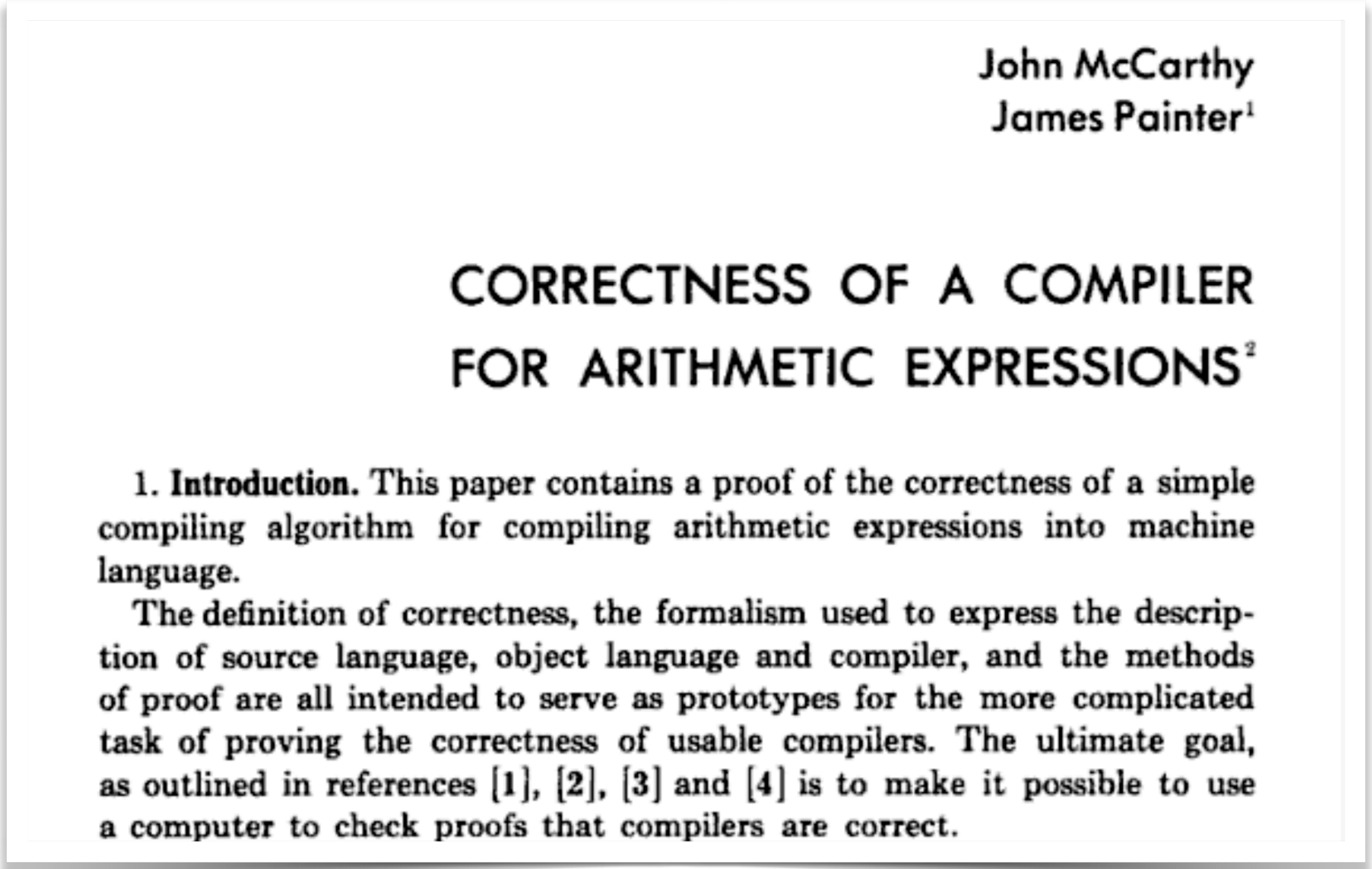
Compilers are complicated programs, but have a rather simple end-to-end specification:

> The generated code must behave as prescribed by the semantics of the source program.

This specification becomes mathematically precise as soon as we have formal semantics for the source language and the machine language.

Then, a formal verification of a compiler can be considered.

An old idea …



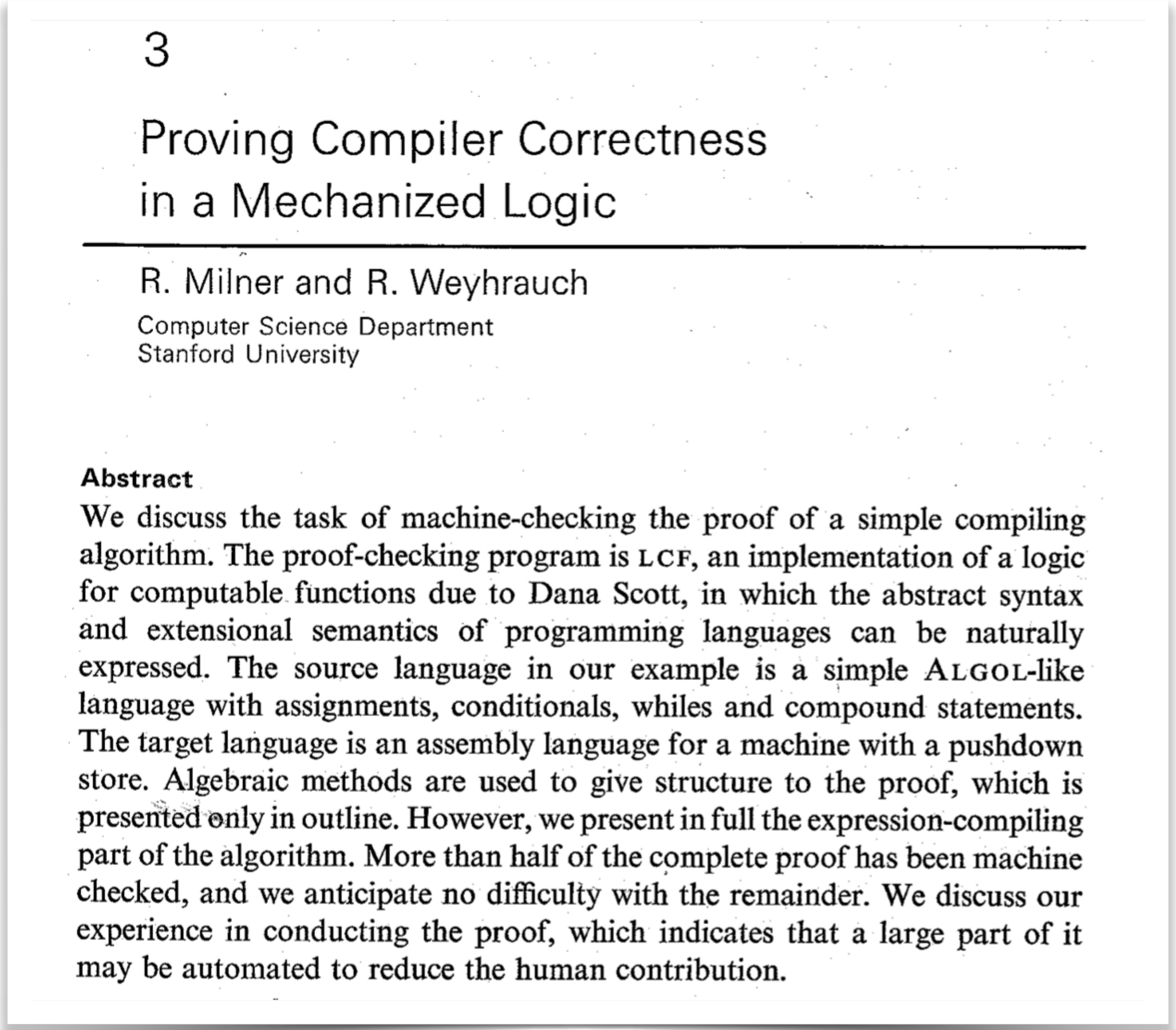
John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

1. **Introduction.** This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

Mathematical Aspects of Computer Science, 1967



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.

Machine Intelligence (7), 1972

Now taught as an exercise

(Mechanized semantics: when machines reason about their languages, X.Leroy)
(Software foundations, B.Pierce et al.: exercise stack_compiler_correct)

```
Inductive aexp := ANum(n:nat)| AId(x:string)| APlus(a1 a2:aexp)| ...
```

```
Definition state := string → nat.
```

```
Fixpoint aeval(s:state)(e:aexp):nat := ...
```

semantics (aeval, s_execute)

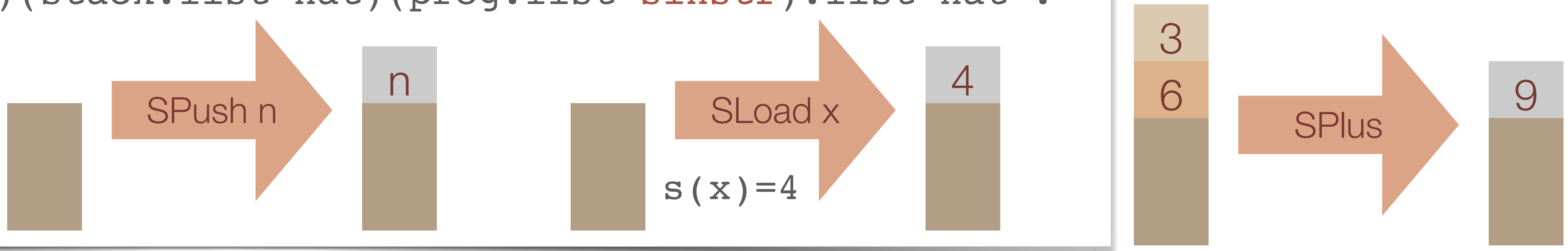
compiler (s_compile)

compilation

```
Fixpoint s_compile(e:aexp):
  list sinstr
  := ...
```

```
Inductive sinstr := SPush(n:nat)| SLoad(x:string)| SPlus| SMinus| SMult.
```

```
Fixpoint s_execute(s:state)(stack:list nat)(prog:list sinstr):list nat :=
 match (prog, stack) with
  | (nil, _ ) => stack
  | ...
 end.
```



Now taught as an exercise

(Mechanized semantics: when machines reason about their languages, X.Leroy)
(Software foundations, B.Pierce et al.: exercise stack_compiler_correct)

semantics (aeval, s_execute)

compiler (s_compile)

interactive proof

```
Fixpoint aeval(s:state)(e:aexp):nat := ...
```

compilation

```
Fixpoint s_compile(e:aexp): list sinstr := ...
```

```
Fixpoint s_execute(s:state)(stack:list nat)(prog:list sinstr):list nat := ...
```

```
Theorem s_compile_correct: ∀ s e,
 s_execute s [] (s_compile e) = [aeval s e].
Proof.
```

Now taught as an exercise

(Mechanized semantics: when machines reason about their languages, X.Leroy)
(Software foundations, B.Pierce et al.: exercise stack_compiler_correct)

```
Fixpoint aeval(s:state)(e:aexp):nat := ...
```

compilation

```
Fixpoint s_compile(e:aexp): list sinstr := ...
```

```
Fixpoint s_execute(s:state)(stack:list nat)(prog:list sinstr):list nat := .
```

```
Theorem execute_app : ∀ st p1 p2 stack,
    s_execute st stack (p1 ++ p2) = s_execute st (s_execute st stack p1) p2.
Proof.
 (* … *)
Qed.
```

```
Theorem s_compile_correct_aux: ∀ s e stack,
 s_execute s stack (s_compile e) = aeval e :: stack.
Proof.
   induction e; (* … *)
Qed.
```

proof by induction on the structure of e

```
Theorem s_compile_correct: ∀ s e,
 s_execute s [] (s_compile e) = [aeval s e].
Proof.
   intros. apply s_compile_correct_aux.
Qed.
```

semantics (aeval, s_execute)

compiler (s_compile)

interactive proof

extraction

toy-compiler.ml

```
Extraction s_compile.
```

Course outline

Formal verification in Coq of a non-optimizing compiler for a simple imperative language (from IMP language to VM language)

Extension of these ideas to CompCert, a realistic C compiler

The CompCert formally verified compiler

(X.Leroy, S.Blazy et al.) https://compcert.org

A moderately optimizing C compiler

Targets several architectures (PowerPC, ARM, RISC-V and x86)

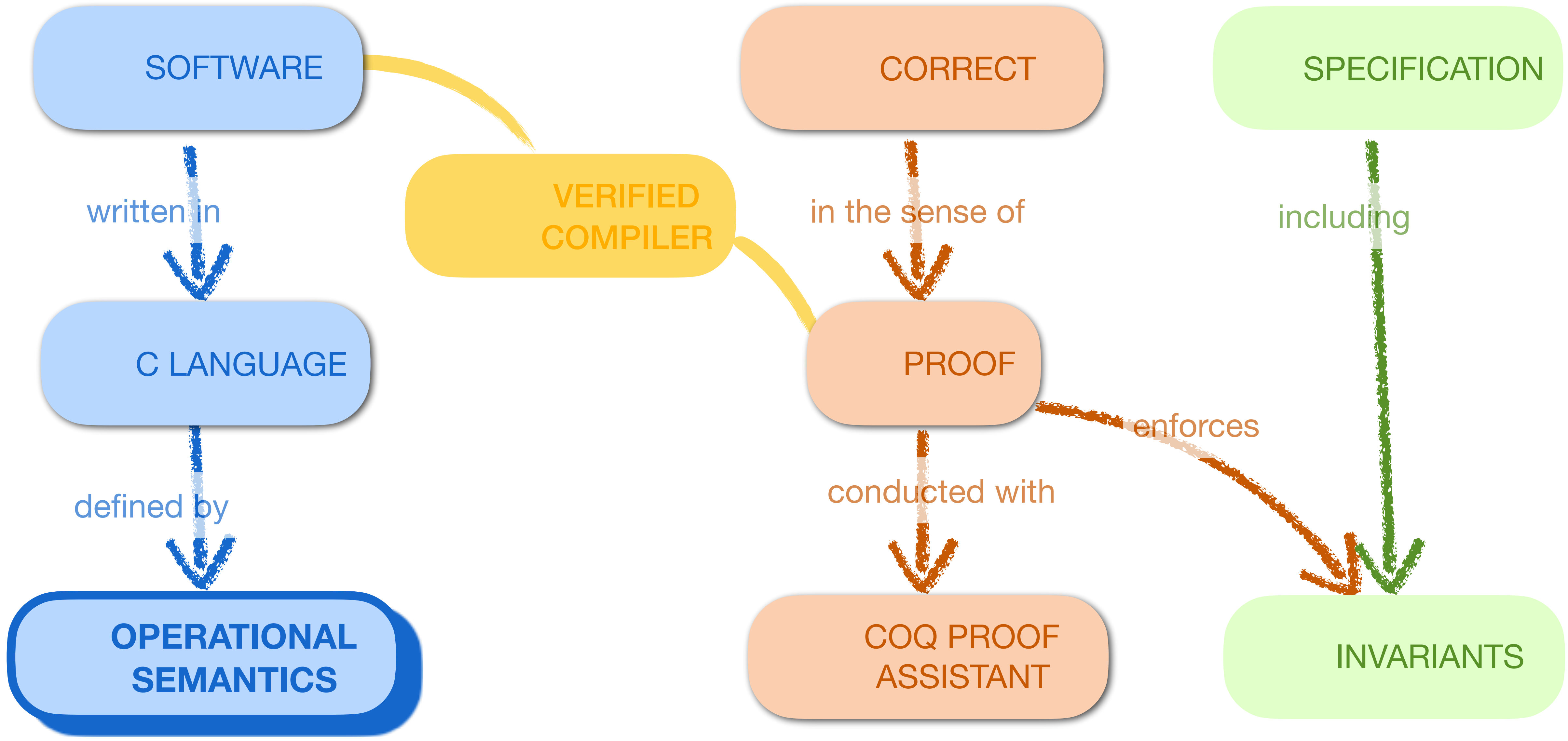
Programmed and verified using the Coq proof assistant

Shared infrastructure for ongoing research

Used in commercial settings (for emergency power generators and flight control navigation algorithms) and for software certification - AbsInt company
Improved performances of the generated code while providing proven traceability information

ACM Software System award 2021
ACM SIGPLAN Programming Languages Software award 2022

Part 3:
basics of
verified compilation



Compiling IMP instructions
Already seen in Imp.v

semantics (aeval, beval, ceval)

Denotational style for the semantics of IMP expressions

```
Fixpoint aeval(s:state)
(e:aexp): nat := ...
```

Big-step (operational) style for commands: relation $c/s \Rightarrow s'$

```
Inductive com :=
   | CSkip
   | CAss (x: string) (a: aexp)
   | CSeq (c1 c2: com)
   | CIf (b: bexp) (c1 c2: com)
   | CWhile (b: bexp) (c: com).
```

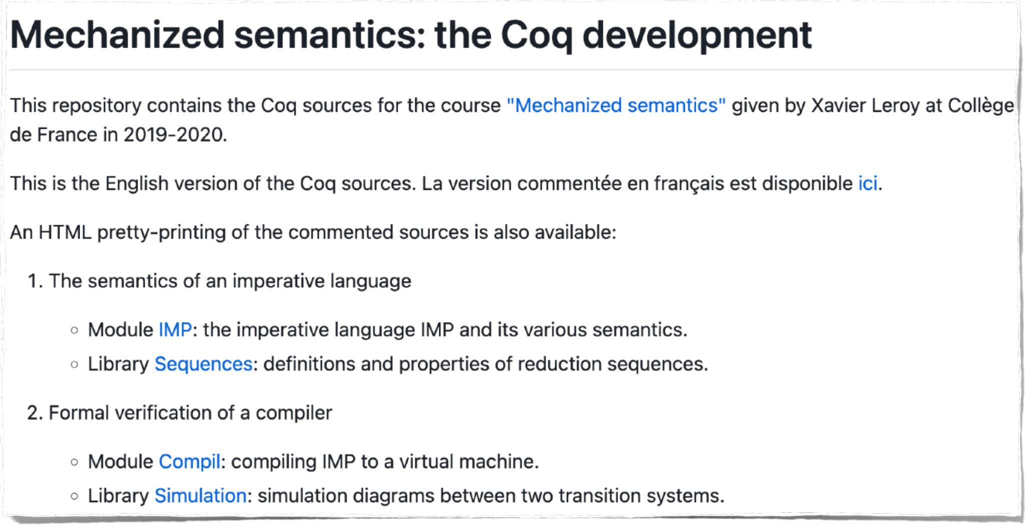
```
Definition example: com :=
<{ X := X + 1 }> .

Definition same_example: com :=
   CAss X (APlus (AId X) (ANum 1)) .
```

boolean expressions

```
Inductive ceval : com → state → state → Prop :=
   | E_Skip : ∀ st, st =[ skip ]=> st
   | E_WhileFalse : ∀ b st c, beval st b = false →
       st =[ while b do c end ]=> st
   | E_WhileTrue : ∀ st st' st'' b c, beval st b = true →
       st  =[ c ]=> st' →
       st' =[ while b do c end ]=> st'' →
       st  =[ while b do c end ]=> st''
   |   …
```

Extending the VM language: instruction set

compil.v 


Mechanized semantics: the Coq development

This repository contains the Coq sources for the course "Mechanized semantics" given by Xavier Leroy at Collège de France in 2019-2020.

This is the English version of the Coq sources. La version commentée en français est disponible ici.

An HTML pretty-printing of the commented sources is also available:

1. The semantics of an imperative language
   - Module IMP: the imperative language IMP and its various semantics.
   - Library Sequences: definitions and properties of reduction sequences.
2. Formal verification of a compiler
   - Module Compil: compiling IMP to a virtual machine.
   - Library Simulation: simulation diagrams between two transition systems.

```
Inductive instr: Type :=
  | Iconst (n: Z)       (* formerly SPush *)
  | Ivar (x: ident)     (* formerly SLoad *)
  | Iadd
  | Isetvar (x: ident) (* pop an integer and assign it to variable *)
  | Ibranch (d: Z)      (* skip forward or backward d instructions *)
  | Iopp                (* pop one integer, push its opposite *)
  | Ibeq (d1 d0: Z) (* pop 2 integers, skip d1 instructions if =, d0 if ≠ *)
  | Ible (d1 d0: Z) (* pop 2 integers, skip d1 instructions if ≤, if > *)
  | Ihalt.              (* stop execution *)

Definition code := list instr.
```

```
Definition ex_code1:code := Ivar "x" :: Iconst 1 :: Iadd :: Isetvar "x" :: nil.
Definition ex_code2:code :=
  Ivar "x" :: Iconst 1 :: Iadd :: Isetvar "x" :: Ibranch (-5) :: nil.
```

x := x + 1

VM semantics

compil.v

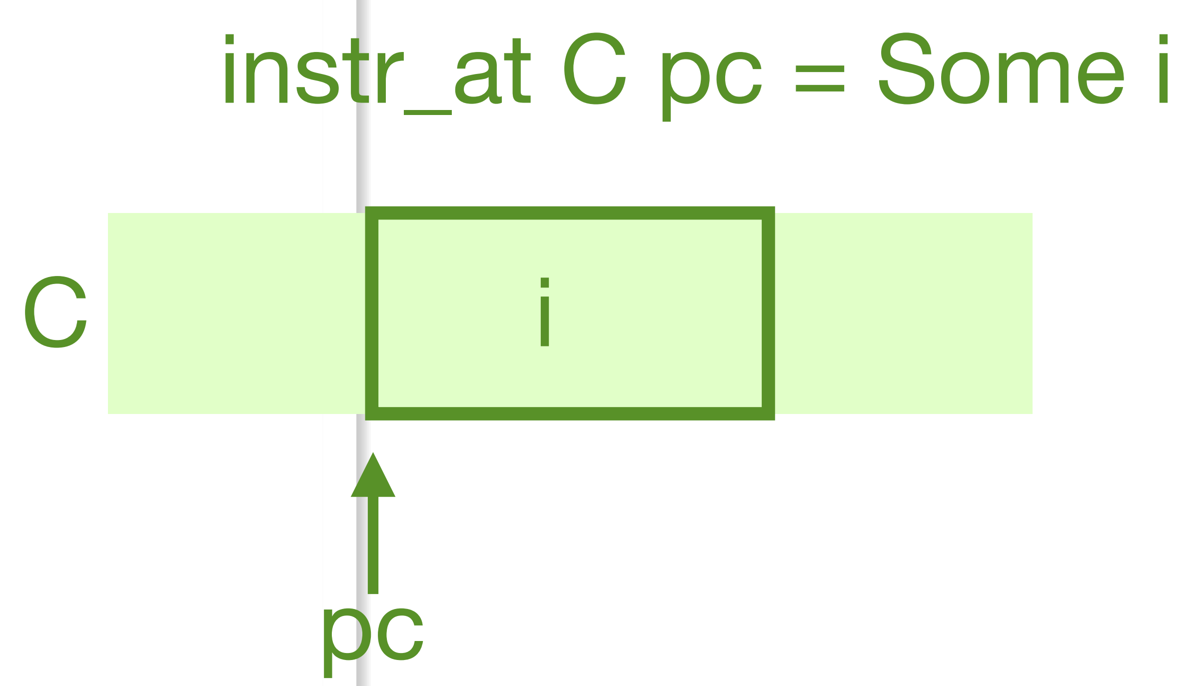
Small-step semantics, given by a transition relation $s \rightarrow s'$

formerly called state

position of the currently executing instruction

fixed list of instructions

```
Definition stack := list Z.
Definition store := ident → Z.
Definition config := (Z * stack * store).

Inductive transition (C:code): config → config → Prop :=
  | trans_const: ∀ pc stack s n,
      instr_at C pc = Some(Iconst n) →
      transition C (pc, stack, s) (pc + 1, n :: stack, s)
  | trans_setvar: ∀ pc stack s x n,
      instr_at C pc = Some(Isetvar x) →
      transition C (pc, n :: stack, s) (pc + 1, stack , update x n s)
  | trans_branch: ∀ pc stack s d pc',
      instr_at C pc = Some(Ibranch d) →
      pc' = pc + 1 + d →
      transition C (pc, stack, s) (pc', stack, s)
  | …
```



increments pc by 1

branch instructions increment by 1+d

Execution of VM programs

Small-step (operational) semantics

```
Definition transitions (C: code): config → config → Prop :=
  star (transition C).
```

reflexive transitive closure

initial states

final states

```
Definition machine_terminates (C: code) (s_init s_final: store) :=
  ∃ pc, transitions C (0, nil, s_init) (pc, nil, s_final)
          ∧ instr_at C pc = Some Ihalt.
```

Sequences of transitions and their properties

Sequences.v

$S \rightarrow S'$

```
Variable A: Type.                  (* type of states *)
Variable R: A → A → Prop.          (* transition relation between states *)
```

$S \rightarrow^{*} S'$

```
Inductive star: A → A → Prop :=
  | star_refl: ∀ a, star a a
  | star_step: ∀ a b c, R a b → star b c → star a c.
```

```
Lemma star_one: ∀ a b, R a b → star a b.
```

```
Lemma star_trans: ∀ a b, star a b → ∀ c, star b c → star a c.
```

$S \rightarrow^{+} S'$

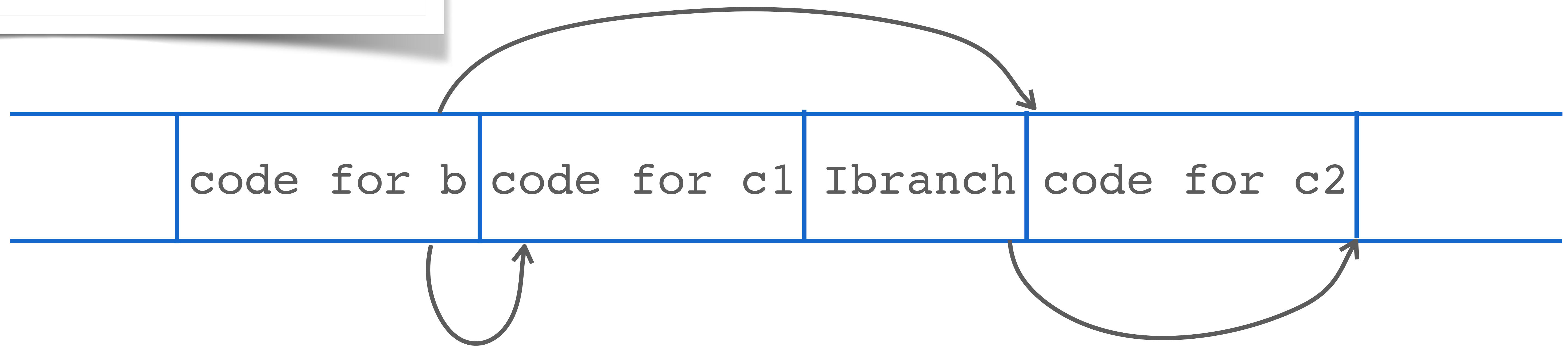
```
Inductive plus: A → A → Prop :=
  | plus_left: ∀ a b c, R a b → star b c → plus a c.
```

```
Lemma plus_star_trans: ∀ a b c, plus a b → star b c → plus a c.
```

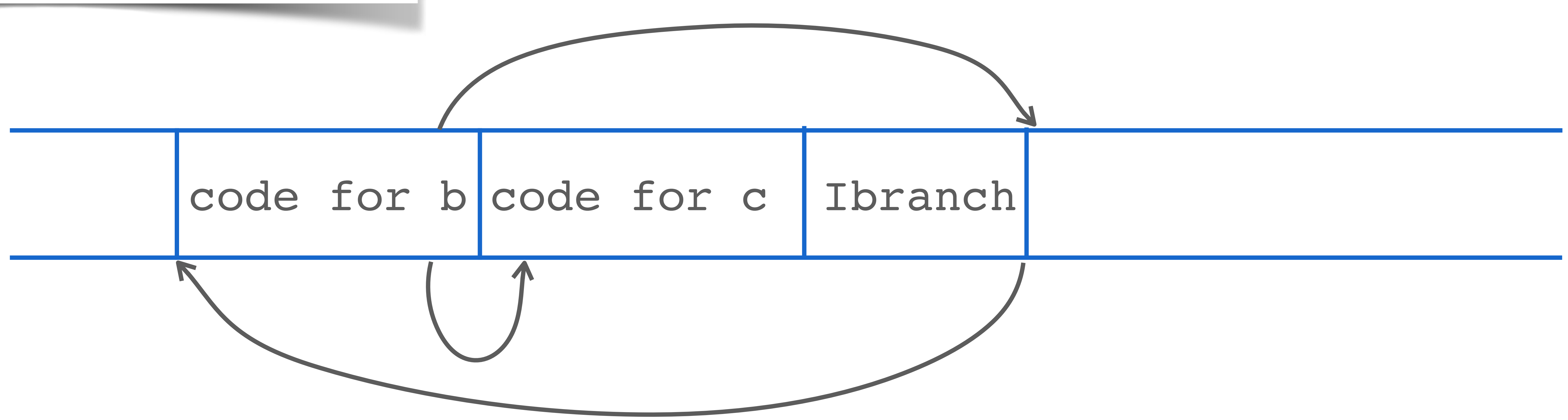
```
Definition irred (a:A): Prop :=   (* stuck states *)
 ∀ b, ~(R a b).
```

Compilation of commands

```
code for (CIf b c1 c2)
```



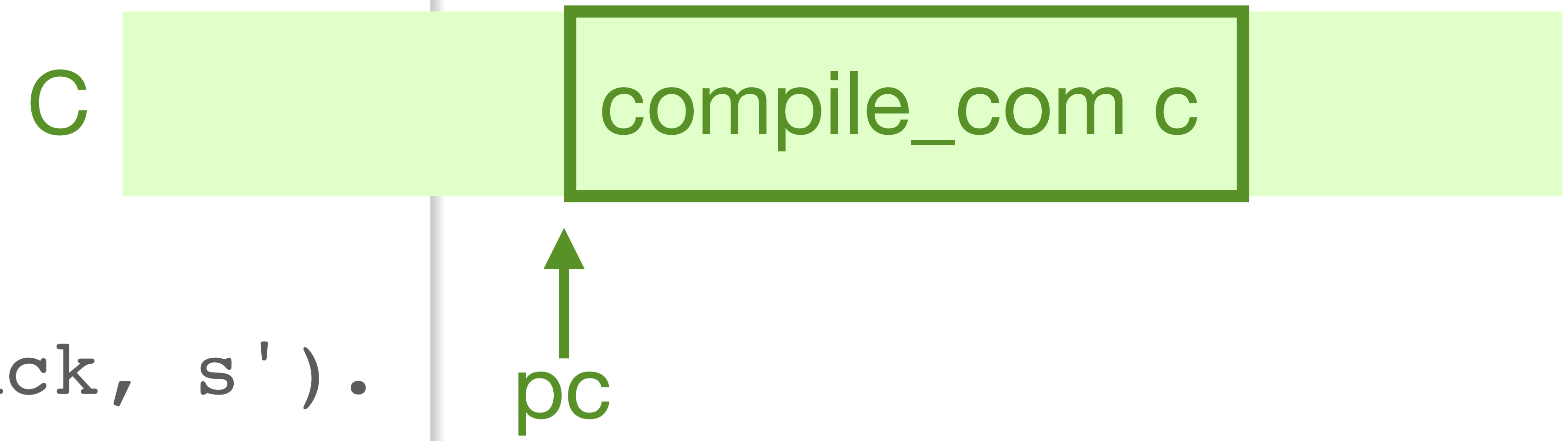

```
code for (CWhile b c)
```



Compiler correctness

compil.v

```
Lemma compile_com_correct_terminating:
  ∀ s c s',
  cexec s c s' →
  ∀ C pc stack,
  code_at C pc (compile_com c) →
  transitions C
      (pc, stack, s)
      (pc + codelen (compile_com c), stack, s').
```



ceval in Imp.v

remember s_compile_correct_aux!

```
Definition machine_terminates (C: code) (s_init s_final: store) :=
  ∃ pc, transitions C (0, nil, s_init) (pc, nil, s_final)
        ∧ instr_at C pc = Some Ihalt.
```

length of the list

proof by induction on the derivation of cexec s c s'

```
Definition compile_program (p: com) : code :=
  compile_com p ++ Ihalt :: nil.

Theorem compile_program_correct_terminating:
  ∀ s c s',
  cexec s c s' →
  machine_terminates (compile_program c) s s'.
```

Part 3: summary

« The generated code must behave as prescribed by the semantics of the source program. »

```
Theorem s_compile_correct: ∀ s e,
 s_execute s [] (s_compile e) = [aeval e].
```

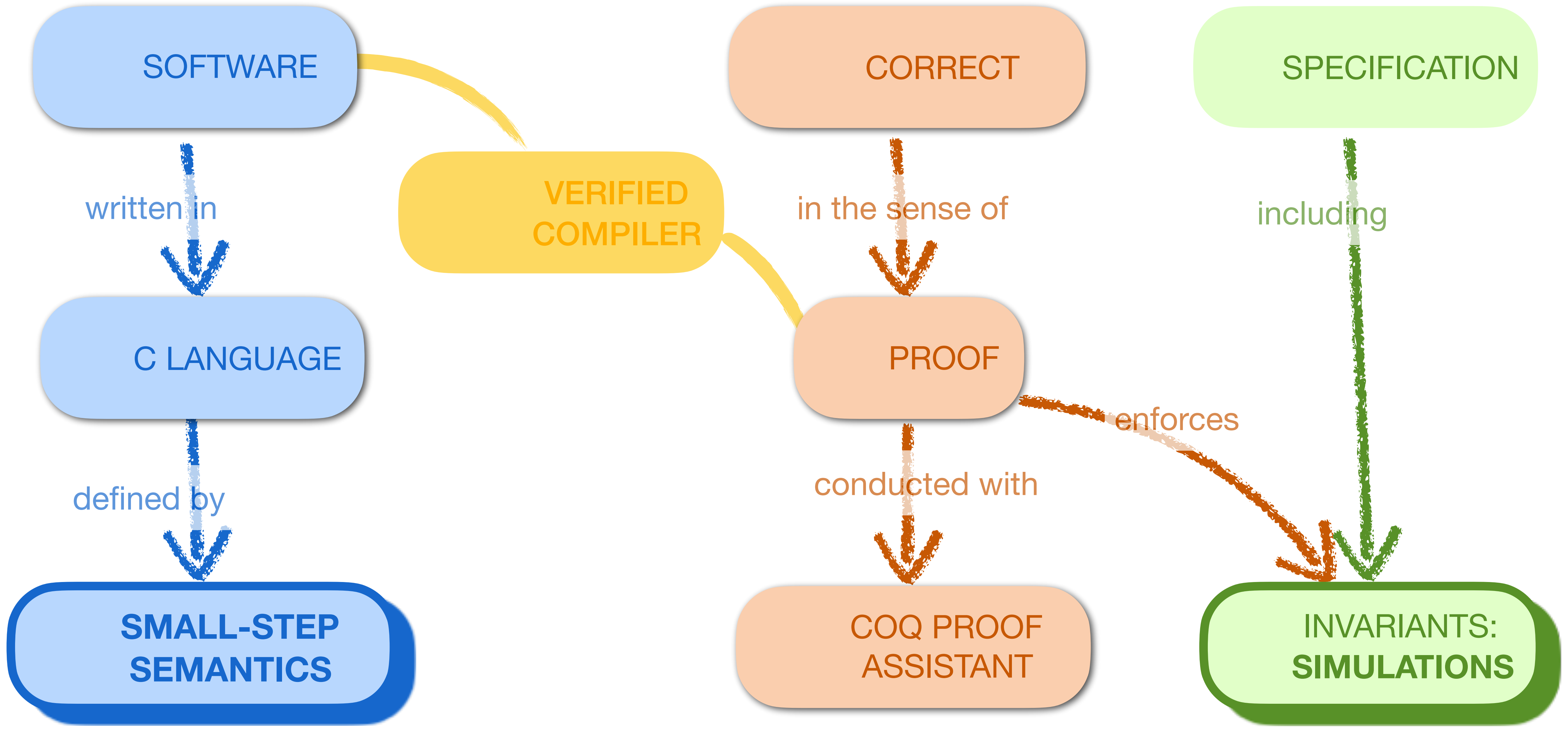
```
Theorem compile_program_correct_terminating:
  ∀ s c s',
  cexec s c s' →
  machine_terminates (compile_program c) s s'.
```

one big step

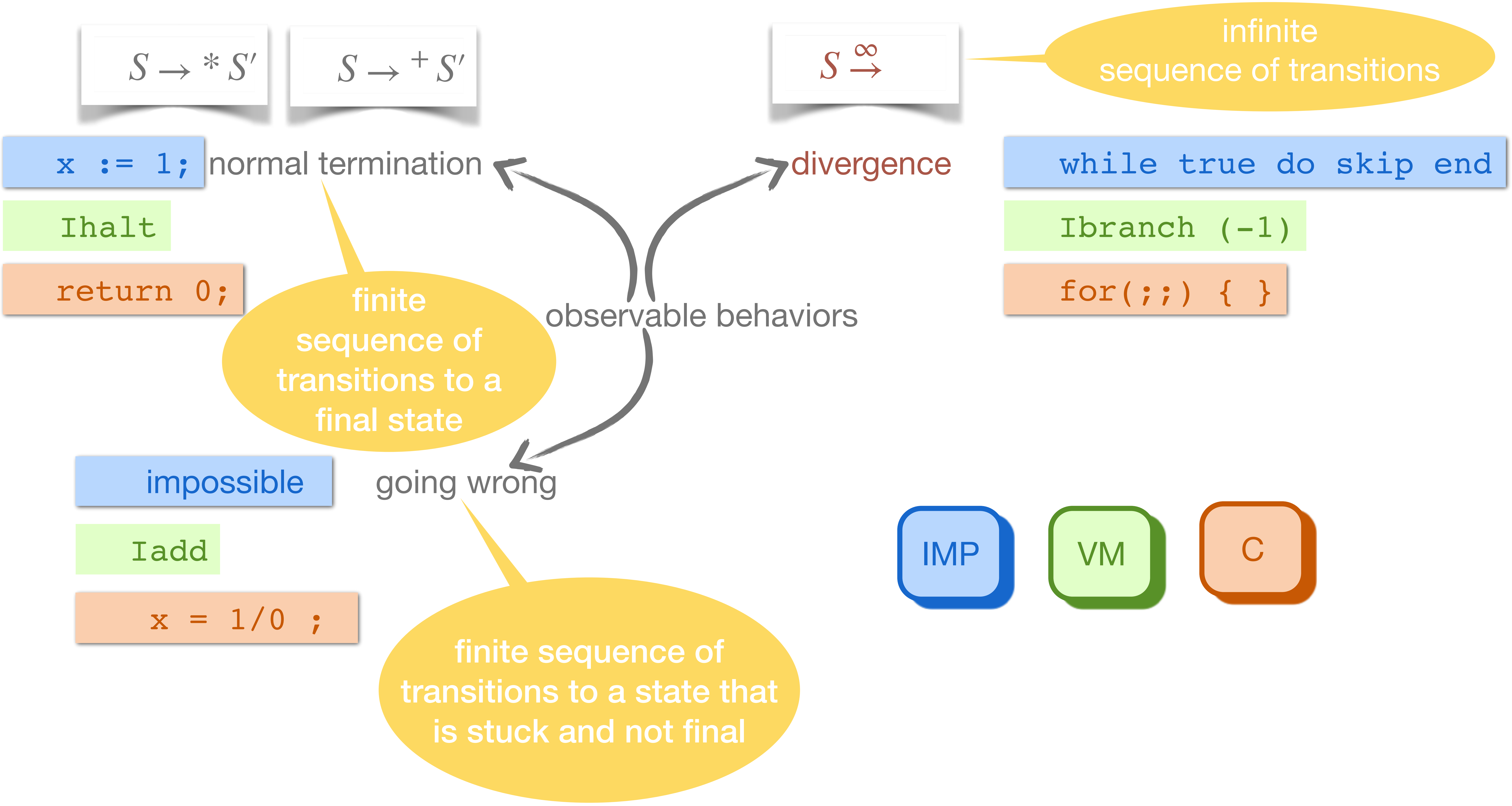
one or several small steps

This is not enough to conclude that the compiler is correct!

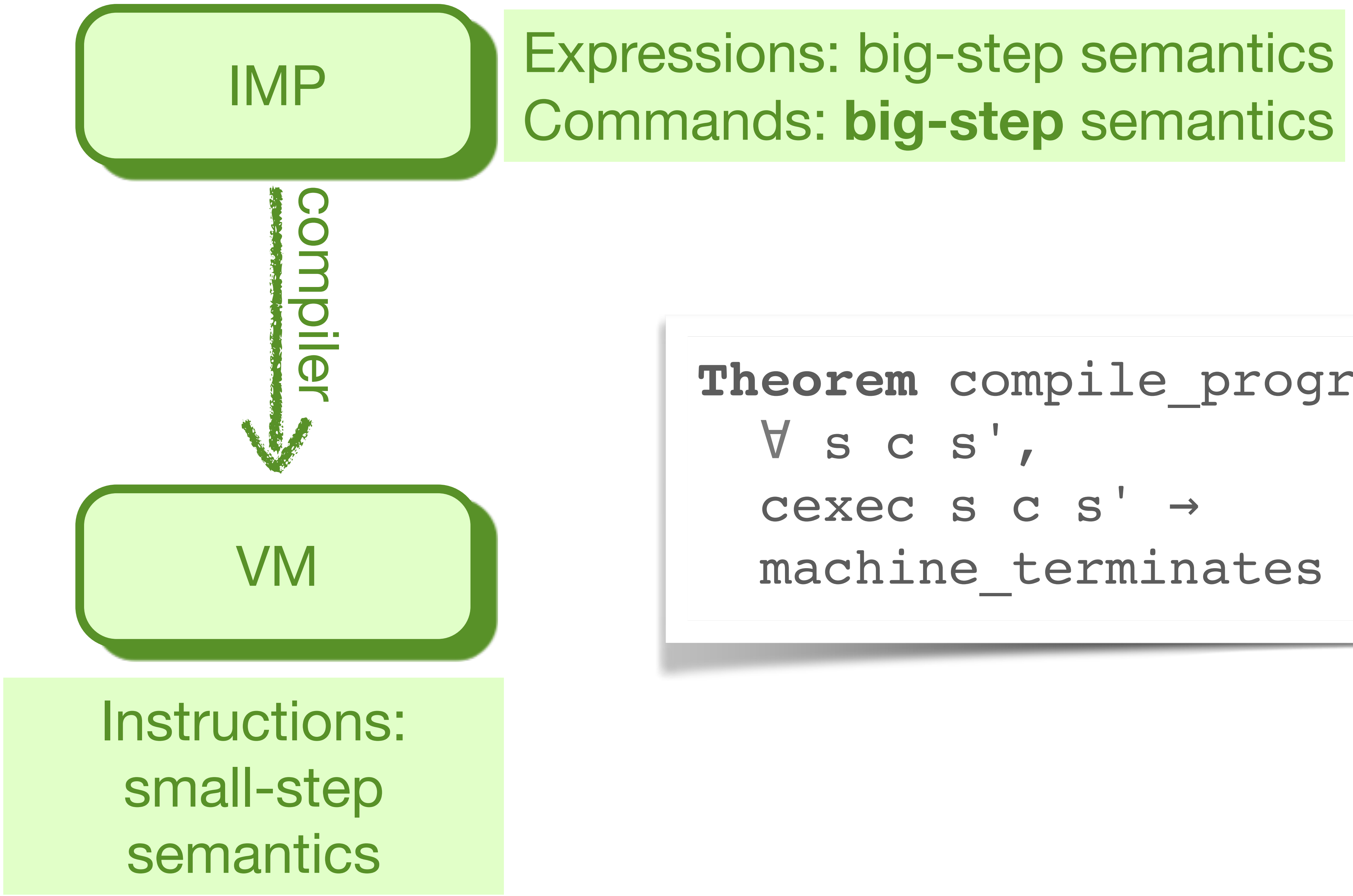
Part 4:
semantic preservation and compiler verification

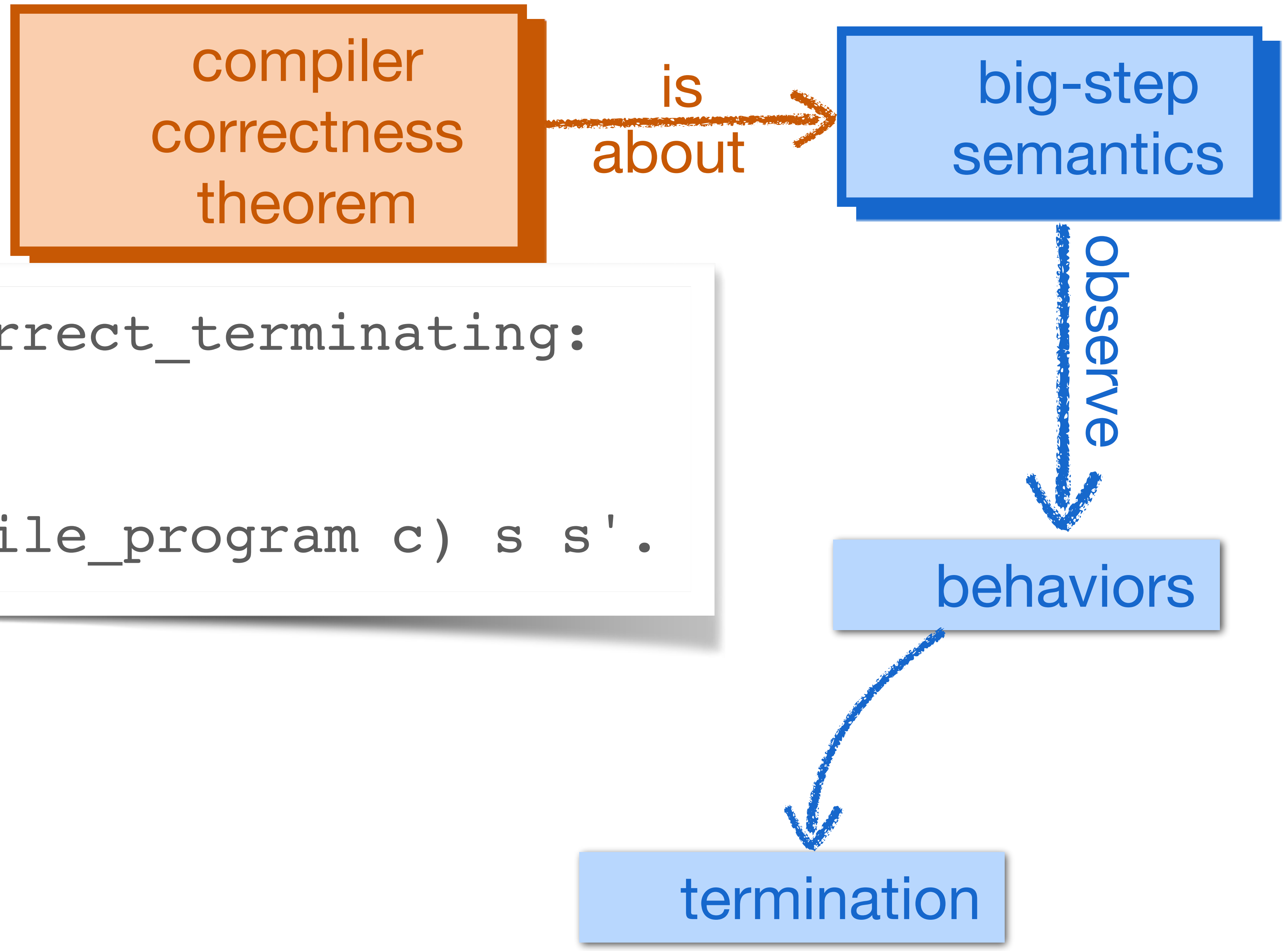


What should be preserved?
Observable behaviors



Summary of yesterday's lecture

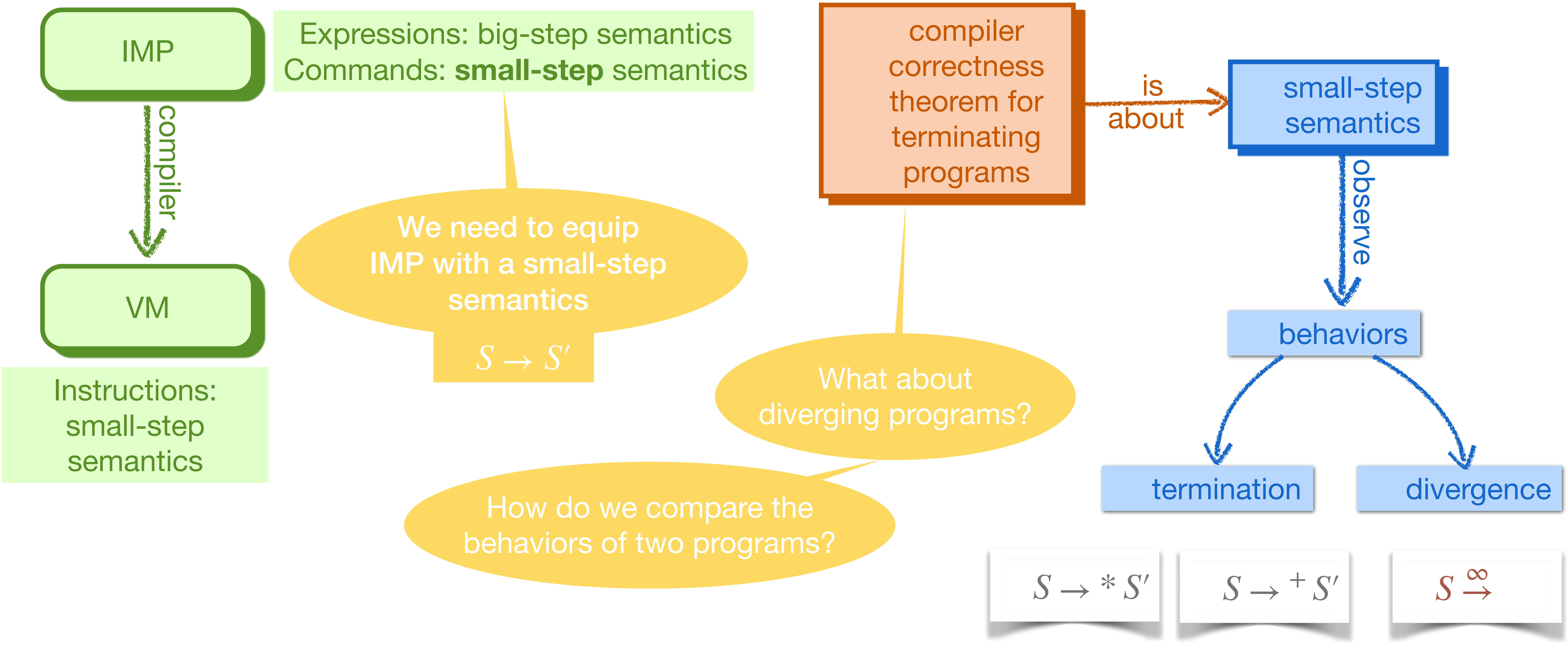





```
Theorem compile_program_correct_terminating:
  ∀ s c s',
  cexec s c s' →
  machine_terminates (compile_program c) s s'.
```

$$c/s \Rightarrow s'$$

Summary of yesterday's lecture



Should «going wrong» behaviors be preserved?

```
#include <stdio.h>
int main()
{
    int x;
    x = 1 / 0;
    return 0;
}
```

Compilers routinely optimize away going-wrong behaviors.

This program goes wrong.

However, the compiler eliminates `x=1/0;` as it is dead code.

Thus, the generated code always terminates.

Justifications

- We know that the program does not go wrong (e.g. by static analysis).
- It is the programmer's responsibility to avoid going-wrong behaviors (C standards).

Should «going wrong» behaviors be preserved?

```
#include <stdio.h>
int main()
{
   int x[2] = { 12, 34 };
   printf("x[2] = %d\n", x[2]);
   return 0;
}
```

This program goes wrong.

However, the code generated by the compiler does not check the array bounds.

The generated code may crash but in general it prints an arbitrary integer and terminates normally.

This out-of-bound access is an example of an undefined behavior (according to the ISO C standard).

Notions of semantic preservation: bisimulation

The source program S and the compiled program have exactly the same behaviors.

- Every possible behavior of S is a possible behavior of C.
- Every possible behavior of C is a possible behavior of S.

Example for the IMP to VM compiler

- (compile_com c) terminates if and only if c terminates, with the same final store
- (compile_com c) diverges if and only if c diverges
- (compile_com c) never goes wrong

Forward simulation

Forward simulation from a source program S to a compiled code C:
every possible behavior of S is a possible behavior of C

Example:

- theorem compile_program_correct_terminating
- If C diverges, (compile_com C) diverges

This looks insufficient: what if C has more behaviors than S? For instance, if C can terminate or go wrong?

Reducing non-determinism during compilation

A language is deterministic if every program has only one behavior.

The C language is not deterministic: the evaluation order is partially unspecified.

```
int x = 0;
int f(void) { x = x + 1; return x; }
int g(void) { x = x - 1; return x; }
```

The expression `f()+g()` can evaluate either to:

- 1 if `f()` is evaluated first (returning 1), then `g()` (returning 0);
- -1 if `g()` is evaluated first (returning 1), then `f()` (returning 0).

Every C compiler chooses one evaluation order at compile-time.
The compiled code therefore has fewer behaviors than the source program (1 instead of 2). Forward simulation and bisimulation fail.

Backward simulation

Backward simulation from a source program S to a compiled code C:
every possible behavior of C is a possible behavior of S.
However, C may have fewer behaviors than S.

If the target language is deterministic, forward simulation implies backward simulation, and therefore bisimulation.

Simulations for safe programs

A program is **safe** when it either terminates or diverges.

Safe forward simulation: any behavior of the source program S other than « going wrong » is a possible behavior of the compiled code C.

Safe backward simulation: for any behavior b of the compiled code C, the source program S can either have behavior b or go wrong.

Simulation diagrams

Behaviors are defined in terms of sequences of transitions.

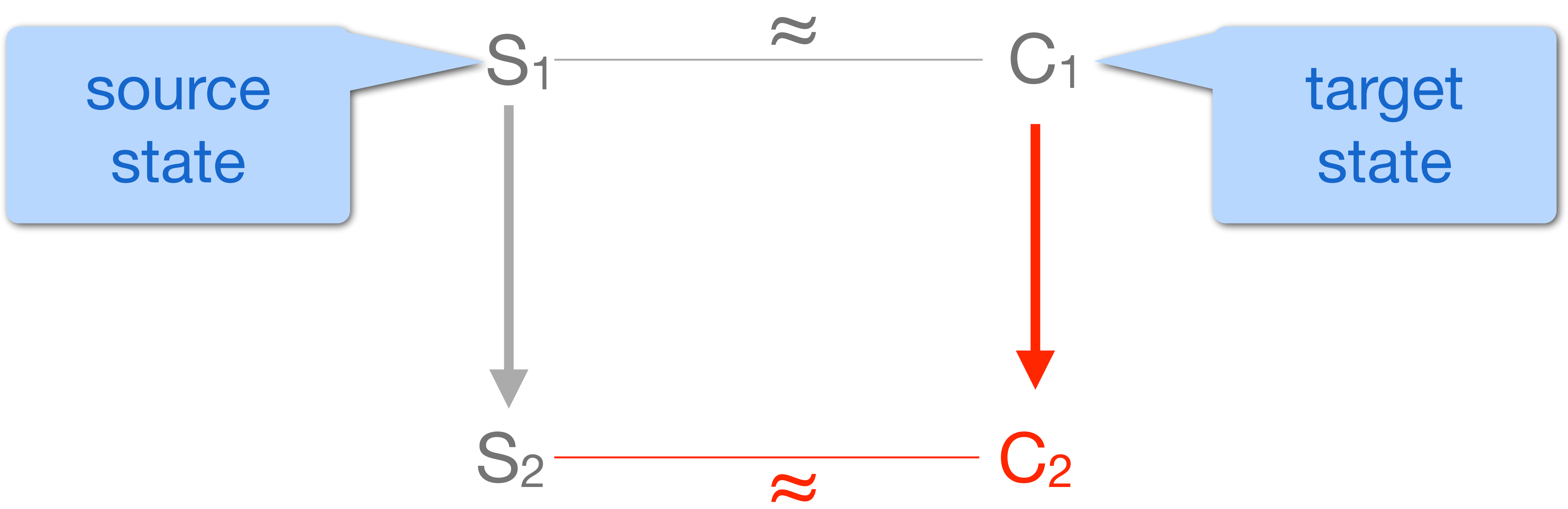
Forward simulation from a source program S to a compiled code C can be proved as follows:

- show that every transition in S is simulated by some transitions in C
- while preserving an invariant ≈ between the states of S and C

Backward simulation is similar but simulates transitions of C by transitions of S.

Lock-step simulation

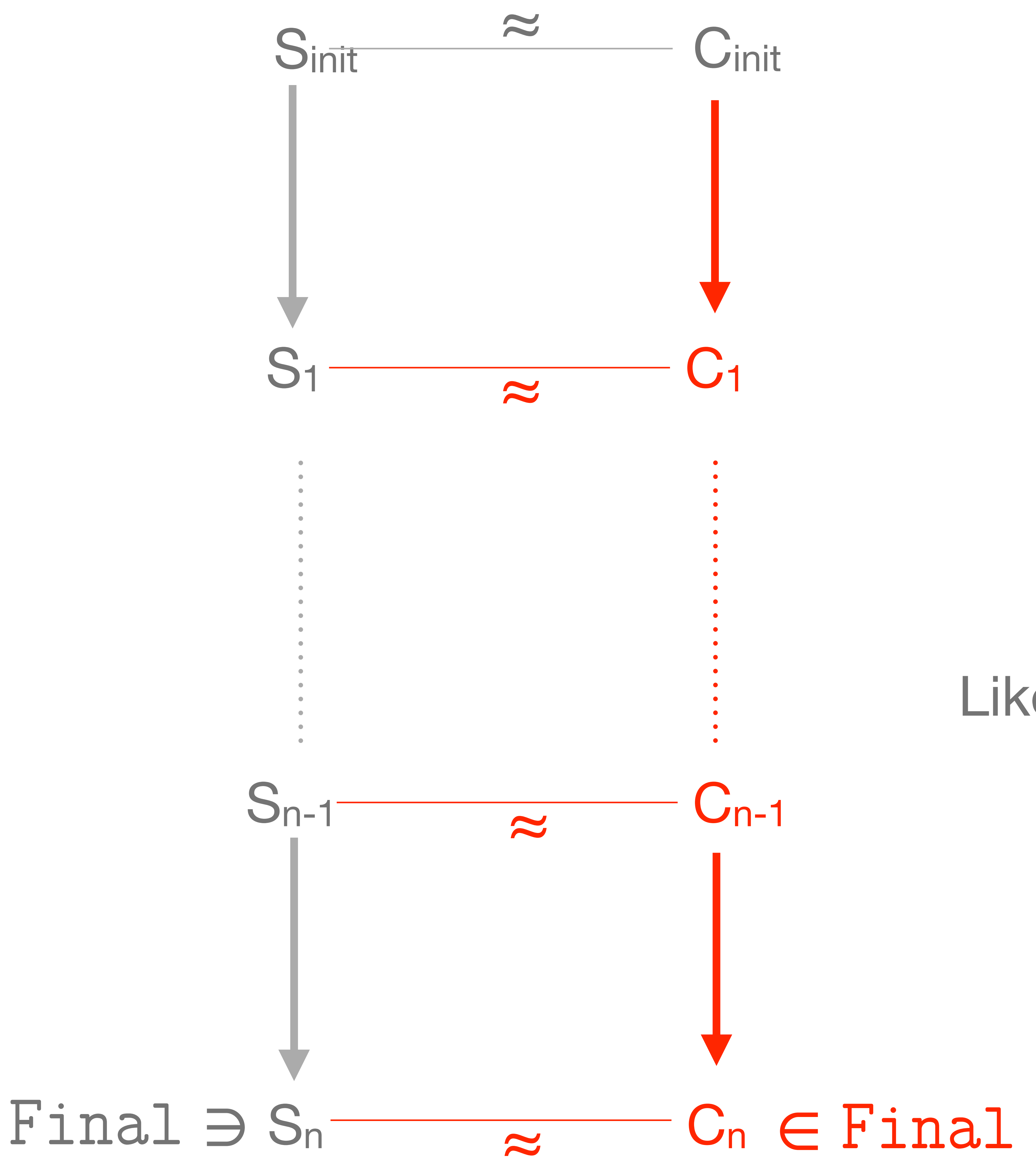
Every transition in the source S is simulated by exactly one transition in the compiled code C



Further show that initial states are related: $S_{init} \approx C_{init}$

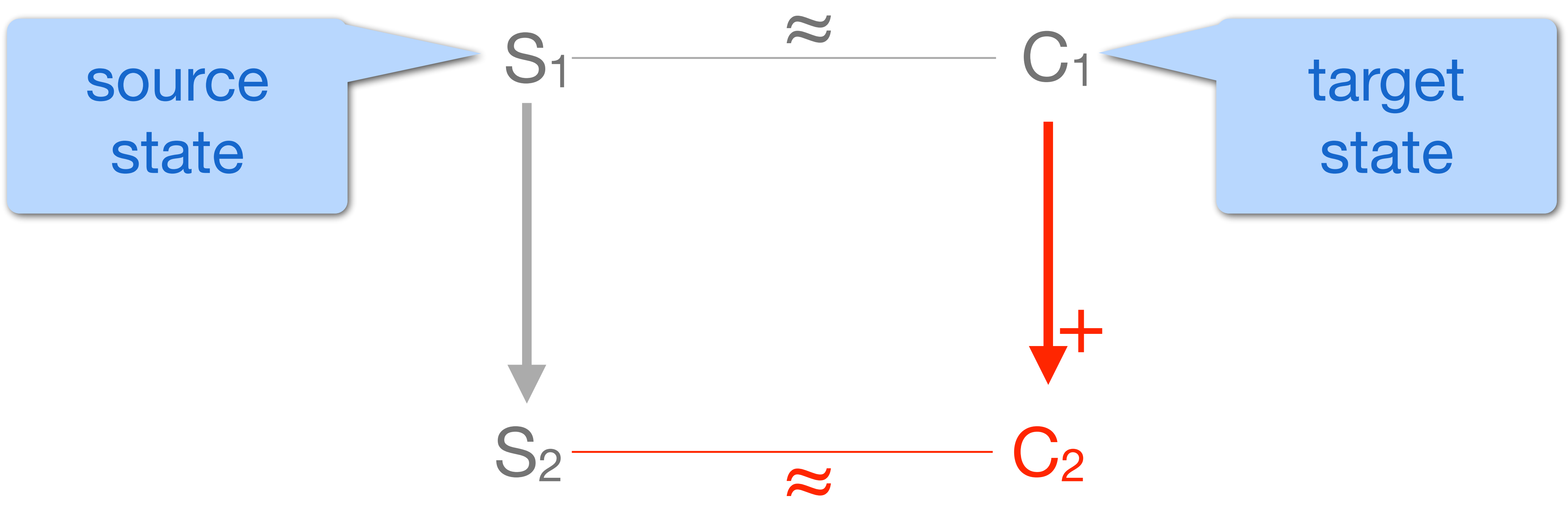
and final states are related: $S \approx C \wedge S \in \mathtt{Final} \Rightarrow C \in \mathtt{Final}$

From lock-step simulation to forward simulation



Likewise if S_{init} makes an infinity of transitions

Plus simulation



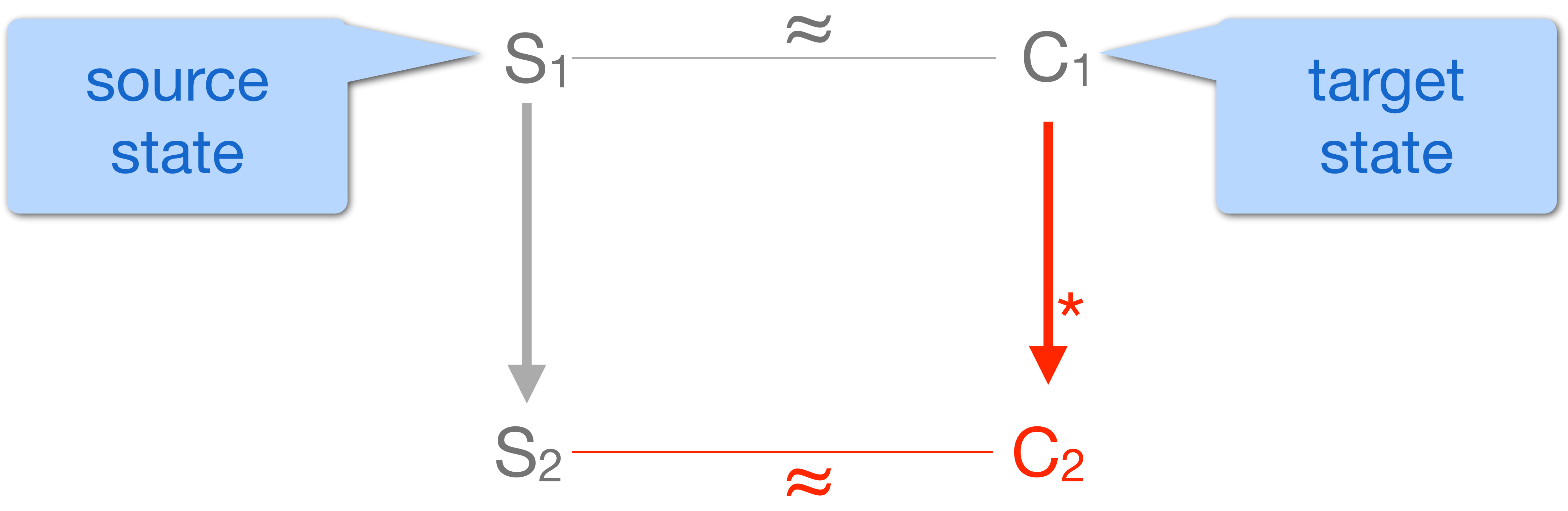
Example: compilation of `x := x + 1` into

```
Ivar "x" :: Iconst 1 :: Iadd :: Isetvar "x" :: nil
```

(already seen on this slide)

Forward simulation still holds

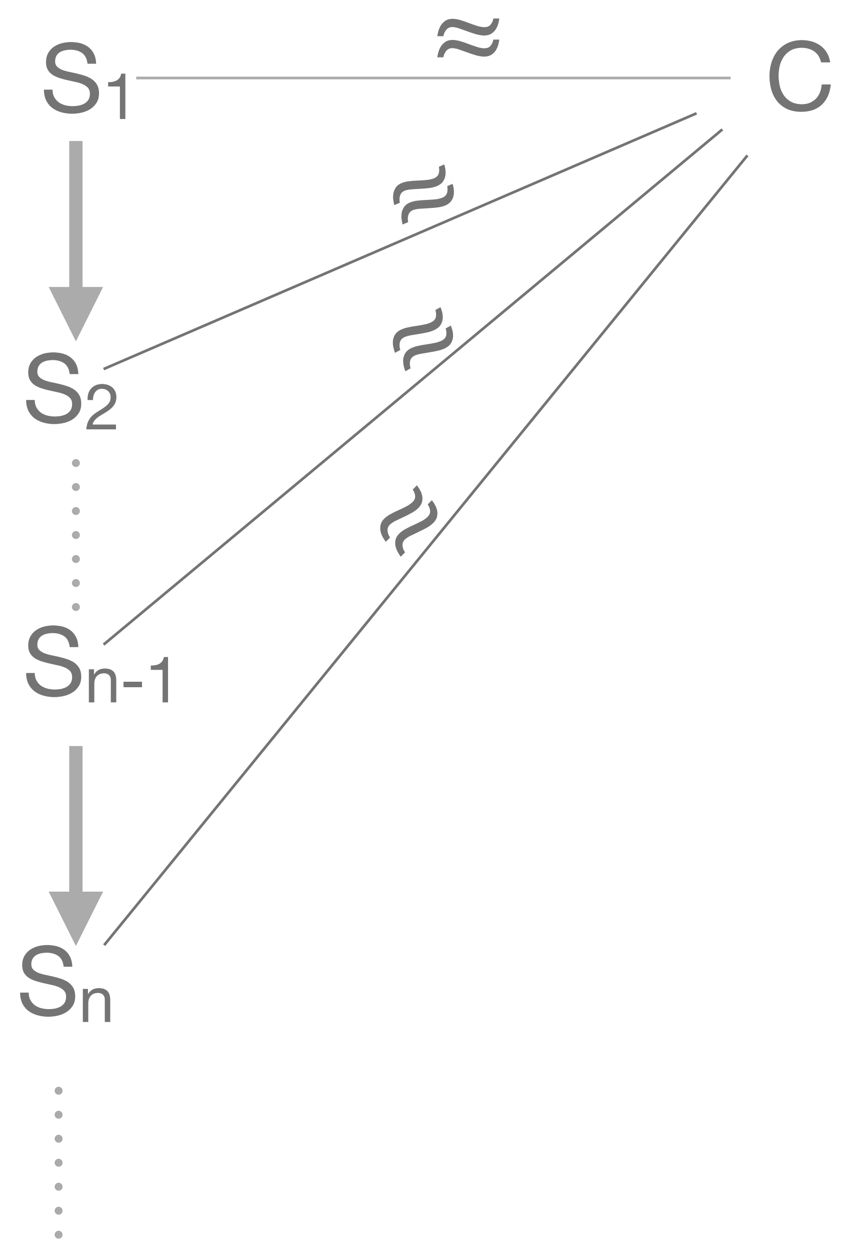
Incorrect star simulation



Forward simulation is not guaranteed:

- terminating executions are preserved,
- but diverging executions may not be preserved

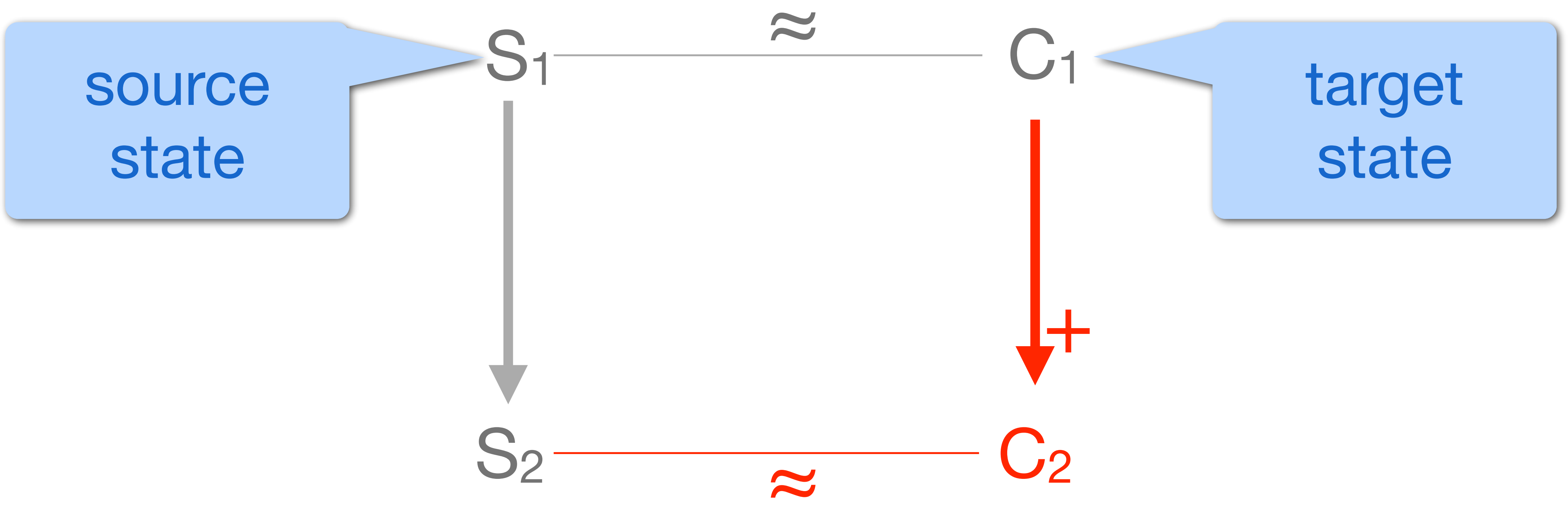
The problem of infinite stuttering



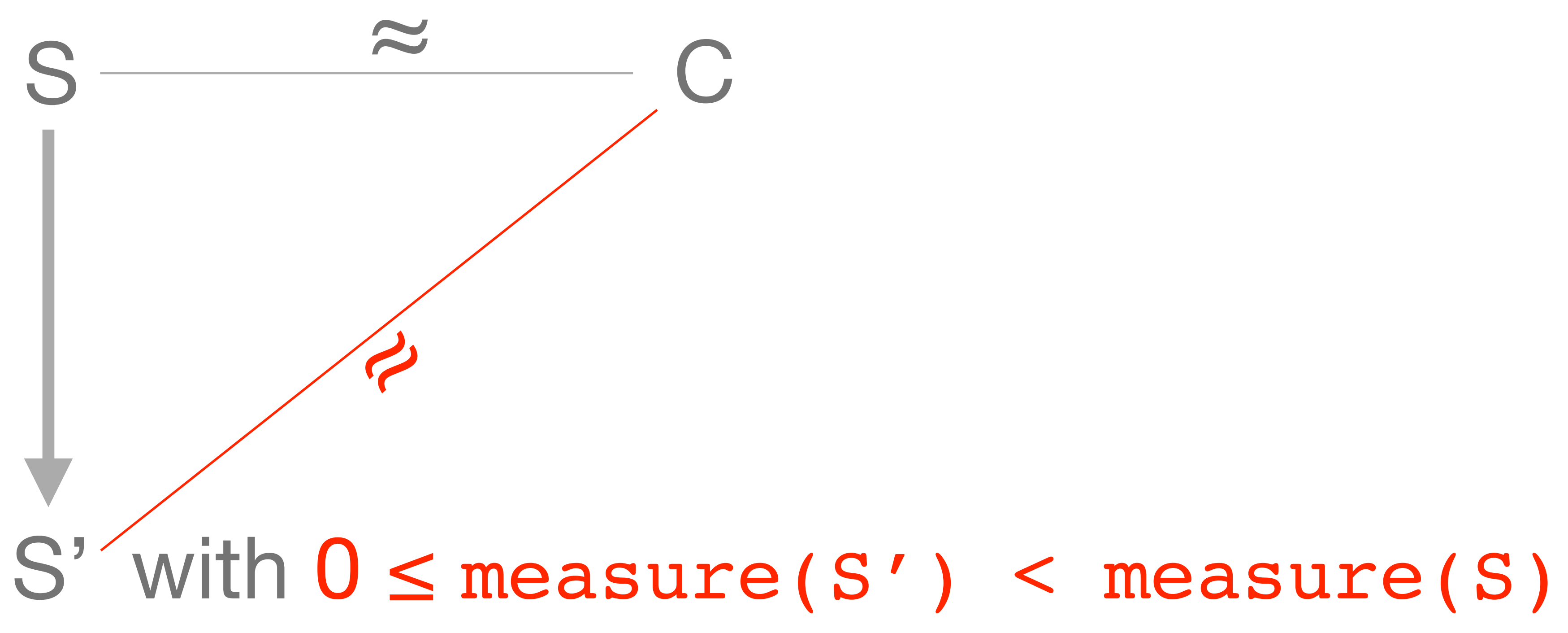
The source program diverges but the compiled code can terminate normally or by going wrong.

This denotes an incorrect optimization of a diverging program,
e.g. compiling `(while true skip)` into `skip`

Corrected star simulation



or



`measure(S):nat` from source states (could be to a well-founded set)

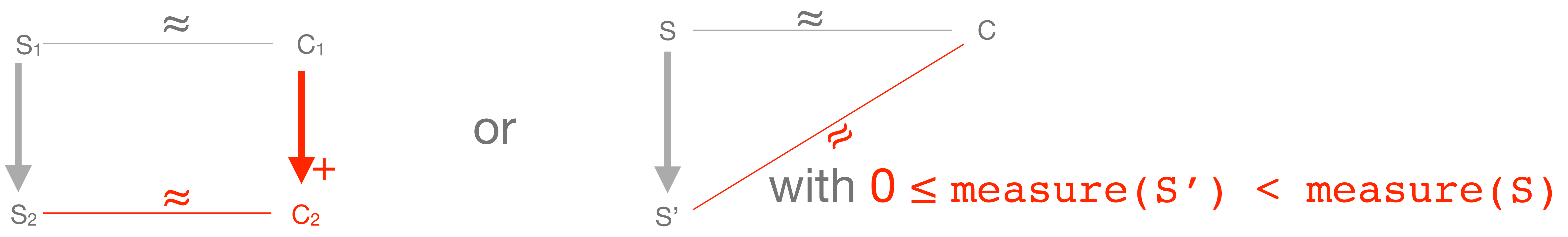
If the source program diverges, it must perform infinitely many non-stuttering steps, so the compiled code executes infinitely many transitions.

Coq library for star simulations: from star simulation to forward simulation

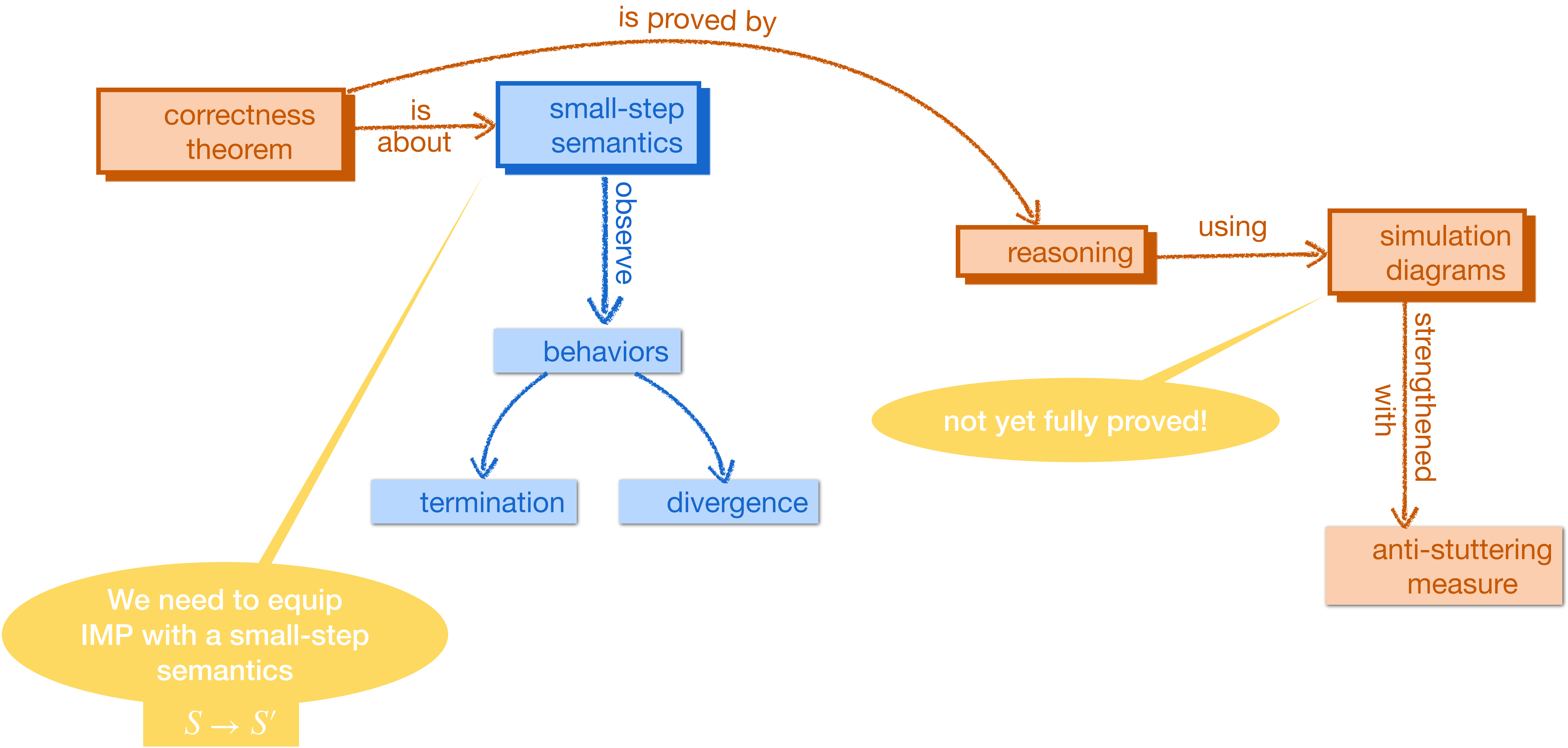
Simulation.v

```
Variable C1: Type.               (* the type of configurations for the source program *)
Variable step1: C1 → C1 → Prop.  (* its transition relation *)

Variable C2: Type.               (* the type of configurations for the transformed program *)
Variable step2: C2 → C2 → Prop.  (* its transition relation *)

Variable inv: C1 → C2 → Prop.    (* the invariant ≈ *)
Variable measure: C1 → nat.      (* the measure that prevents infinite stuttering *)
```

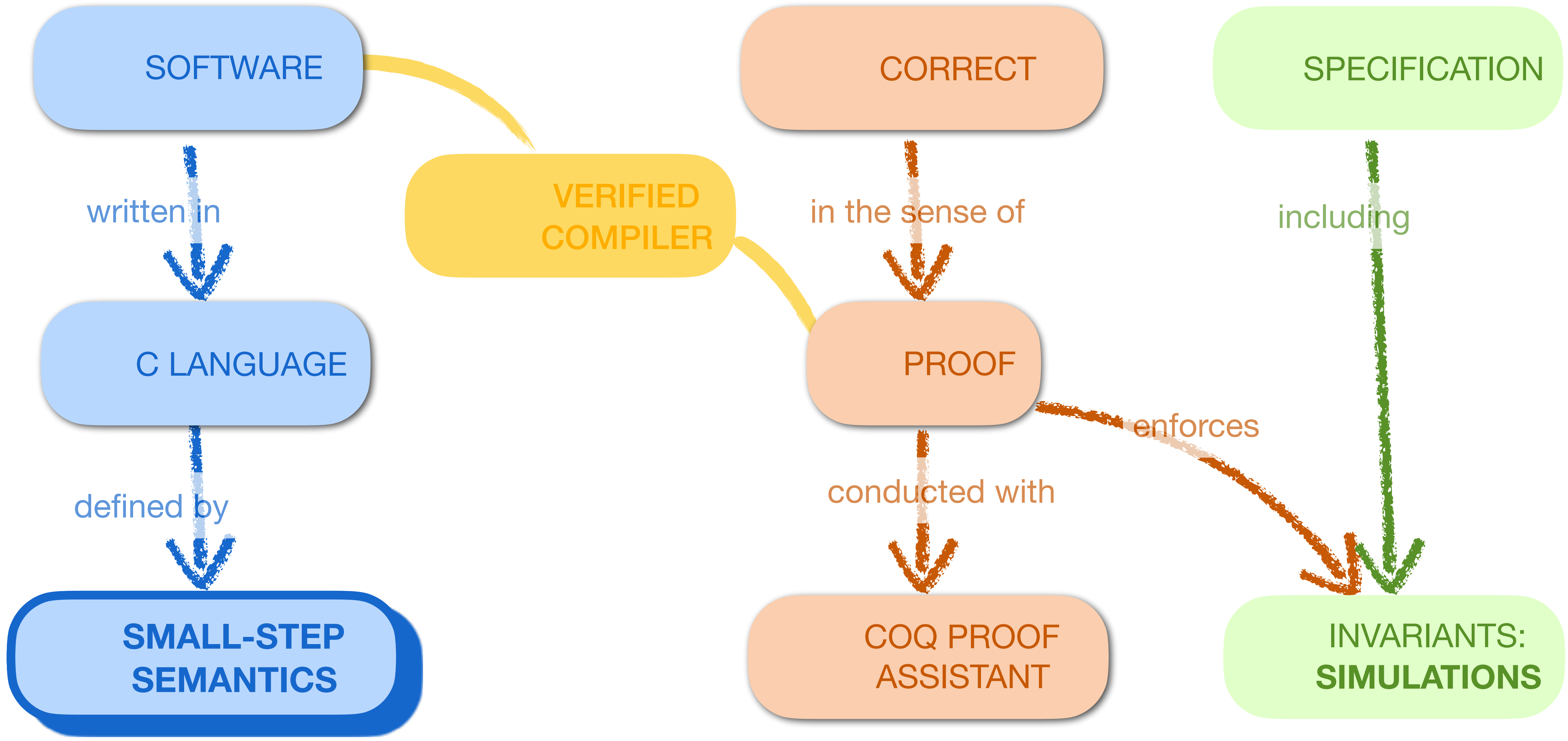
```
Hypothesis simulation:
  ∀ c1 c1', step1 c1 c1' →
  ∀ c2, inv c1 c2 →
  ∃ c2', (plus step2 c2 c2' ∨ (star step2 c2 c2' ∧ measure c1' < measure c1))
         ∧ inv c1' c2'.
```



Part 4: summary



Part 5:
small-step semantics and compiler verification



A small-step semantics for IMP

Relation: c / s → c' / s'

big-step semantics for expressions

x := a / s → skip / x ↦(aeval a s); s

notation used in Imp.v

(c; skip) / s → c / s

$$\frac{c1 / s1 \to c2 / s2}{(c1 ; c) / s1 \to (c2 ; c) / s2}$$

$$\frac{\text{eval s b = true}}{\text{(if b then c1 else c2) / s} \to \text{c1 / s}}$$

$$\frac{\text{eval s b = false}}{\text{(if b then c1 else c2) / s} \to \text{c2 / s}}$$

$$\frac{\text{eval s b = false}}{\text{(while b do c end) / s} \to \text{skip / s}}$$

$$\frac{\text{eval s b = true}}{\text{(while b do c end) / s} \to \text{c; while b do c end / s}}$$

Equivalence with big-step semantics

IMP.v

A classic result:

$c/s \Rightarrow s'$ if and only if $c/s \rightarrow * \ \texttt{skip}/s'$

This proof is useful to build confidence in both semantics

Spontaneous generation of commands

Some rules generate fresh commands that are not subterms of the source program.

(if b then c1 else c2); c / s → (c1; c) / s

Raises two issues when using simulation diagrams:

- impractical to reason on the execution relation
- difficult to define the measure

Small-step semantics with continuations

Instead of rewriting whole commands: c / s → c’ / s’

rewrite pairs of (subcommand under focus, continuation): c / k / s → c’ / k’ / s’

Continuation

- remainder of command
- context in which it occurs (control stack)

  Kstop nothing remains to be done

  c • k execution of a sequence of two commands

  Kwhile b c k execution of a loop

Small-step semantics with continuations

c / k / s → c’ / k’ / s’

No generation of fresh commands: c’ is always a subterm of c

(if b then c1 else c2) / k / s → c1 / k / s   when eval s b = true

New kinds of rules for dealing with continuations

(c1;c2) / k / s → c1 / c2 • k / s

Focus (on the left of a sequence)

skip / c • k / s → c / k / s

Resume (the remaining computations)

A small-step semantics for IMP

c / k / s → c’ / k’ / s’

x := a / k / s → skip / k / x ↦(aeval a s); s

(c1 ; c2) / k / s → c1 / c2 ● k / s

$$\frac{\text{eval s b = true}}{\text{(if b then c1 else c2) / k / s} \to \text{c1 / k / s}}$$

$$\frac{\text{eval s b = false}}{\text{(if b then c1 else c2) / k / s} \to \text{c2 / k / s}}$$

$$\frac{\text{eval s b = false}}{\text{(while b do c end) / k / s} \to \text{skip / k / s}}$$

$$\frac{\text{eval s b = true}}{\text{(while b do c end) / k / s} \to \text{c; while b do c end / Kwhile b c k / s}}$$

skip / c ● k / s → c / k / s

skip / Kwhile b c k / s → while b do c end / k / s

Program execution

Termination

```
Definition kterminates (s: store) (c: com) (s': store) :=
  star step (c, Kstop, s) (SKIP, Kstop, s’).
```

Divergence

```
Definition kdiverges (s: store) (c: com) :=
  infseq step (c, Kstop, s).
```

Equivalence between small-step semantics

```
Theorem equiv_smallstep_terminates:
  ∀ s c s', terminates s c s' ↔ kterminates s c s'.

Theorem equiv_smallstep_diverges:
 ∀ s c, diverges s c ↔ kdiverges s c.
```

Full proof of compiler correctness
Simulation diagram

with $0 \leq$ `measure(c₂,k₂) < measure(c₁,k₁)`

Difficulties

- find the invariant ≈ between source and target states
- find the measure from source states to a natural number

Full proof of compiler correctness
The anti-stuttering measure

When do the source program stutter? When no VM instruction is executed.

(c1 ; c2) / k / s → c1 / c2 • k / s

skip / c • k / s → c / k / s

(if true then c1 else c2) / k / s → c1 / k / s

(while true do c end) / k / s → c; while b do c end / Kwhile b c k / s

`measure(c,k):` sum of the sizes of `c` and all the commands appearing in `k`

length of the list

Full proof of compiler correctness
The simulation invariant

Remember this slide:

```
Lemma compile_com_correct_terminating:
  ∀ s c s', ceval s c s' →
  ∀ C pc stack,
  code_at C pc (compile_com c) →
  transitions C (pc, stack, s)
    (pc + codelen(compile_com c), stack, s').
```

C ⊢ c/k/s ≈ (pc, stack, s') is defined as:

- s = s'
- stack = [ ]
- `code_at C pc (compile_com c)`
- C contains compiled code matching k at `pc + codelen(compile_com c)`

Compiler correctness: wrapping up

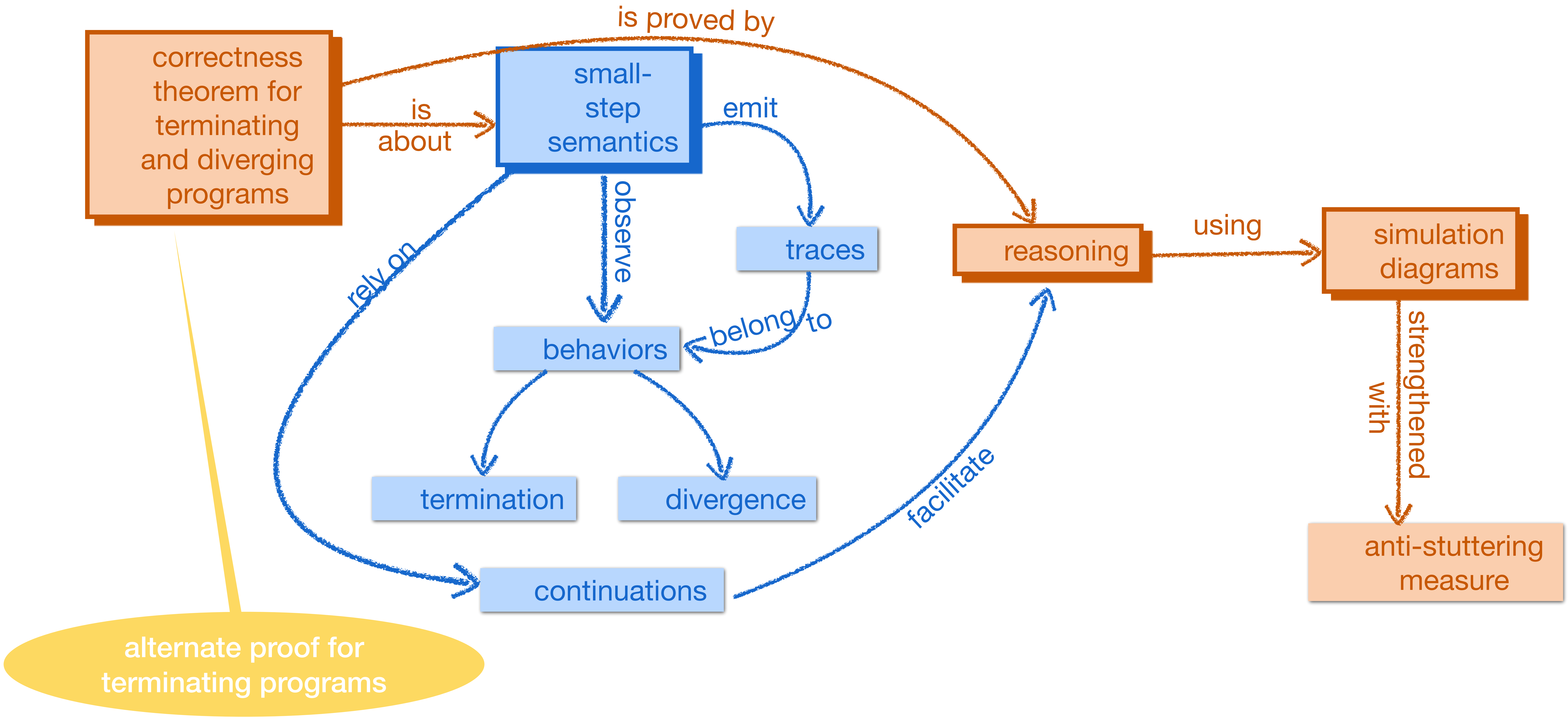
compil.v

```
Theorem compile_program_correct_terminating:
  ∀ s c s',
  ceval s c s' →
  machine_terminates (compile_program c) s s'.
```

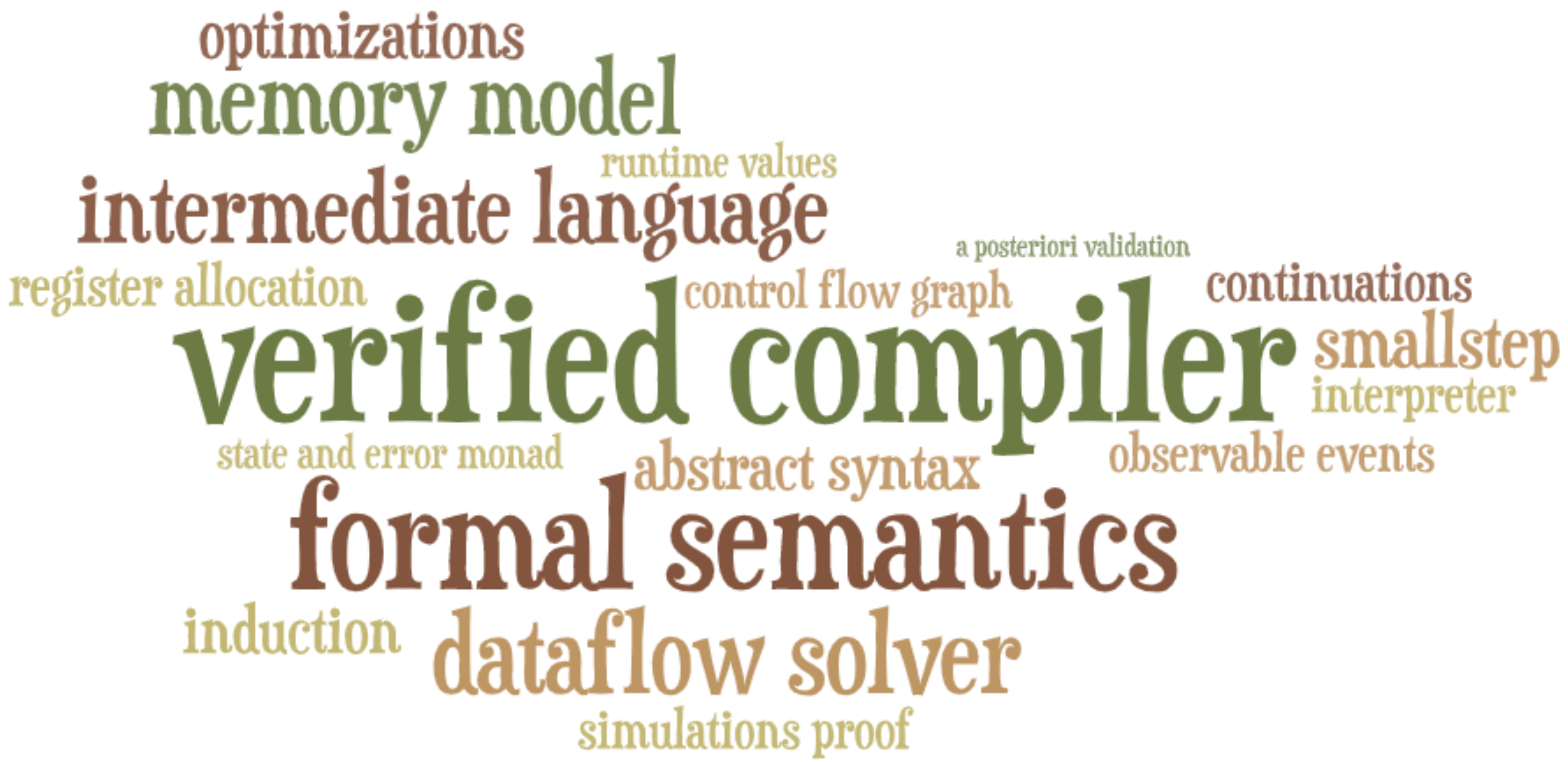
```
Theorem compile_program_correct_terminating_2:
  ∀ s c s',
  star step (c, Kstop, s) (SKIP, Kstop, s') →
  machine_terminates (compile_program c) s s'.
```

```
Theorem compile_program_correct_diverging:
  ∀ c s,
  infseq step (c, Kstop, s) →
  machine_diverges (compile_program c) s.
```

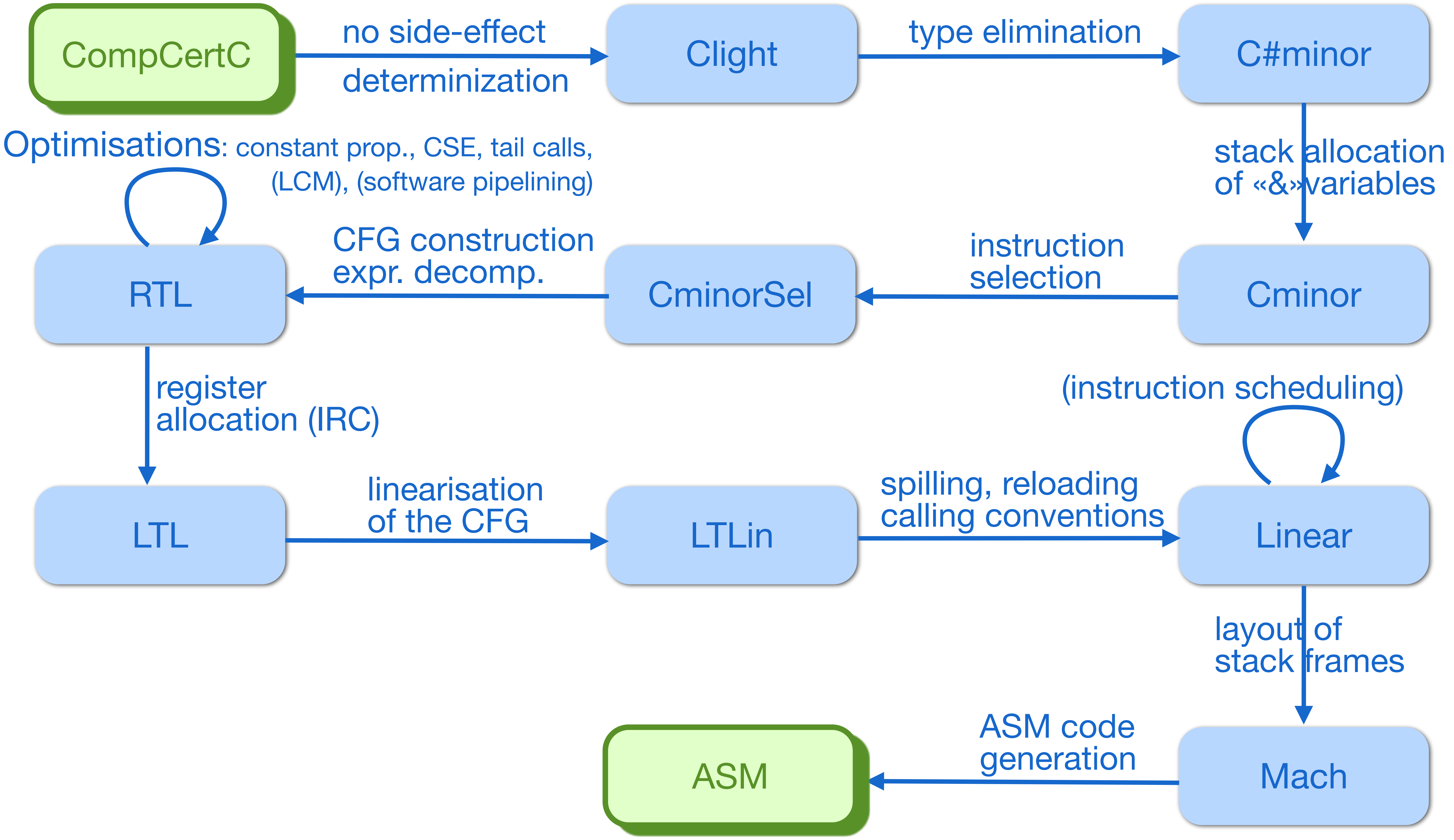
Part 5: summary



Part 6
How to turn CompCert
from a prototype in a lab
into a real-world compiler?



CompCert compiler: 11 languages, 18 passes



Multiplicity of source behaviors
Reducing non-determinism during compilation

The C language is not deterministic: the evaluation order is partially unspecified.

```
int x = 0;
int f(void) { x = x + 1; return x; }
int g(void) { x = x - 1; return x; }
```

The expression `f()+g()` can evaluate either to:

- 1 if `f()` is evaluated first (returning 1), then `g()` (returning 0);
- -1 if `g()` is evaluated first (returning 1), then `f()` (returning 0).

Every C compiler chooses one evaluation order at compile-time.
The compiled code therefore has fewer behaviors than the source program (1 instead of 2). **Forward simulation fails.**

Back to simulations

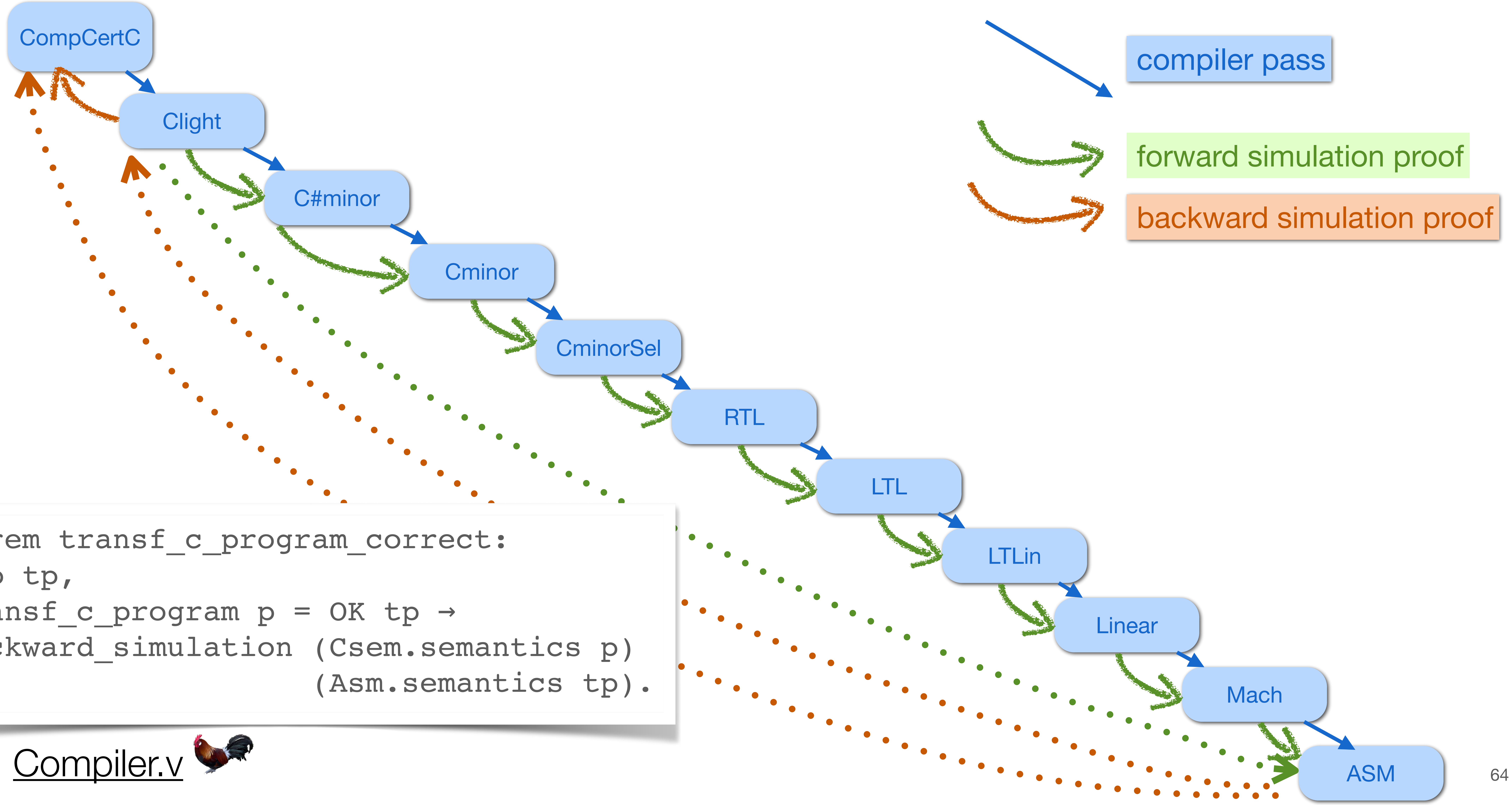
S: source program   C: compiled program

Backward simulation: every possible behavior of C is a possible behavior of S

Safe backward simulation: for any behavior b of C, S can have either behavior b or go wrong

If the target language is deterministic, forward simulation implies backward simulation (and therefore bisimulation)

Handling multiple compilation passes

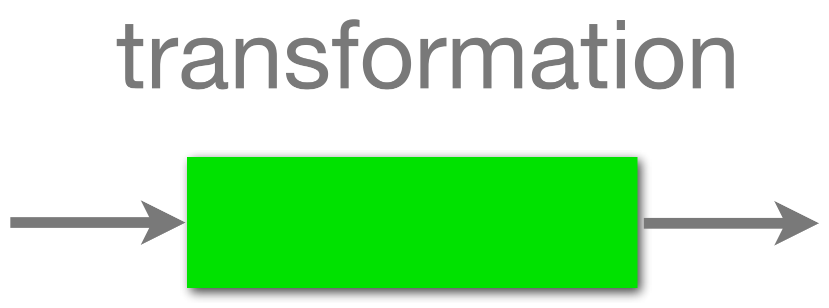



```
Theorem transf_c_program_correct:
  ∀ p tp,
  transf_c_program p = OK tp →
  backward_simulation (Csem.semantics p)
                      (Asm.semantics tp).
```

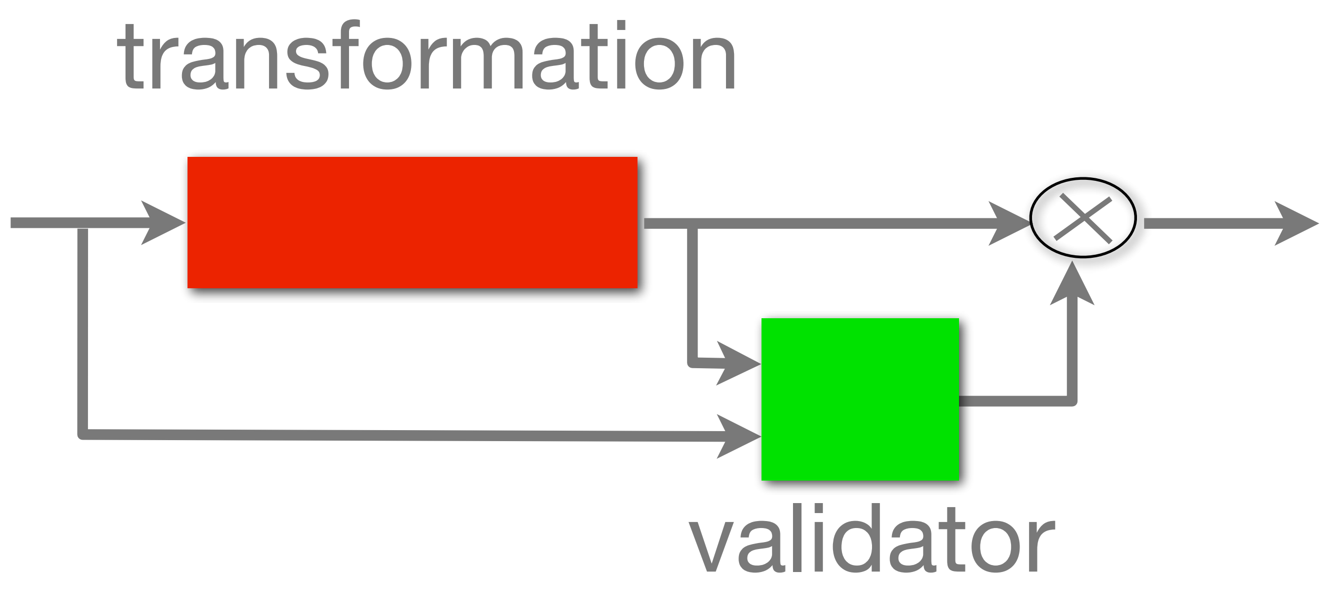
Compiler.v

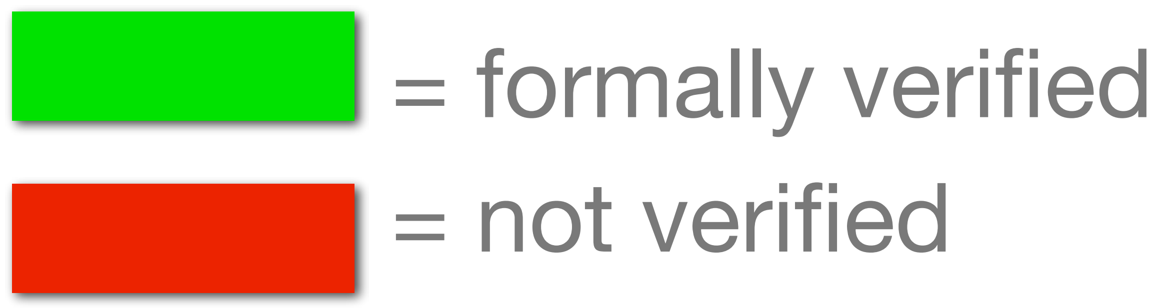
Verification patterns

Verified transformation



Verified translation validation

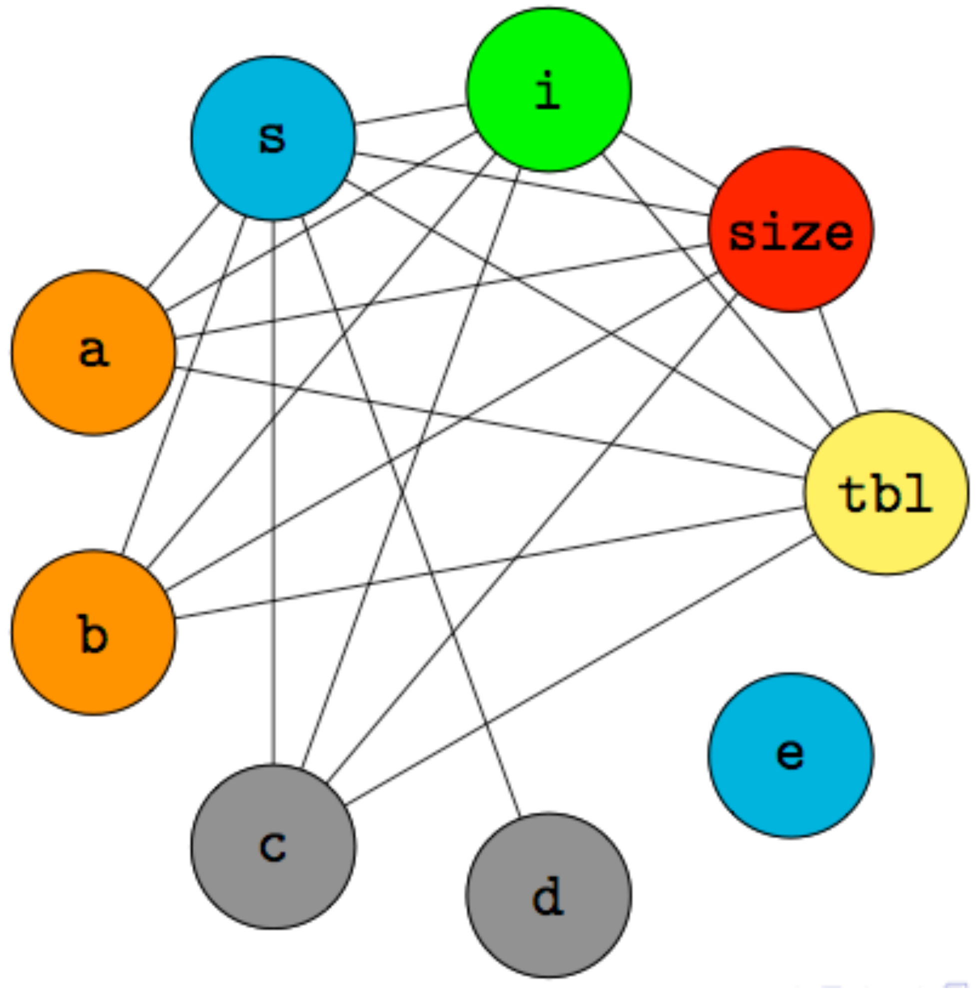




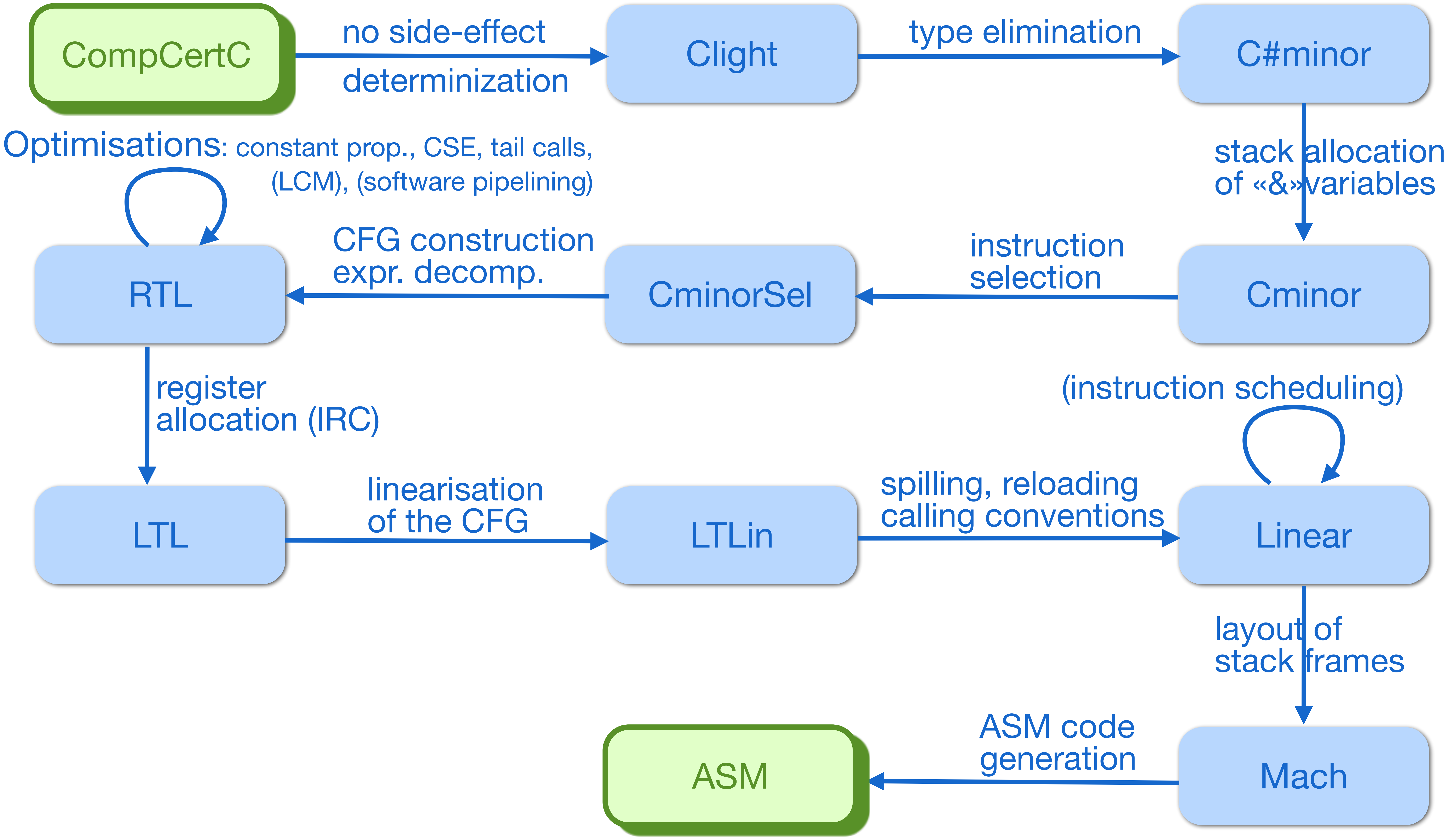
Verified validator

- Less to prove (if validator simpler than transformation)
- Validator reusable for several variants of an optimization
- Can be efficient (cheap enough to be invoked on every compiler run)

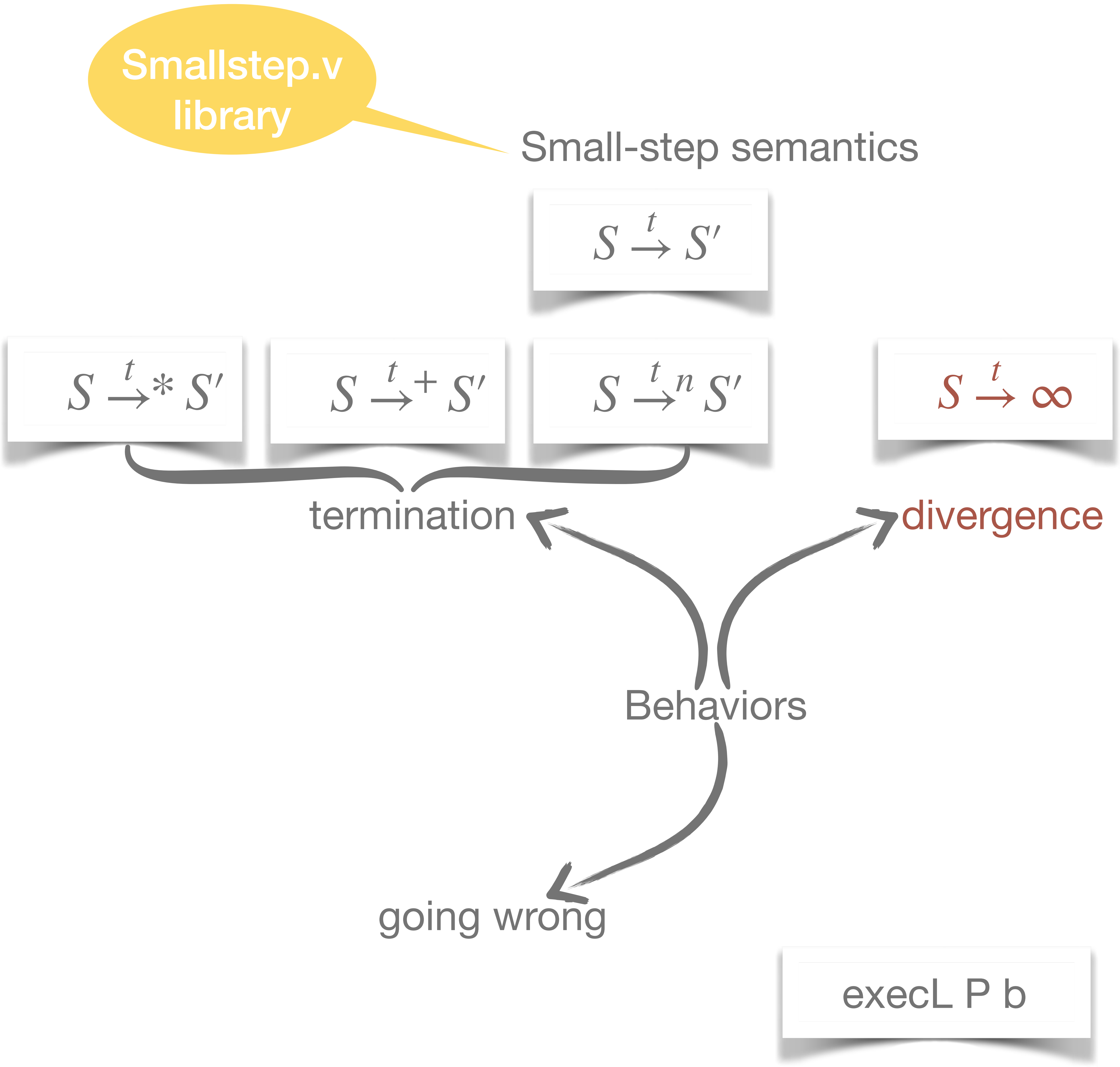
Example: register allocation with advanced spilling and splitting



CompCert compiler: 11 languages, 18 passes



CompCert compiler: 11 languages, 18 passes



Observable behaviors

Behaviors.v and Events.v

```
Inductive program_behavior :=
  | Terminates (t: trace) (n: int)
  | Diverges (t: trace)
  | Reacts (tinf: traceinf)
  | Goes_wrong (t: trace).
```

`trace` = list of I/O events

`traceinf` = infinite list of I/O events

I/O event

- call to an external function (e.g. `printf`)
- memory accesses to global volatile variables (hardware devices)

General form of small-step semantics

Smallstep.v

observed events

$G \vdash S \xrightarrow{t} S'$

does not change during transitions

`initial_state(`S`)`

`final_state(`S`,n)`

return value

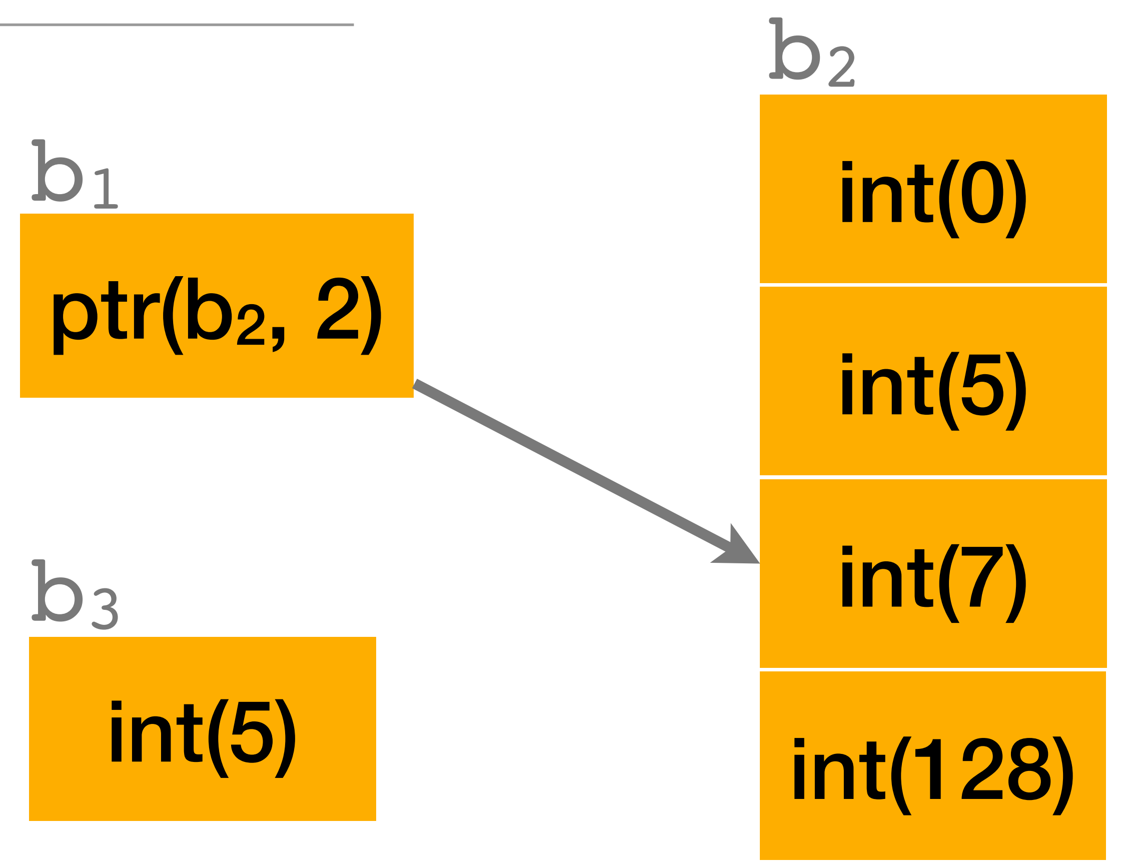
G maps:

- each name of a function or global variable to a memory address
- each function pointer to a function definition

Semantic states S include a memory state, mapping addresses to values

The CompCert memory model

Memory.v 



Shared by all the languages of the compiler

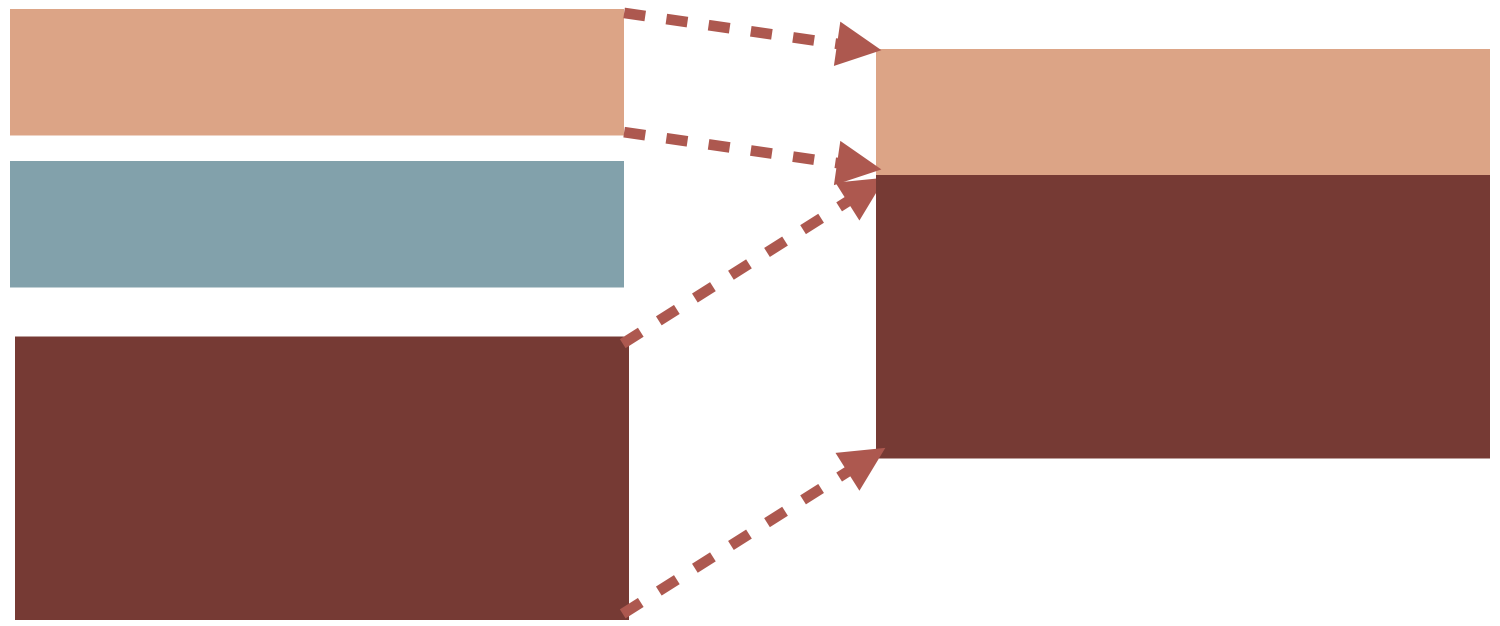
An abstract view of memory refined into a concrete memory layout

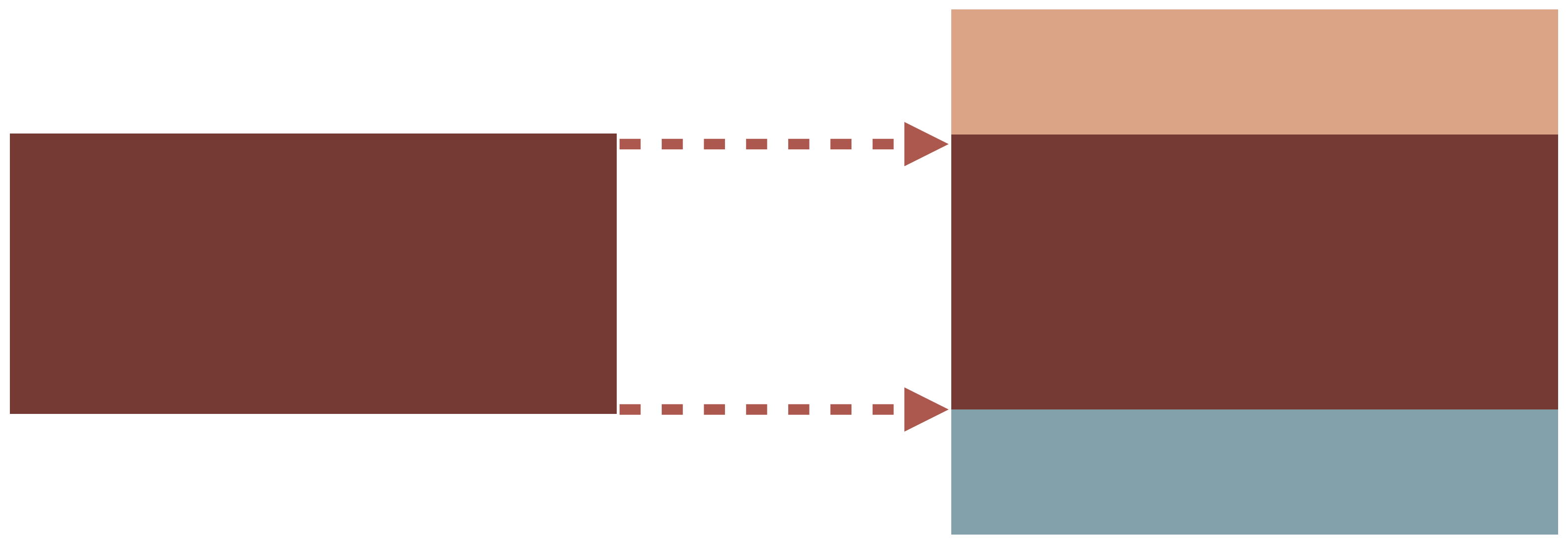
In the semantics: `m: mem`

Memory operations (load, store, alloc, free) over values (machine integers, pointers, floating-point numbers)

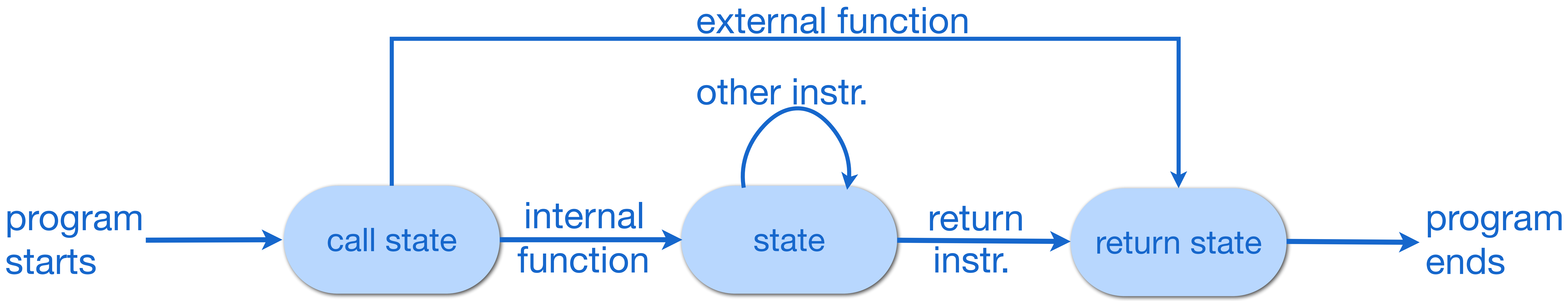
Memory safety preserved by CompCert (good variable properties)

Generic memory injections and memory extensions





Semantic states



Exemple: Clight

```
Inductive state :=
  | State (f: function)(s: statement)(k: cont)(e: env)(le: temp_env)(m: mem)
  | Callstate (fd: fundef)(args: list val)(k: cont)(m: mem)
  | Returnstate (res: val)(k: cont)(m: mem).
```

Exception: assembly languages, where a state is a pair of a memory and a mapping from processor registers to values

CompCert C source language
(see chapter 4 of the user's manual)

Expressions are annotated with their type

```
Eval(int(5), Tint(I32,Signed)): expr
```

Overloading and implicit conversions between types

Expressions have side-effects

- Assignments are expressions

Non-deterministic evaluation of expressions (e.g., see this slide)

Numerous semantic rules in small-step style

Commands

All C constructs: loops, `switch`, `goto`, `break`, `continue`, `return`

Numerous semantic rules in small-step style

Clight language

Clight.v

Expressions are annotated with their type

```
Econst_int(int(5), Tint(I32,Signed)): expr
```

No overloading and explicit conversions between types and arithmetic operators

Expressions are pure

Temporary variables do not reside in memory

```
&v
```

19 semantic rules in big-step style

Commands

Assignments are commands

Single syntax for loops, continue command

```
Sloop s1 s2
```

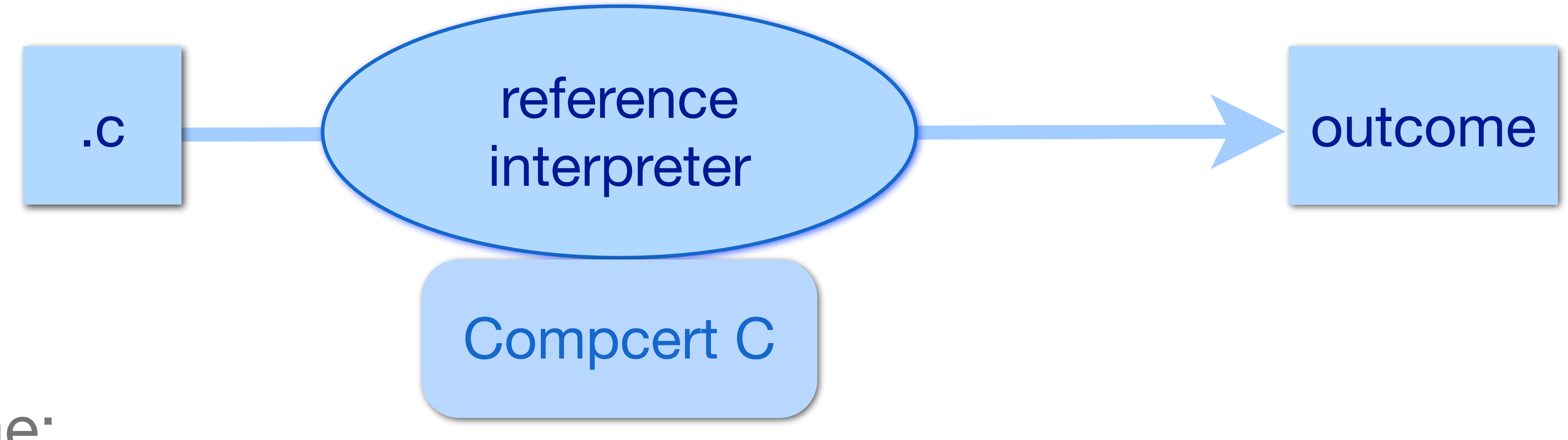
- C loops are derived forms

25 semantic rules in small-step style

+ numerous rules for unary and binary operators, memory loads and stores

The CompCert C reference interpreter

Cexec.v



Outcome:

- normal termination or aborting on an undefined behavior
- observable effects (I/O events: `printf`, `malloc`, `free`)

Faithful to the formal semantics of CompCert C; the interpreter displays all the behaviors according to the semantics

```
step: state → trace → state → Prop
```

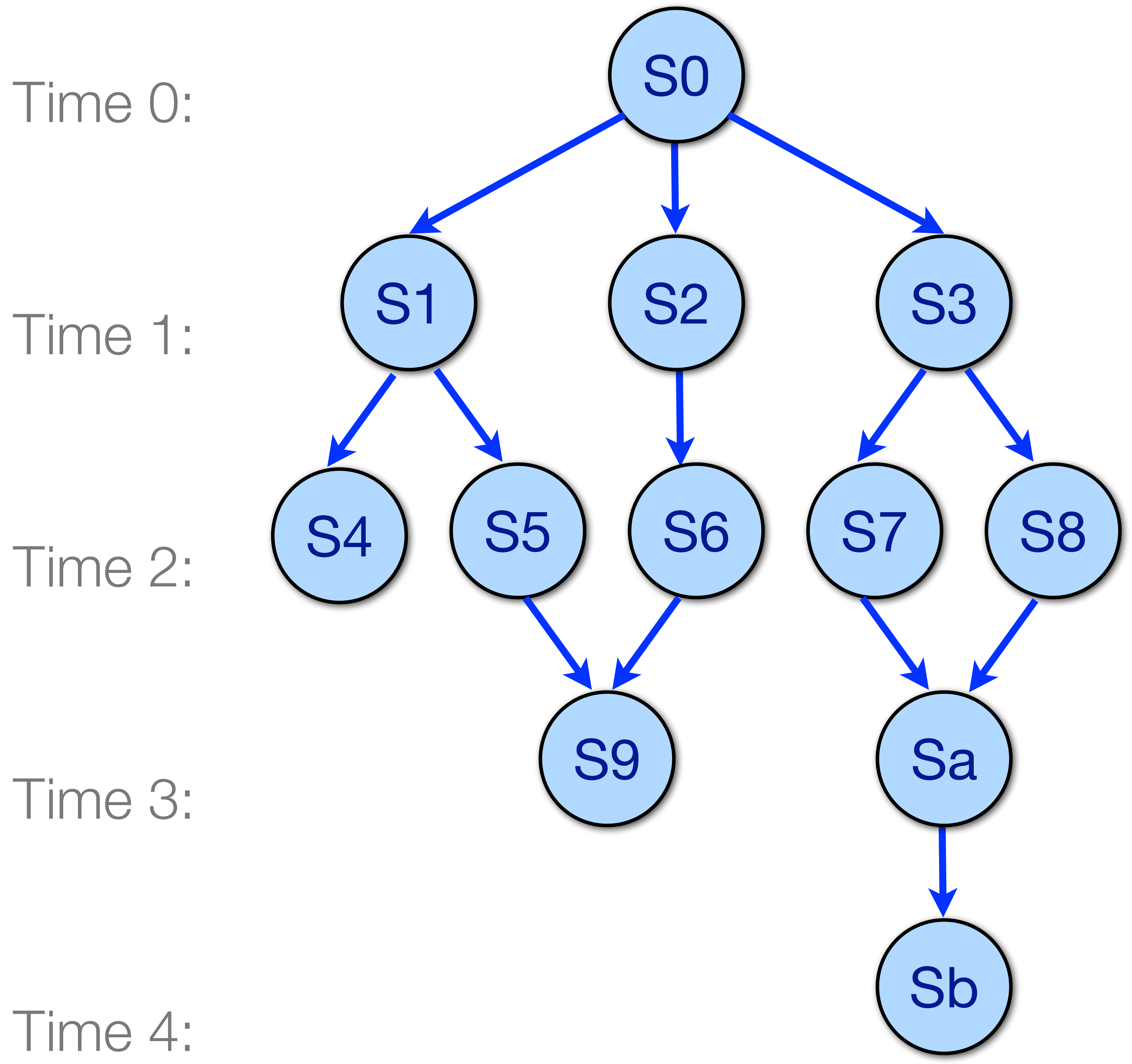
predicate

```
do_step: world → state → list (trace * state)
```

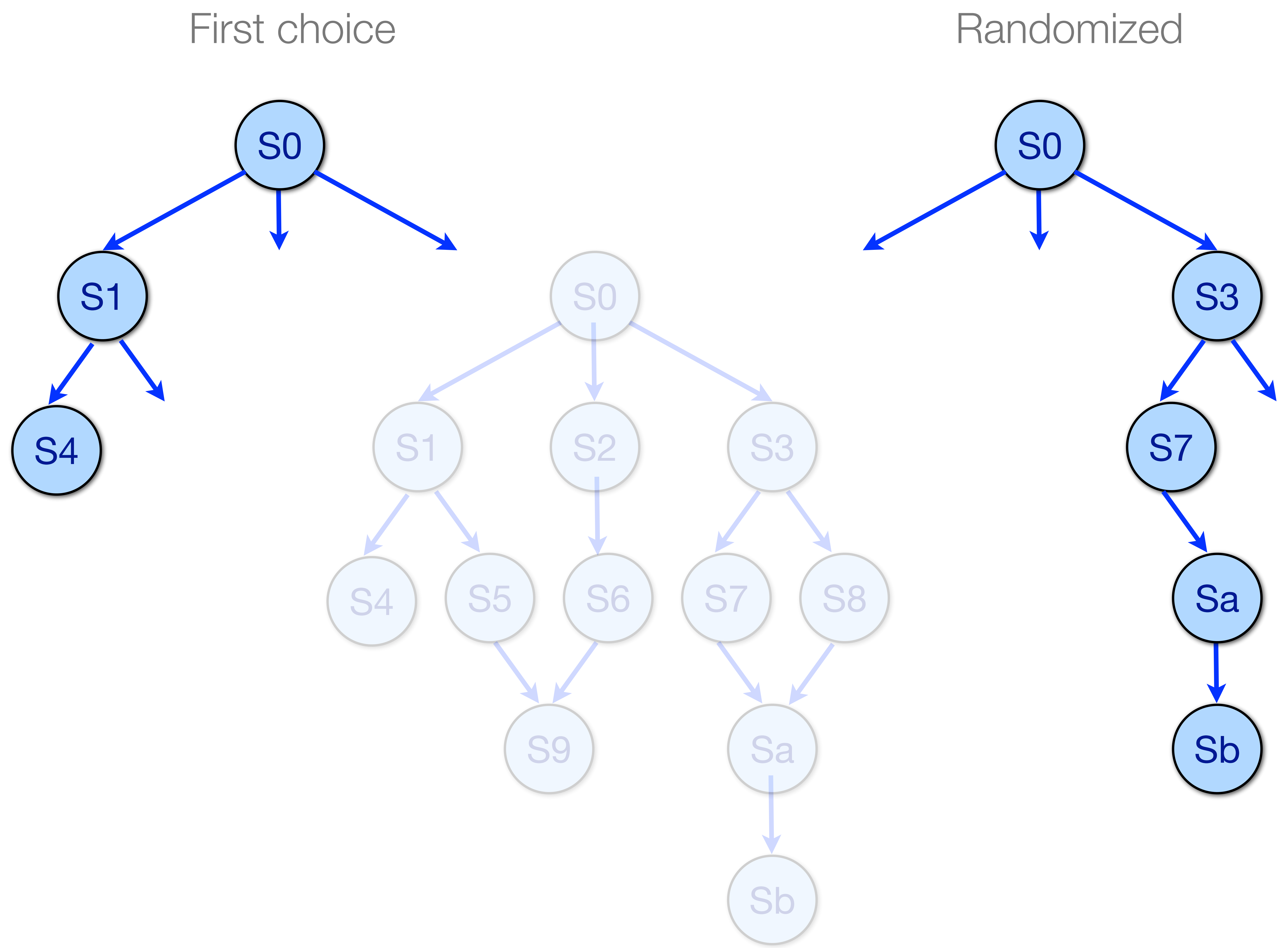
external world: uniquely determines the results of external calls

function

Using the reference interpreter: exhaustive exploration

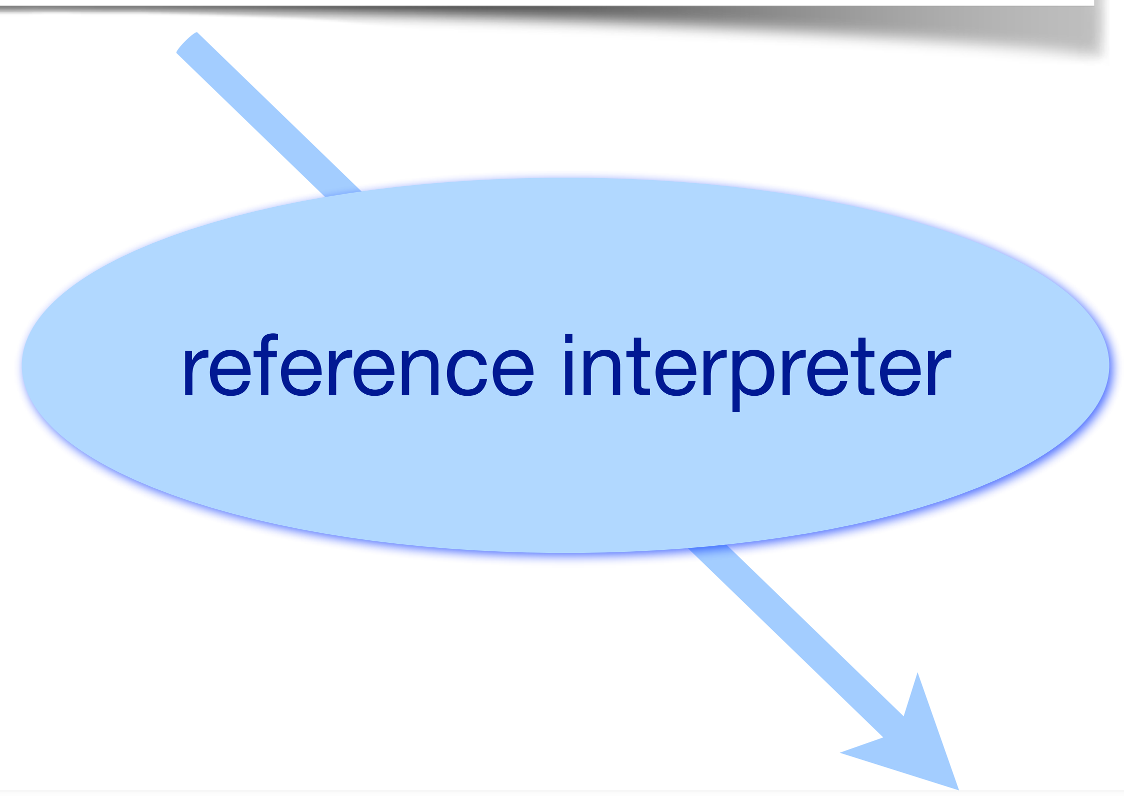


Using the reference interpreter: randomized exploration



Using the reference interpreter
A first example

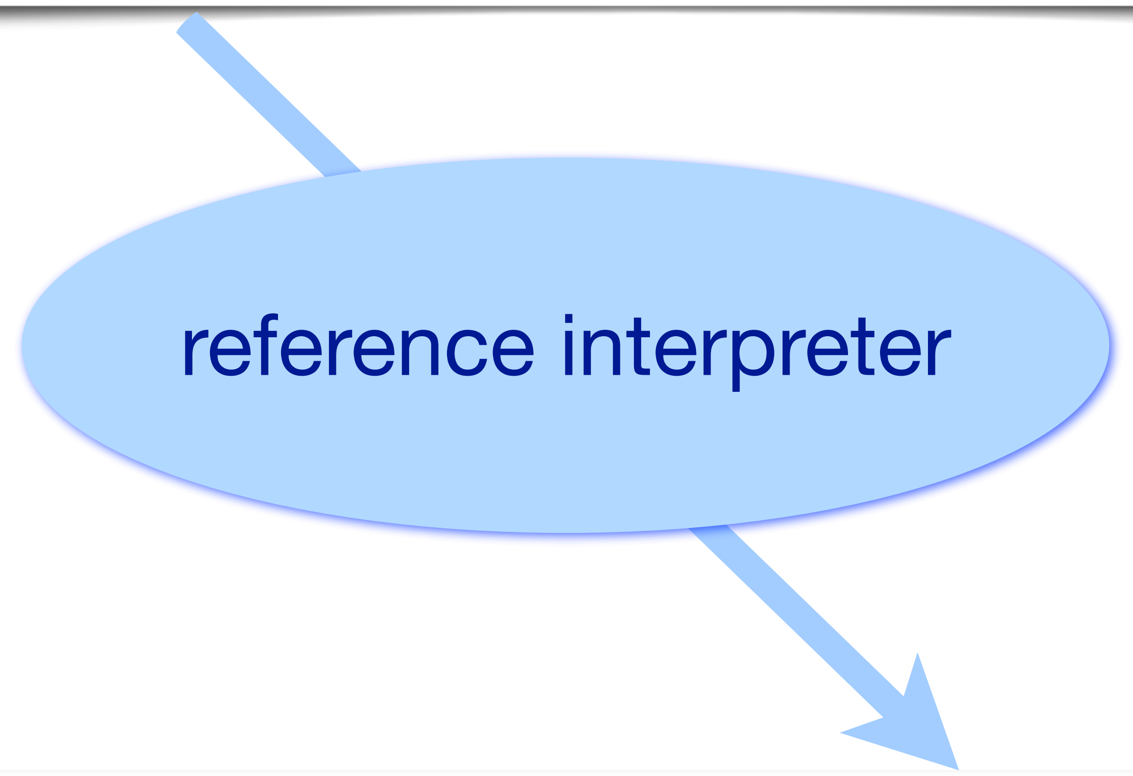
```
int main(void)
{  int x[2] = { 12, 34 };
  printf("x[2] = %d\n", x[2]);
  return 0;  }
```




```
Stuck state: in function main, expression
  <printf>(<ptr __stringlit_1>, <loc x+8>)
Stuck subexpression: <loc x+8>
ERROR: Undefined behavior
```

Using the reference interpreter
A second example: randomized exploration

```
int a() { printf("a "); return 1; }
int b() { printf("b "); return 2; }
int c() { printf("c "); return 3; }

int main () { printf("%d\n", a() + (b() + c())); return 0;  }
```




```
State 45.9:  returning 3
State 45.10: returning 2
State 45.11: returning 1

State 55.1:  returning 0
Time 55: program terminated (exit code = 0)
```

RTL language

RTL.v

Each function is represented by its CFG

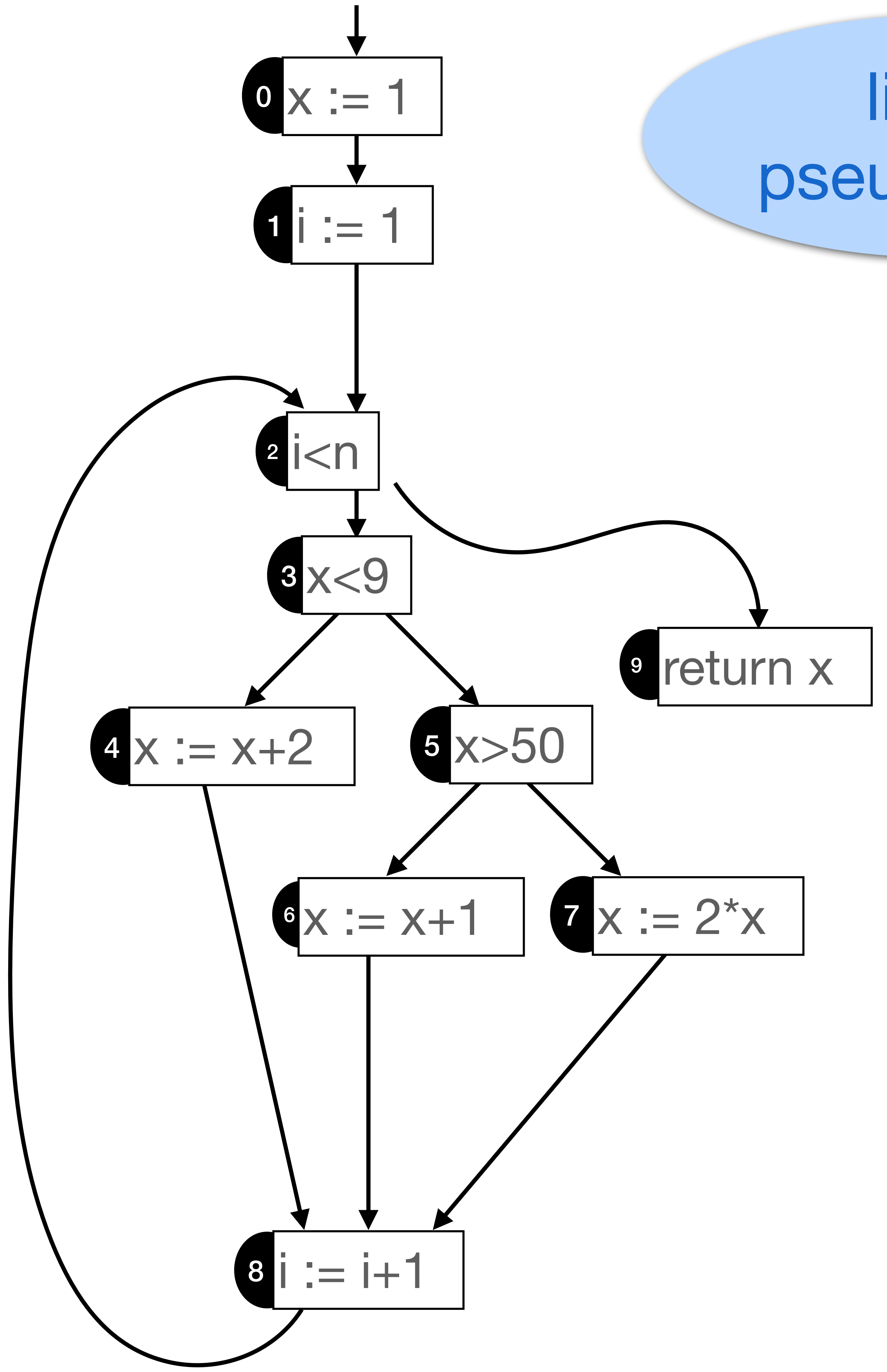
Instructions only

Unlimited supply of pseudo-registers

```
Iop(int(5), args, dest, succ): instruction
```

- args: list of pseudo-regs
- dest: register to store the result
- succ: successor node

```
int f(int n) {
 int x = 1;
 for (int i = 1; i < n; i++)
   if (x < 9) x = x + 2;
   else if (x > 50) x = x + 1;
   else x = 2 * x;
 return x;  }
```



Part 6: summary

Proving a compiler pass mainly amounts to proving a simulation diagram

Many reusable libraries:

- simulations, memory model, C semantics, Clight and RTL languages
- machine integers, dataflow solver

Some compilation options

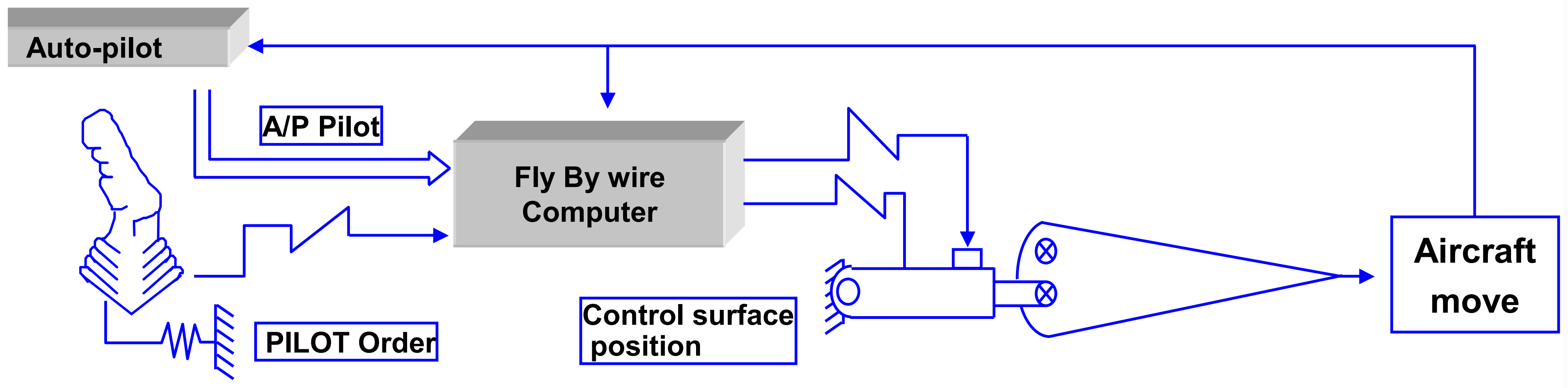
- using the CompCert C interpreter: `-interp` (`-trace`, `-all`, `-random`)
- tracing options: `-dc`, `-dclight`, `-drtl, …`
- show the time spent in compiler passes: `-timing`

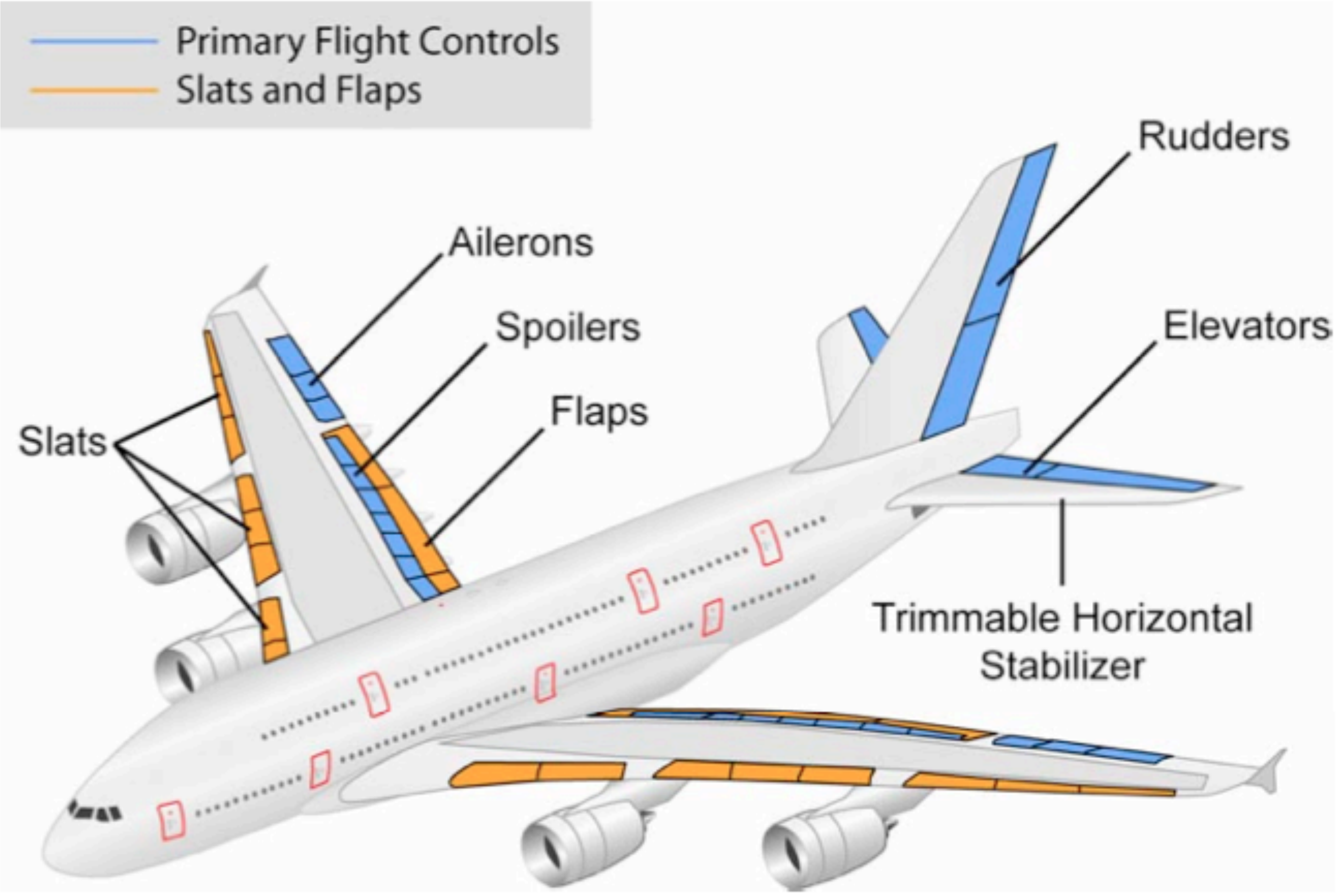
Part 7: Compiling critical embedded software with CompCert

Fly-by-wire software

Execute pilot's commands

Flight assistance: keep aircraft within safe flight envelope





Mostly control-command code (Scade) +
a minimalistic OS (C)

100k - 1M LOC code, but mostly generated from block diagrams (Simulink, Scade)

Fly-by-wire software

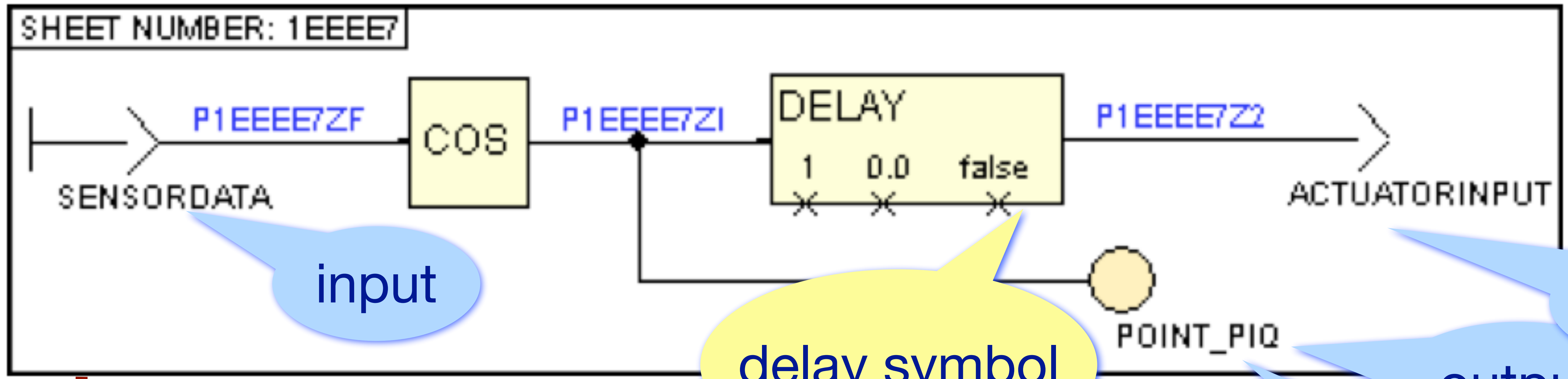
The qualification process (DO-178)

Rigorous validation: review (qualitative), analysis (quantitative), testing (huge amounts)

Conducted at multiple levels, from design to final product

Meticulous development process; extensive documentation

From block diagrams to assembly




```
/*Sheet Number: 1EEEE7*/

#include "other_includes.h"
#include "delay.mac"
#include "cos.mac"
#include "piq_x.mac"

m_START_NUM_INPUT_ZONE
m_INPUT(SENSORDATA,NUM)
m_END_NUM_INPUT_ZONE

m_START_NUM_OUTPUT_ZONE
m_OUTPUT(ACTUATORINPUT,NUM)
m_END_NUM_OUTPUT_ZONE

m_START_NUM_OBS_ZONE
m_OBS_NUM(POINT_PIQ)
m_END_NUM_OBS_ZONE

m_START_CST_INPUT_ZONE
m_CST(c_DELAY_P3_2_N1EEEE7,ENT, 1)
m_END_CST_INPUT_ZONE

m_START_STATIC_ARRAY_ZONE
m_STATIC_ARRAY(t_DELAY_2_N1EEEE7, 1,NUM)
m_END_STATIC_ARRAY_ZONE

m_START_SHEET(1E,EE,E7)

m_CONNECTION(P1EEEE7Z2_N1EEEE7,NUM)
m_CONNECTION(loc_c_DELAY_P3_2_N1EEEE7,ENT)
m_CONNECTION(P1EEEE7ZF_N1EEEE7,NUM)
m_CONNECTION(P1EEEE7ZI_N1EEEE7,NUM)

m_VtS(SENSORDATA,P1EEEE7ZF_N1EEEE7)
m_CtS(c_DELAY_P3_2_N1EEEE7,loc_c_DELAY_P3_2_N1EEEE7)

COS(0_N1EEEE7,P1EEEE7ZF_N1EEEE7,P1EEEE7ZI_N1EEEE7)
PIQ_X(1_N1EEEE7,P1EEEE7ZI_N1EEEE7,POINT_PIQ)
DELAY(2_N1EEEE7,P1EEEE7ZI_N1EEEE7,\
loc_c_DELAY_P3_2_N1EEEE7,P1EEEE7Z2_N1EEEE7,t_DELAY_2_N1EEEE7)

m_StV(P1EEEE7Z2_N1EEEE7,ACTUATORINPUT)
m_END_SHEET
```

```
    Nm_1EEEE7:
    ; annotation: Symbol DELAY number 2_N1EEEE7 ,\
 inputs: f3, r31 and one static
100 addis      r12, 0, (_DELAY_2_N1EEEE7_R2)@ha
104 lwz        r4, (_DELAY_2_N1EEEE7_R2)@l(r12)
    ; annotation: Variable to search: loc_c_DELAY_P3_2_N1EEEE7
    ; annotation: DELAY; is entered with r4 = from 0 to
108 mr         r7, r4
10c addis      r12, 0, (t_DELAY_2_N1EEEE7)@ha
110 addi       r8, r12, (t_DELAY_2_N1EEEE7)@l
114 rlwinm     r10, r7, 3, 0, 28 ; 0xfffffff8
118 add        r10, r8, r10
11c lfd        f2, 0(r10)
    ; annotation: Variable to search: loc_c_DELAY_P3_2_N1EEEE7
    ; annotation: DELAY; is entered with r4 = from 0 to
120 mr         r8, r4
124 addis      r12, 0, (t_DELAY_2_N1EEEE7)@ha
128 addi       r6, r12, (t_DELAY_2_N1EEEE7)@l
12c rlwinm     r5, r8, 3, 0, 28 ; 0xfffffff8
130 add        r9, r6, r5
134 stfd       f3, 0(r9)
138 addi       r4, r4, 1
13c cmpw       cr0, r4, r31
140 bt         0, .L101
144 addi       r4, 0, 0
    .L101:
148 addis      r12, 0, (_DELAY_2_N1EEEE7_R2)@ha
14c stw r4, (_DELAY_2_N1EEEE7_R2)@l(r12)
    ; annotation: End of DELAY number 2_N1EEEE7 , output: f2
```

Program annotations

A mechanism to attach annotations to program points

- Mark specific program points
- Provide information about the location of C variables
- Ensure that some variables are preserved (e.g. x must be kept in a register)

Annotations are preserved during compilation.

- Each annotation generates an observable event
- The correctness theorem ensures preservation of the sequencing of 1) symbols, and 2) of accesses to hardware devices (volatile variables)

```
_annot("Begin of a loop");
...
x = 1;
_annot("Here x is at %1",x);
...
_annot("End of a loop");
```

compiler

```
; annotation: Begin of a loop
...
addi r3, 0, 1
; annotation: Here x is at r3
...
; annotation: End of a loop
```

Conformance to the qualification process

A formally verified compiler gives traceability guarantees.

Simplified example

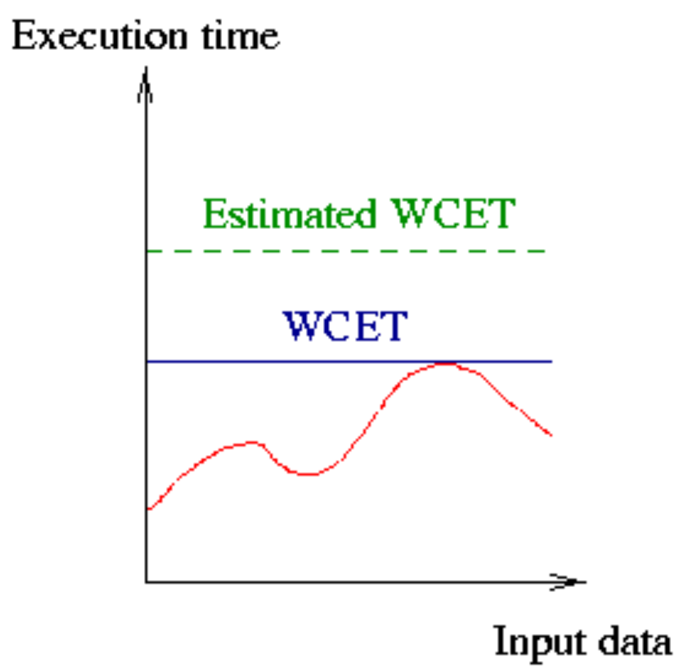
- The semantics preservation theorem ensures preservation of:
  - the sequencing of symbols,
  - the sequencing of accesses to hardware devices (volatile variables).

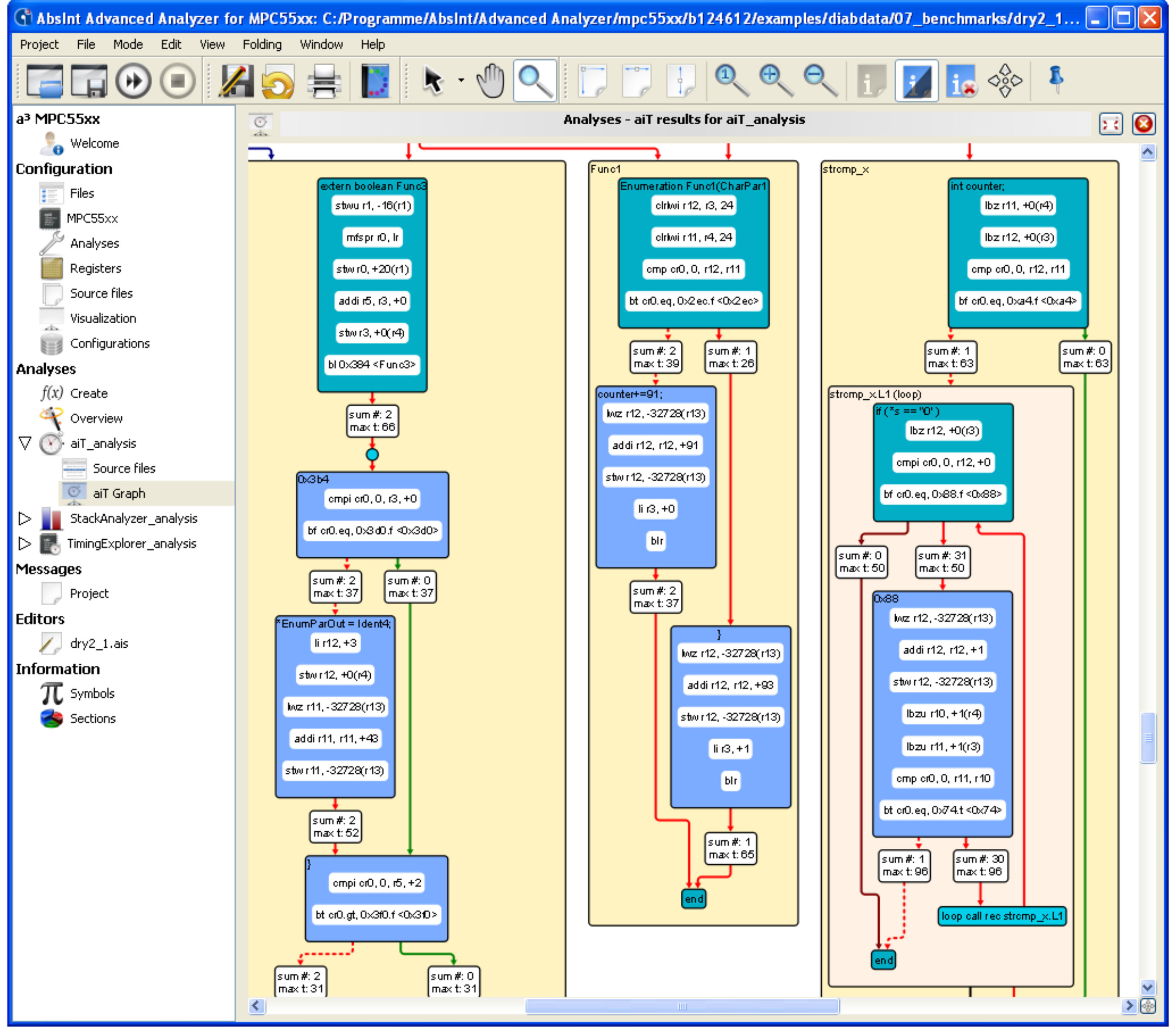
Remember the main theorem: If the source program can not go wrong, then the behavior of the generated assembly code is exactly one of the behaviors of the source program.

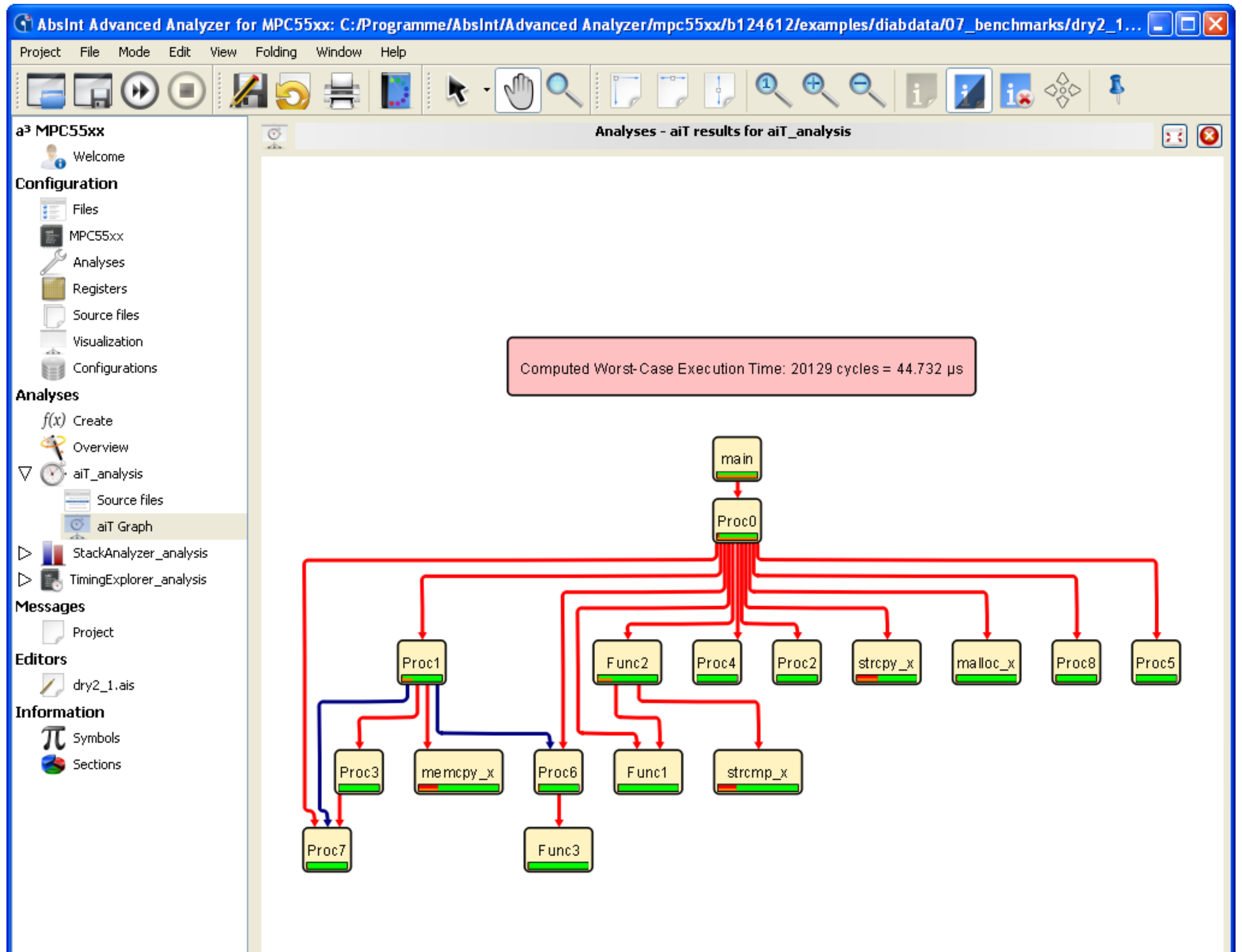
How good is the compiled code ?

Trade-off between
• traceability guarantees
• and efficiency of the generated code

Low-level verifications
• reviews of the assembly
• computation of a WCET estimation





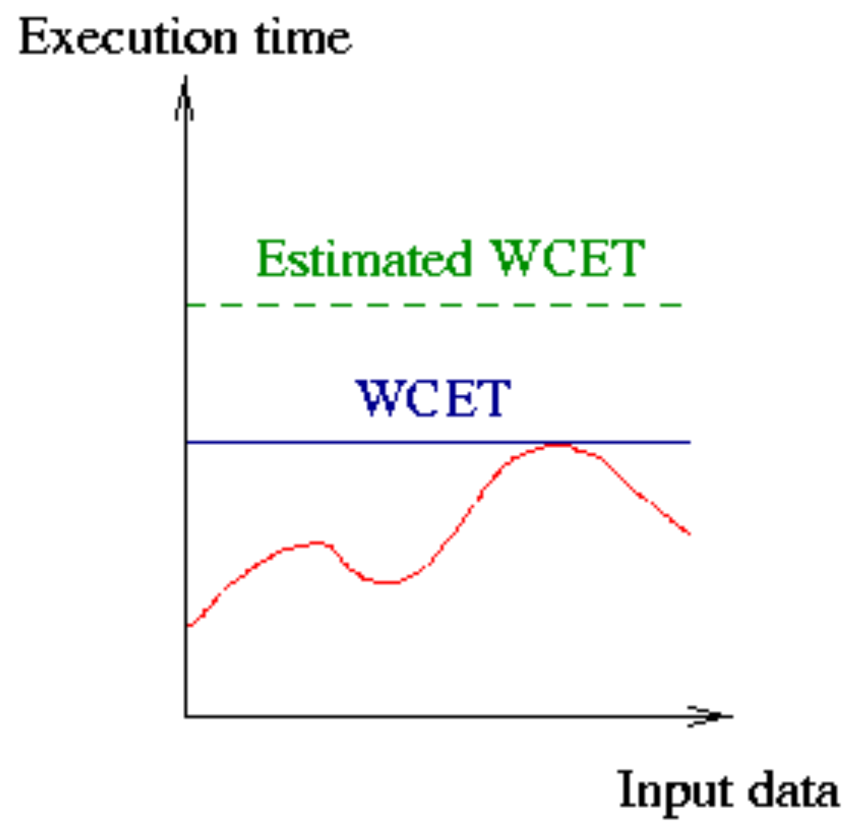


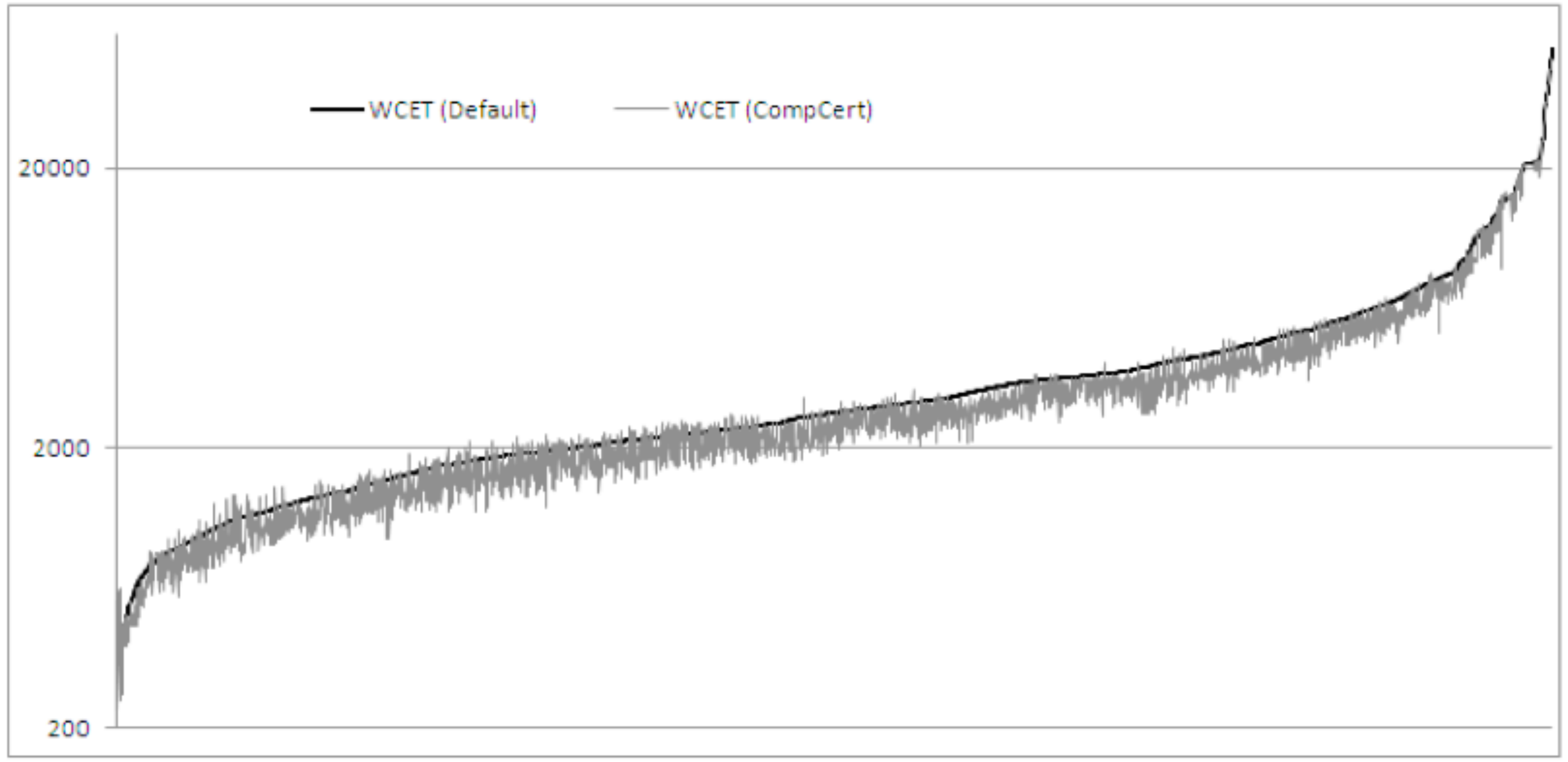
Compiling critical embedded software

Improved performances of the generated code,
while providing proven traceability guarantees thanks to annotations

FCGU A380: 3600 files, 3.96 MB of assembly code

- Estimated WCET for each file
- Average improvement per file: 13,5%
- Compiled with CompCert 1.10, 2012





Overall assessment

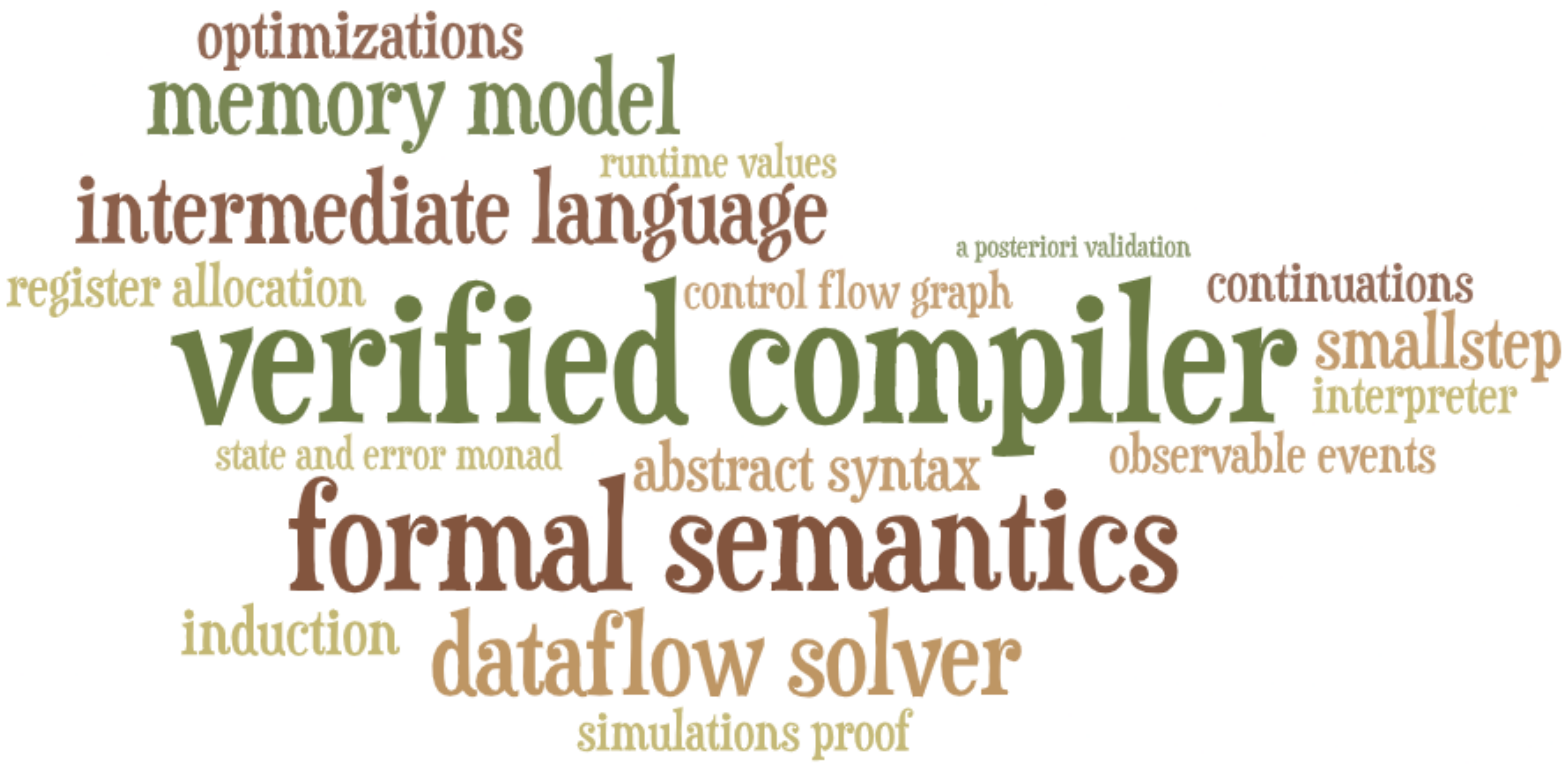
The improvement mainly comes from the register allocation pass.

- From: no register allocation
- To: sharing of local variables among available registers

Traceability guarantees

- From: tracking of all program variables
- To: tracking of meaningful variables (used in block diagrams)

Part 8:
CompCert, a shared infrastructure for ongoing research



The Verasco abstract interpreter

[Jourdan, Laporte, Blazy, Leory, Pichardie, POPL’15] [Blazy, Laporte, Pichardie, ICFP’16]

A holistic effect with compiler verification

CompCert compiler

```
Theorem csharpminor_compiler_correct_alt:
  ∀ p tp b,
  transf_c_program p = OK tp →
  execC p b →
  execASM tp b.
```

forward simulation

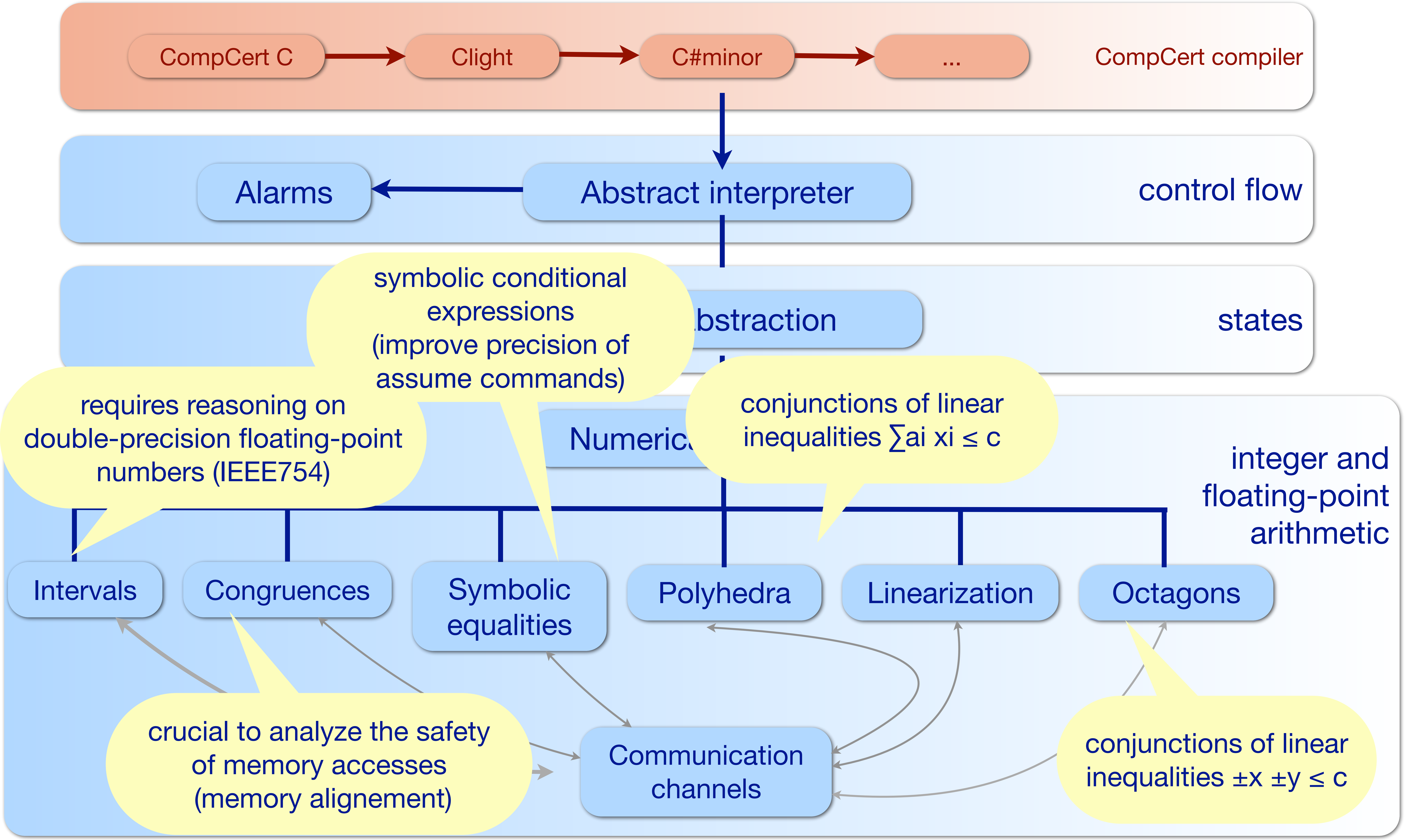
Verasco abstract interpreter

```
Theorem analyzer_is_correct:
  ∀ p b,
  static_analyzer p = Success →
  execC p b.
```

p can not go wrong

```
Theorem csharp_compiler_correct_stronger:
  ∀ p tp b,
  transf_c_program p = OK tp →
  execASM tp b.
```

Verasco architecture



Turning CompCert into a secure compiler

CT-CompCert [Barthe, Blazy, Grégoire, Hutin, Laporte, Pichardie, Trieu, POPL'20]

Cryptographic constant-time (CCT) programming discipline

```
unsigned nok-function (unsigned x, unsigned y, bool secret)
{ if (secret) return y; else return x; }
```

```
unsigned ok-function (unsigned x, unsigned y, bool secret)
{ return x ^ ((y ^ x) & (-(unsigned)secret)); }
```

How to turn CompCert into a formally-verified secure compiler?

```
Theorem compiler-correct:
  ∀ S C b,
  compiler S = OK C →
  execCompCertC S b →
  execASM C b.
```

```
Theorem compiler-preserves-CCT:
  ∀ S C,
  compiler S = OK C →
  isCCT S →
  isCCT C.
```

Which proof technique for the isCCT policy?

Observational non-interference: observing program leakage (boolean guards and memory accesses) during execution does not reveal any information about secrets

```
Theorem compiler-preserves-CCT:
  ∀ S C,
  compiler S = OK C →
  isCCT S →
  isCCT C.
```

isCCT S

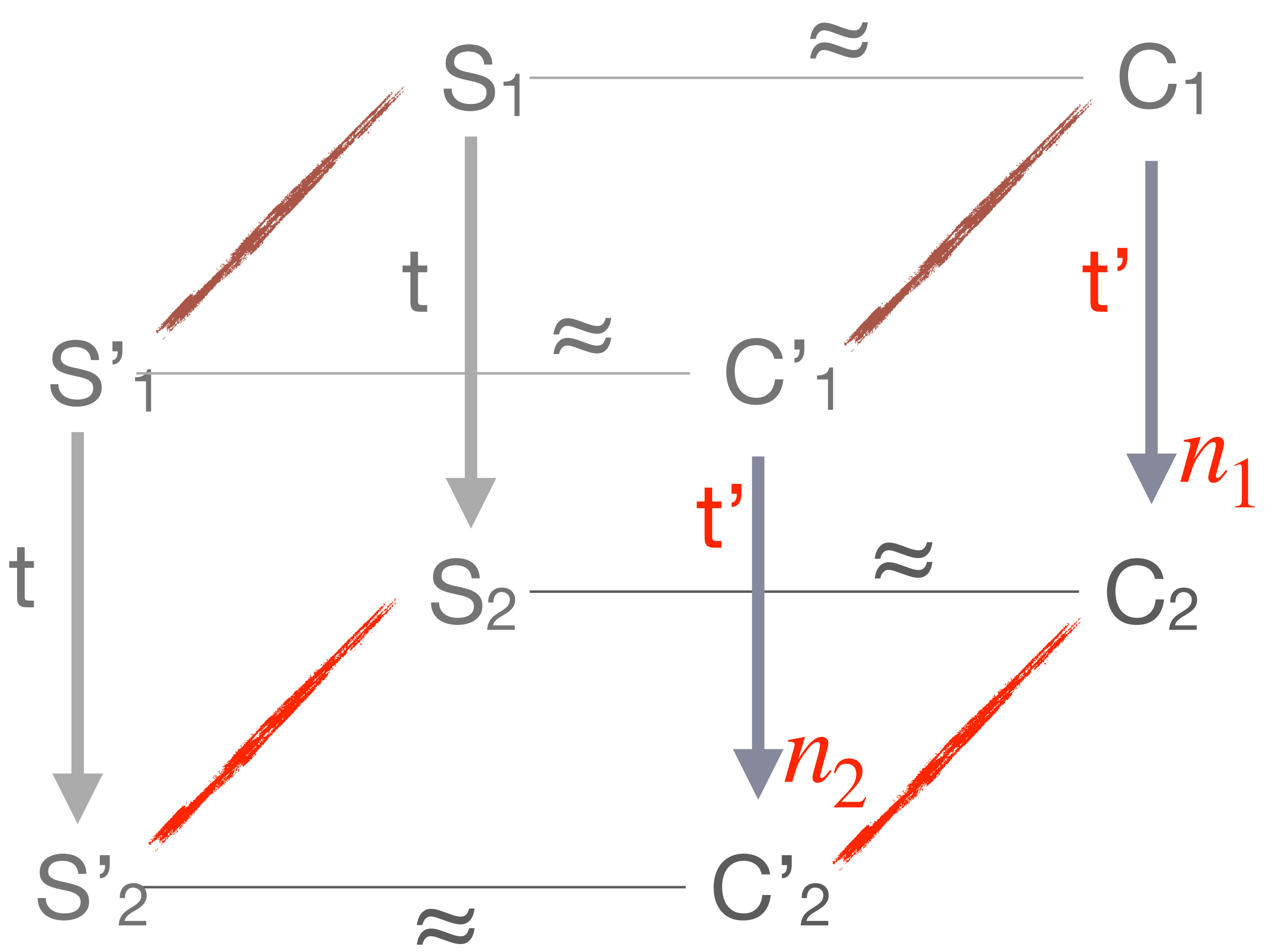
$S_1 \xrightarrow{\ell} S_2$

$S'_1 \xrightarrow{\ell'} S'_2$

with $\varphi(S_1, S'_1)$

implies $\ell = \ell'$

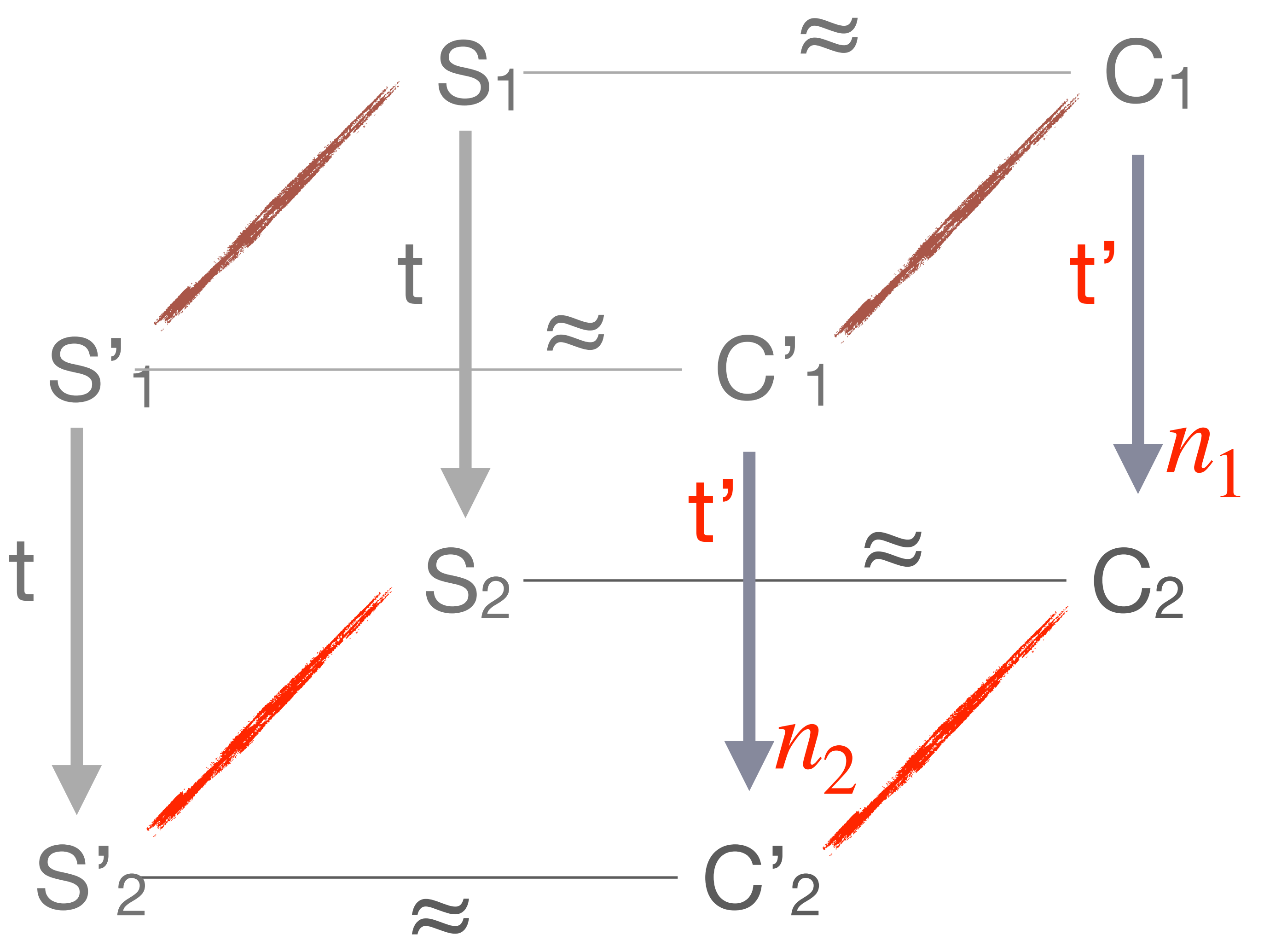
Indistinguishability property $\varphi(S_i, S'_i)$: share public values, but may differ on secret values

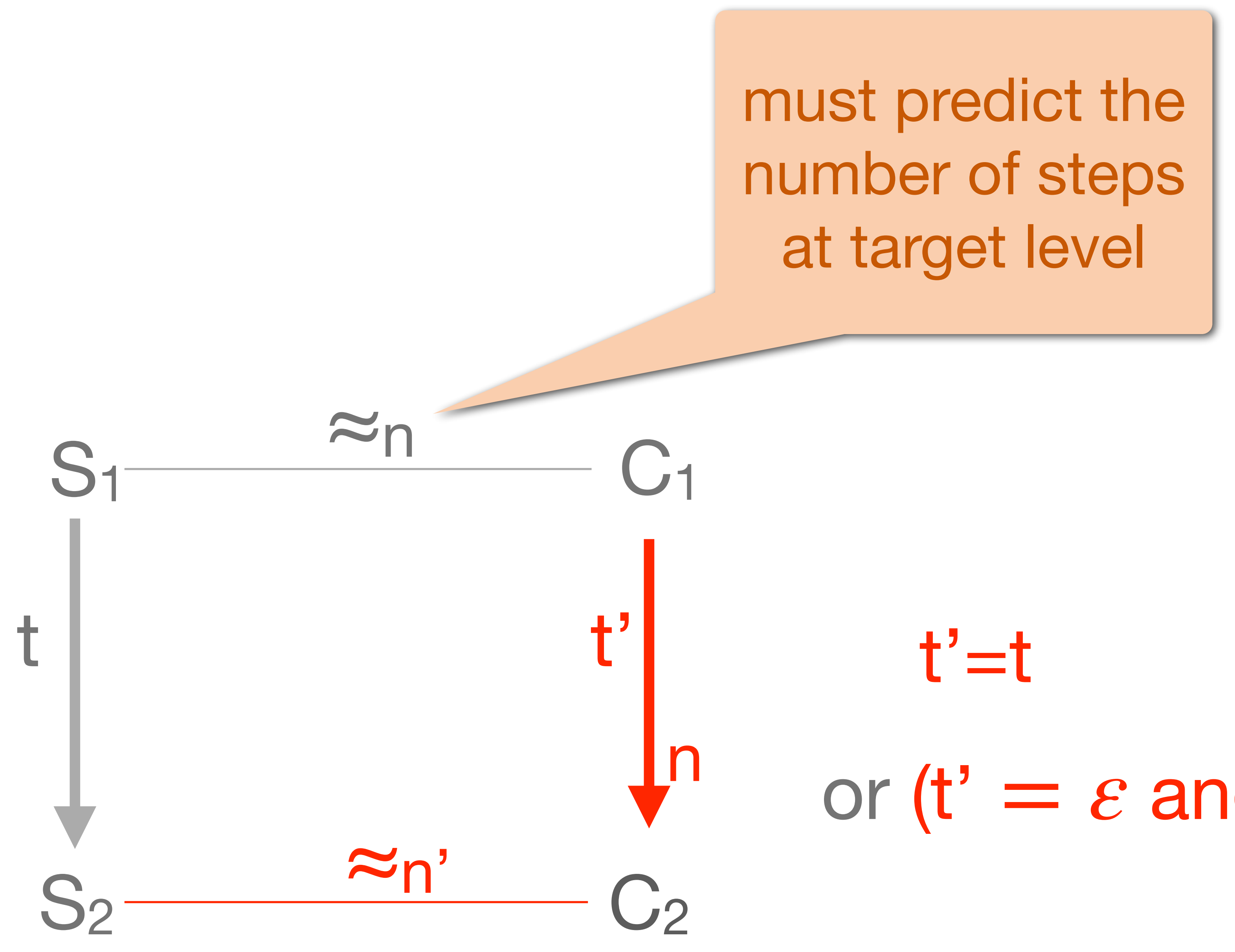


Difficulty: tricky proofs!

Proving CCT preservation: back to simulation diagrams

Proof-engineering: leverage the **existing proof scripts** as much as possible



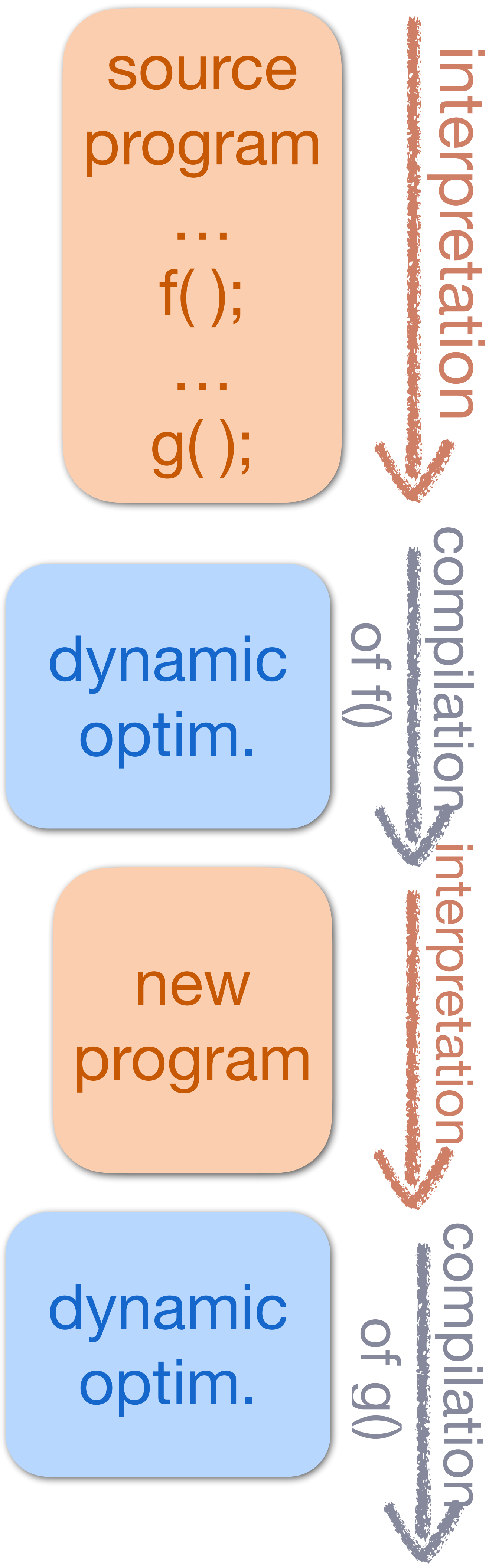


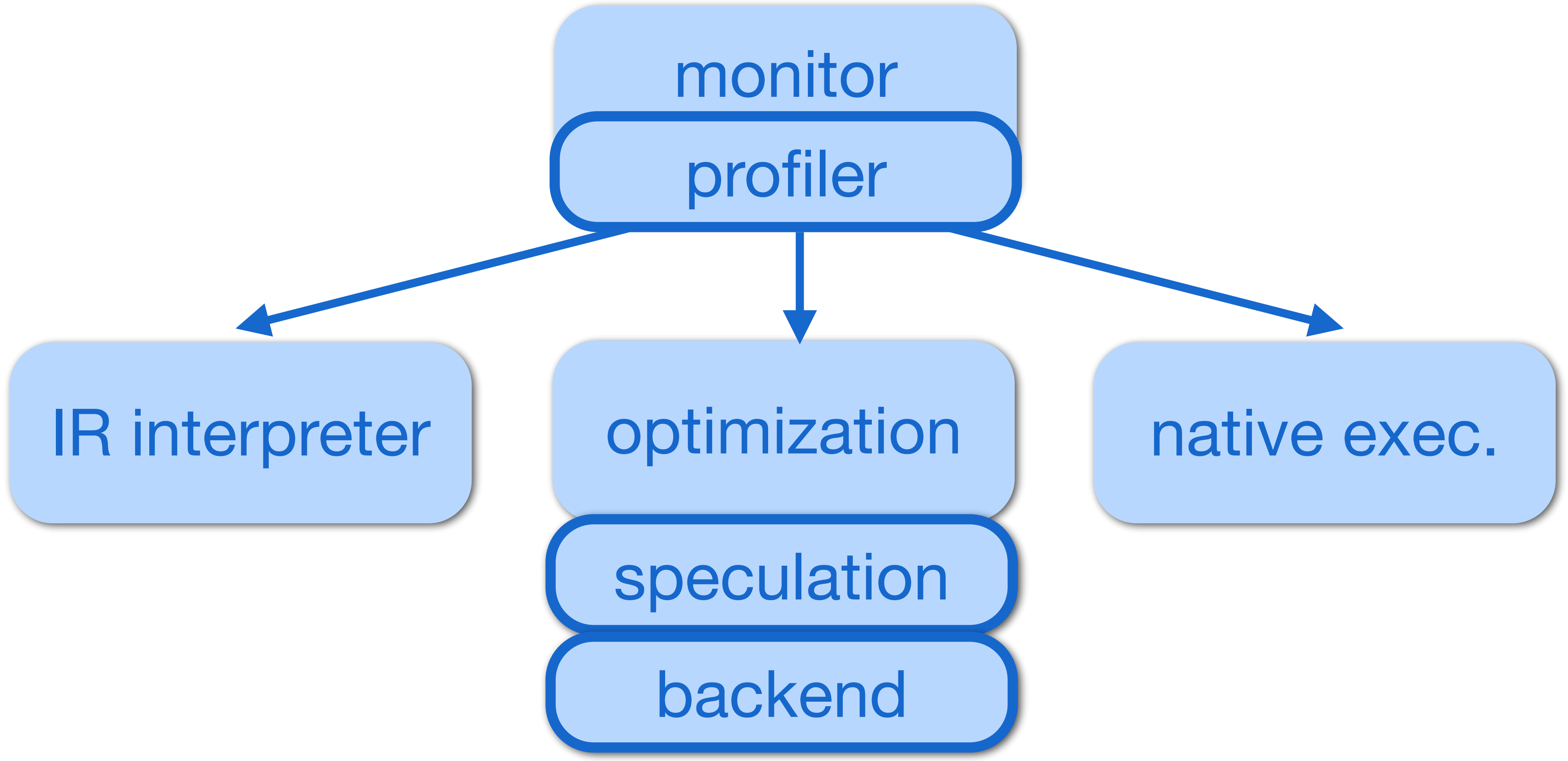
t'=t

or (t' $= \varepsilon$ and t is leak only)

Verifying just-in-time (JIT) compilation [Barrière's PhD 12/2022]

[Barrière, Blazy, Flückiger, Pichardie, Vitek, POPL'21] and [Barrière, Blazy, Pichardie, POPL'23]



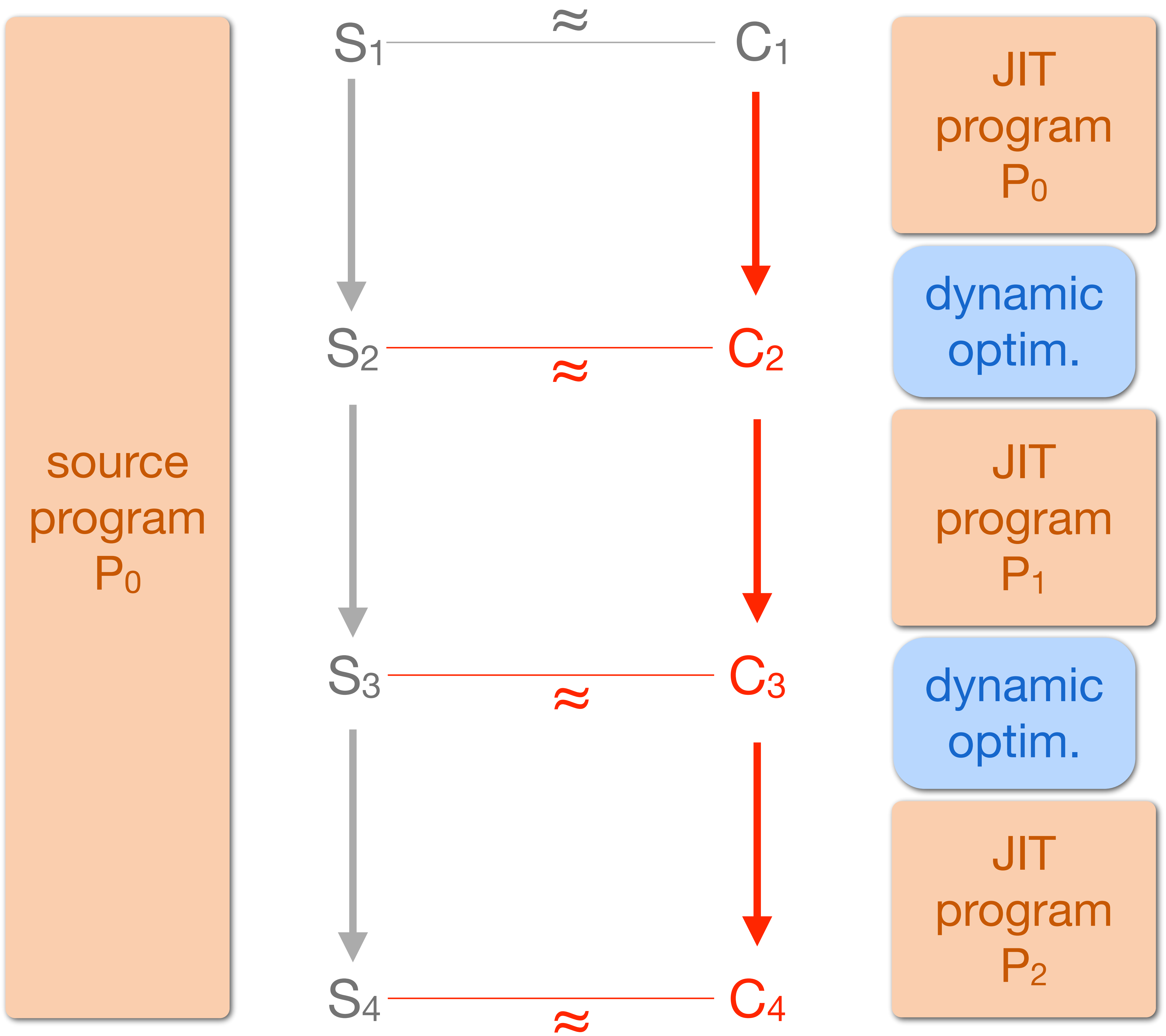


A JIT compiler **interleaves** the execution of a program with its optimizations

Dynamic speculation: specializes functions, requires deoptimization

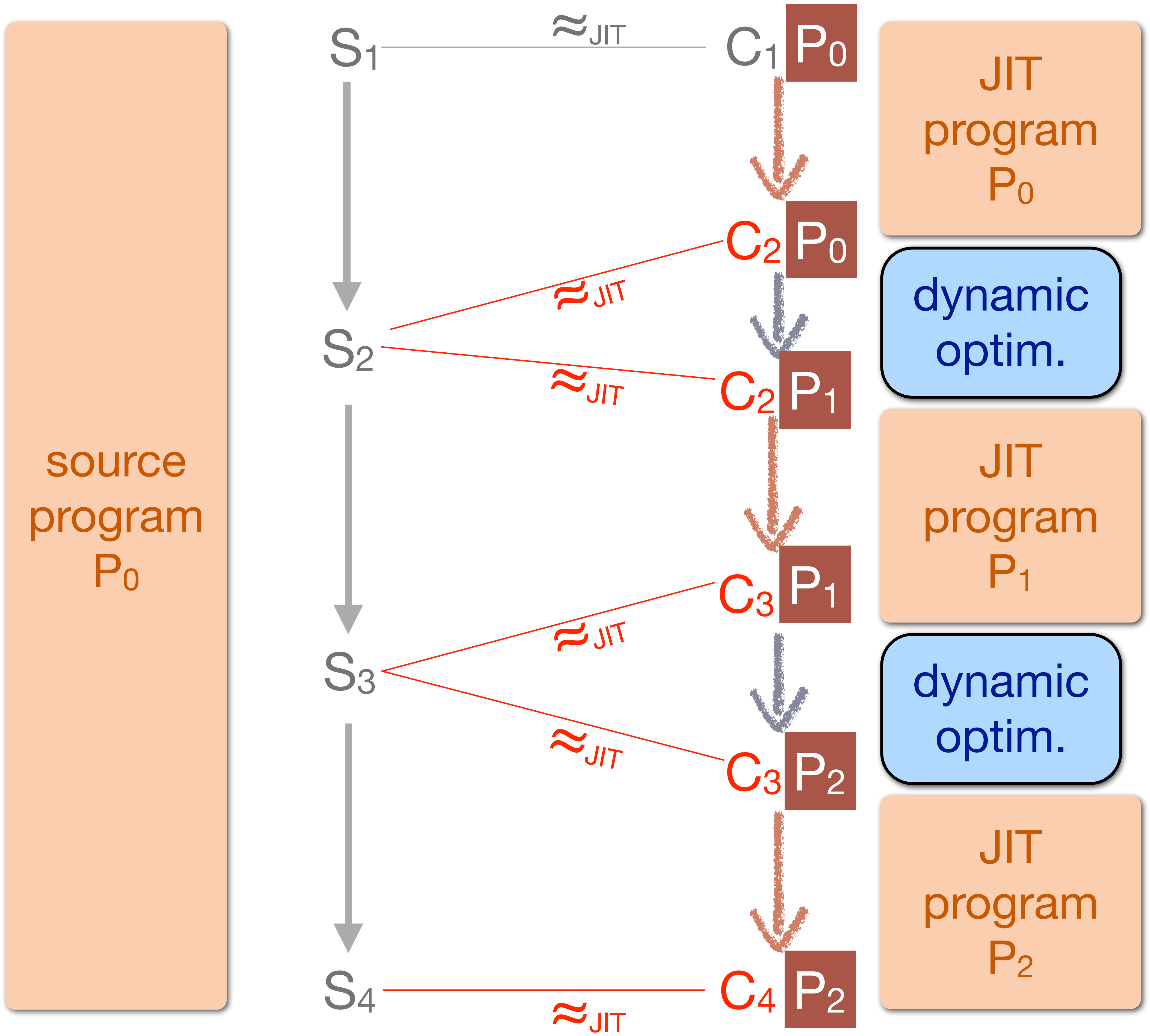
Non-deterministic semantics: either deoptimize to the source program or continue to the next instruction in the optimized program

Proving semantics preservation: the simulation approach



Both the program and the execution state are evolving

Nested simulations for JIT verification



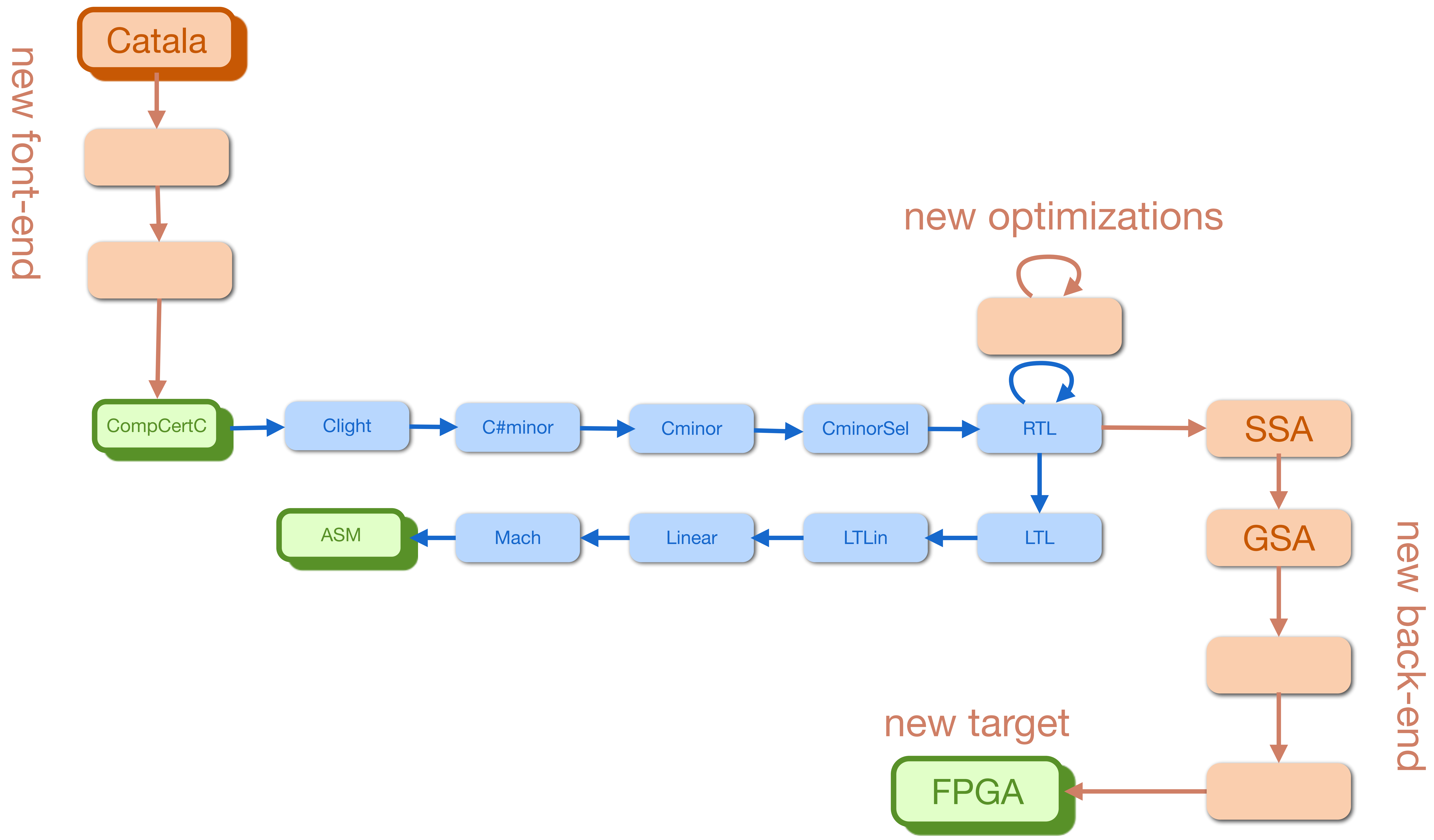
Both the program and the execution state are evolving

Invariant $\approx_{JIT}$: at any point during JIT execution

- the current state C_i corresponds to a source state S_i
- the curent JIT program P_i is equivalent to the source program P_0

Nested simulation: this equivalence is expressed with another simulation

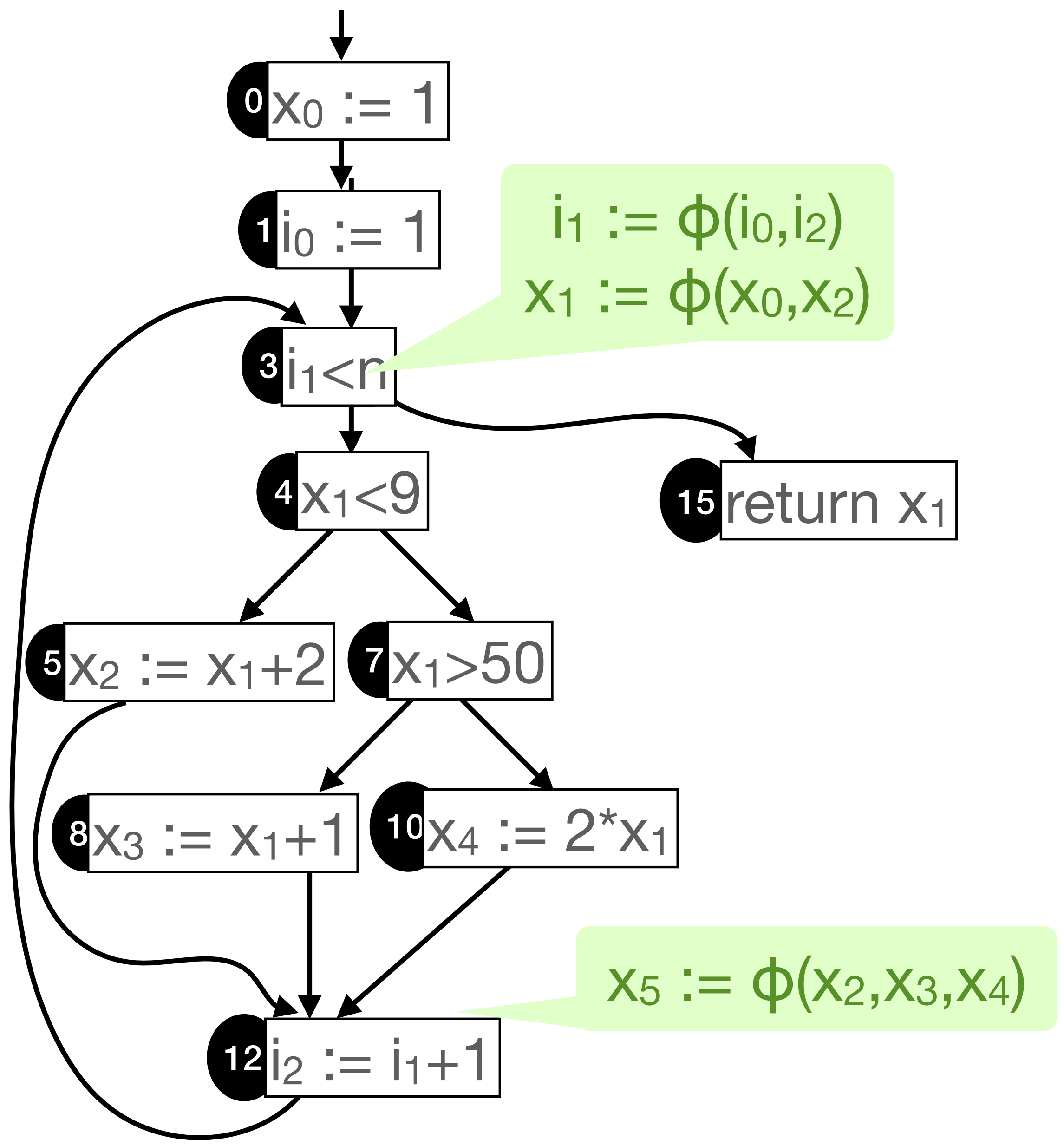
Work in progress 🏗



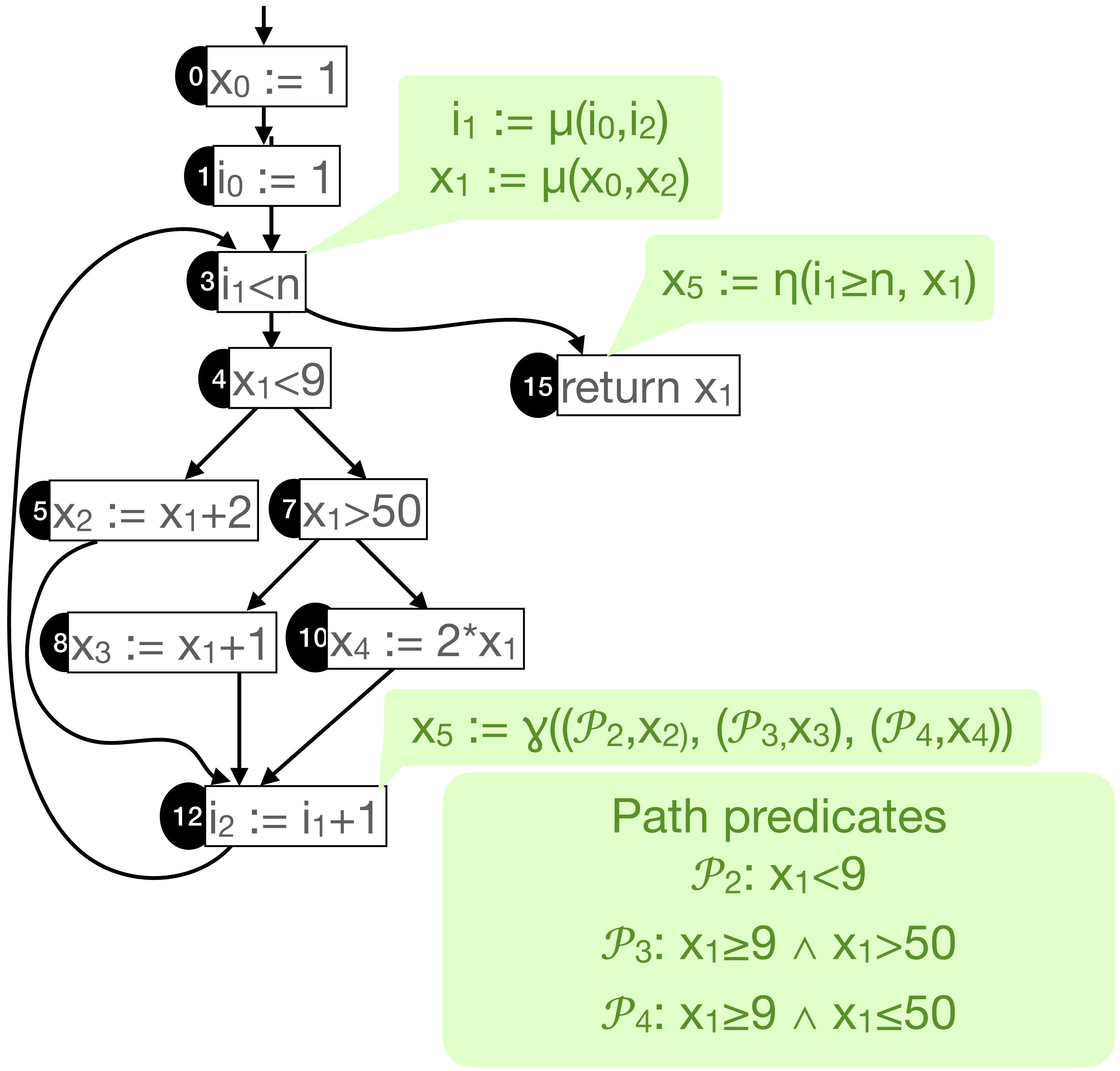
Gated SSA (static single assignment) intermediate representation

```
int f(int n) {
 int x = 1;
 for (int i = 1; i < n; i++)
   if (x < 9) x = x + 2;
   else if (x > 50) x = x + 1;
   else x = 2 * x;
 return x;
}
```

C program



Program in SSA form



Program in GSA form

Conclusion and perspectives

CompCert is a shared infrastructure for ongoing research

- **compilation** : ProbCompCert (Boston College, USA), L2C (Tsinghua, China), Velus (DIENS, Fr), CompCertO (Yale, USA), VeriCert (Imperial College, GB), CompCert-KVX (Verimag, Fr)
- **program logics**: VST (Princeton, USA), Gillian (Imperial College, GB), VeriFast (KUL, Be)
- **static analysis** : Verasco (Inria, Fr)

Opens the way to the trust of development tools

From early intuitions to fundamental formalisms …
verification tools that automate these ideas …
actual use in the critical software industry

Questions?

Bibliography

- Boyer, R. S., Moore, J S. MJRTY - A Fast Majority Vote Algorithm. *Essays in Honor of Woody Bledsoe*. 1991.
- Yang, Chen, Eide, Regehr. Finding and understanding bugs in C compilers. PLDI'11.
- Leroy. Formal verification of a realistic compiler. Communications of the ACM 52(7), 2009.
- Leroy. A formally verified compiler back-end. JAR 43(4), 2009.
- Appel, Blazy. Separation logic for small-step Cminor. TPHOLs'07.
- Blazy, Leroy. Mechanized semantics for the Clight subset of the C language. JAR 43(3), 2009.
- Leroy, Appel, Blazy, Stewart. The CompCert memory model. 2014. Program Logics for Certified Compilers.
- Leroy, Blazy, Kästner, Schommer, Pister, Ferdinand. CompCert - A formally verified optimizing compiler. ERTS2'16.
- Kumar, Myreen, Norrish, Owens. CakeML: a verified implementation of ML. POPL'14.
- Jourdan, Laporte, Blazy, Leroy, Pichardie. A formally-verified static analyzer. POPL'15.
- Blazy, Laporte, Pichardie. An Abstract Memory Functor for Verified C Static Analyzers. ICFP'16.
- Barthe, Blazy, Grégoire, Hutin, Laporte, Pichardie, Trieu. Formal verification of a constant-time preserving C compiler. POPL'20.
- Barrière, Blazy, Flückiger, Pichardie, Vitek. Formally verified speculation and deoptimization in a JIT compiler. POPL'21.
- Barrière, Blazy, Pichardie. Formally verified native code generation in an effectful JIT - or: Turning the CompCert backend into a formally verified JIT compiler. POPL'23.
- Barthe, Demange, Pichardie. Formal Verification of an SSA-based middle-end for CompCert. TOPLAS'14.
- Herklotz, Demange, Blazy. Mechanised semantics for gated static single assignment. CPP'23.
- Merigoux, Chataing, Protzenko. Catala: a programming language for the law. Proc. ICFP'21.