How to provide proof that software is bug-free? Verified compilation to the rescue

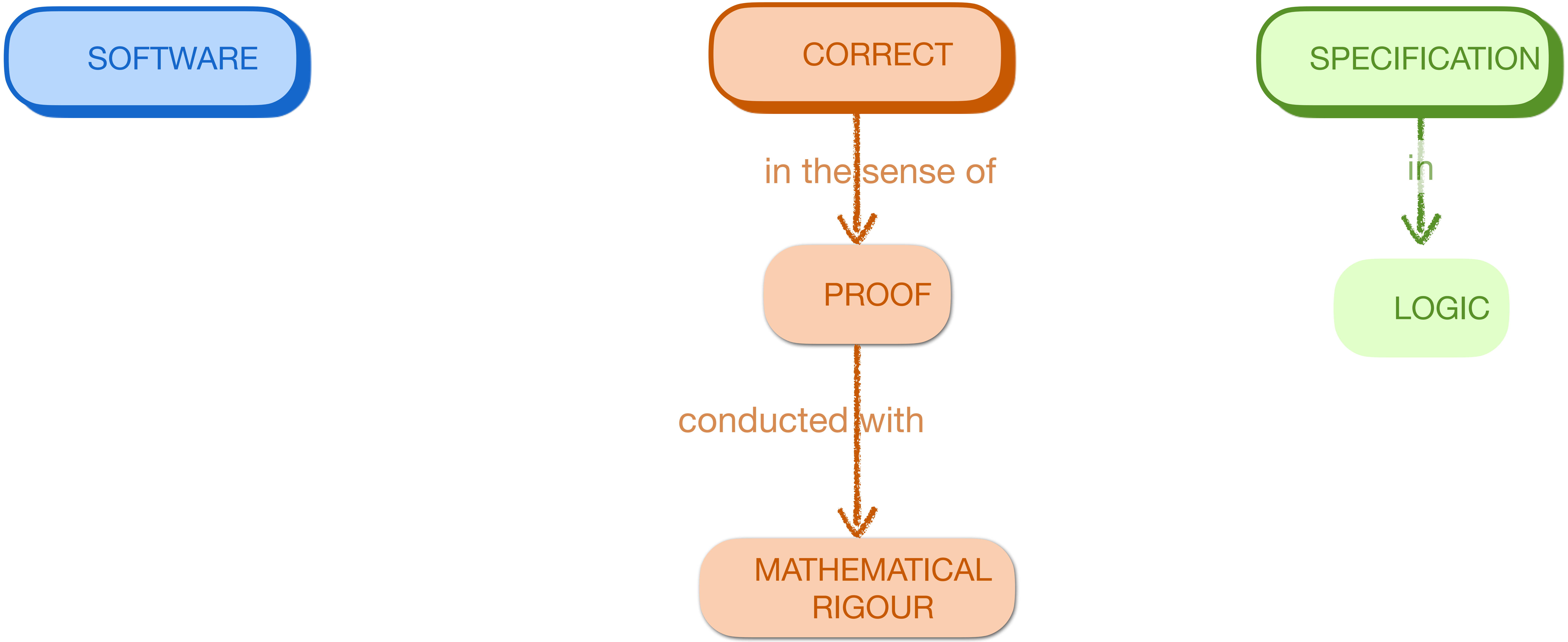
Sandrine Blazy

Université de Rennes

UMR IRISA

GDR GPL, Rennes, 2023-06-07

Deductive verification



From early intuitions …

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

Friday, 24th June.

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

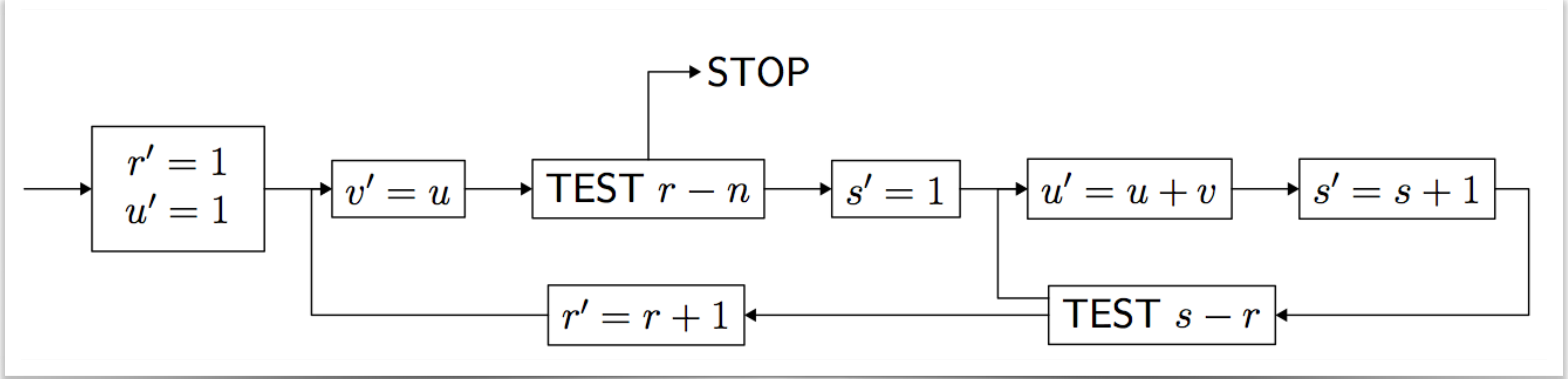
How can one check a routine in the sense of making sure that it is right?

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

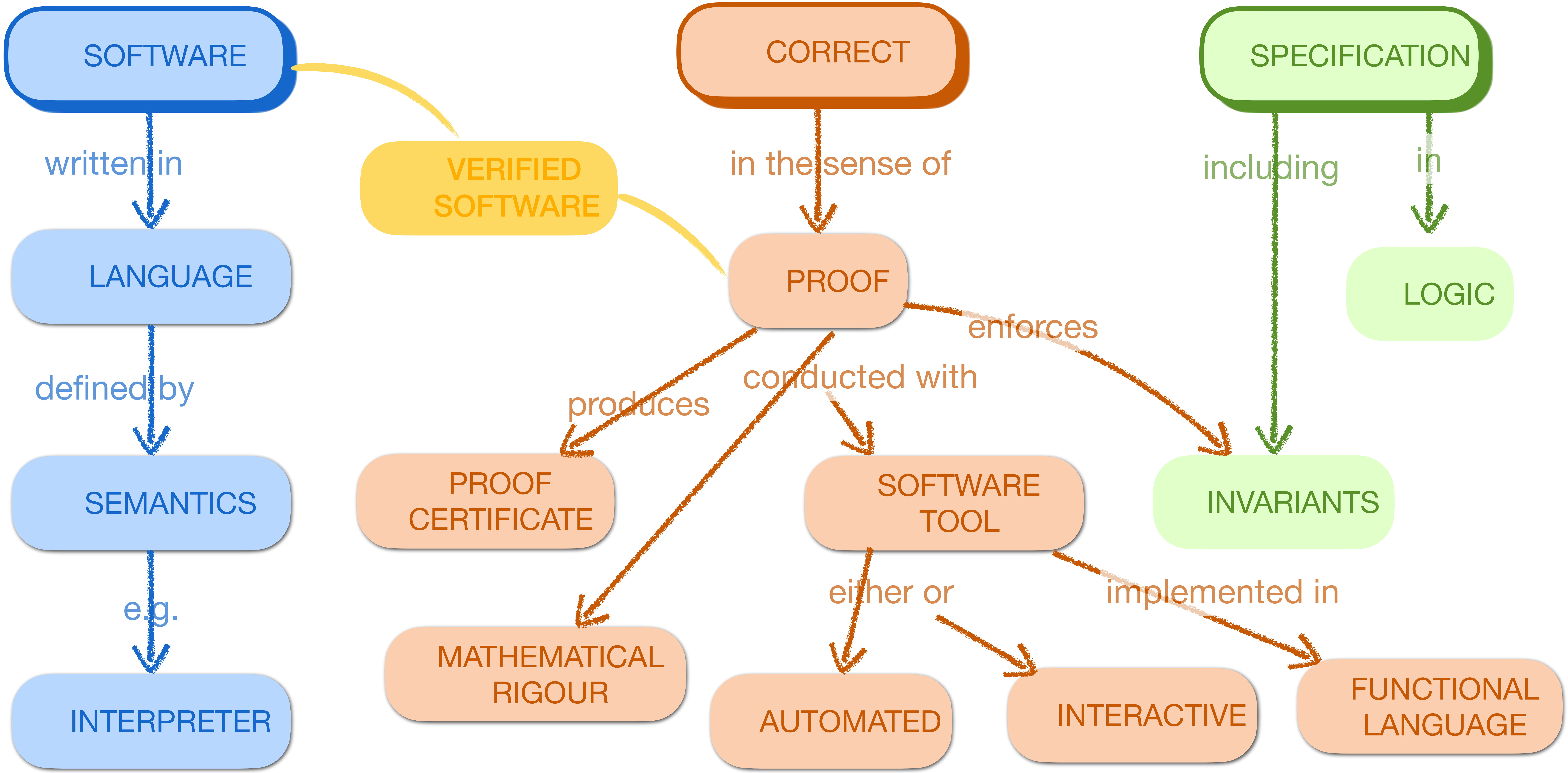
one must check the whole at one sitting, because of the carries.




```
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

A	A	A	C	C	B	B	C	C	C	B	C	C

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

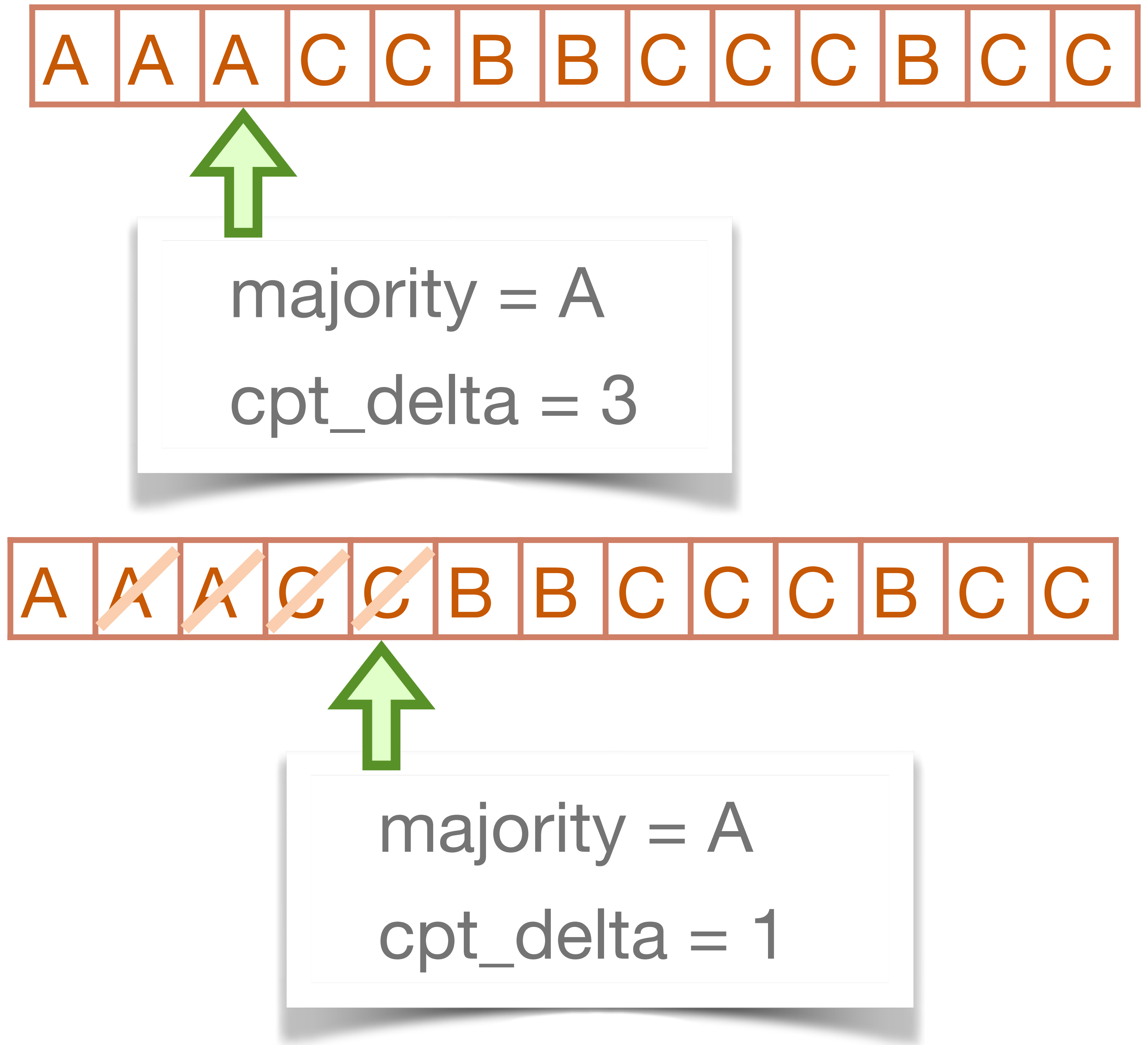
Abstract

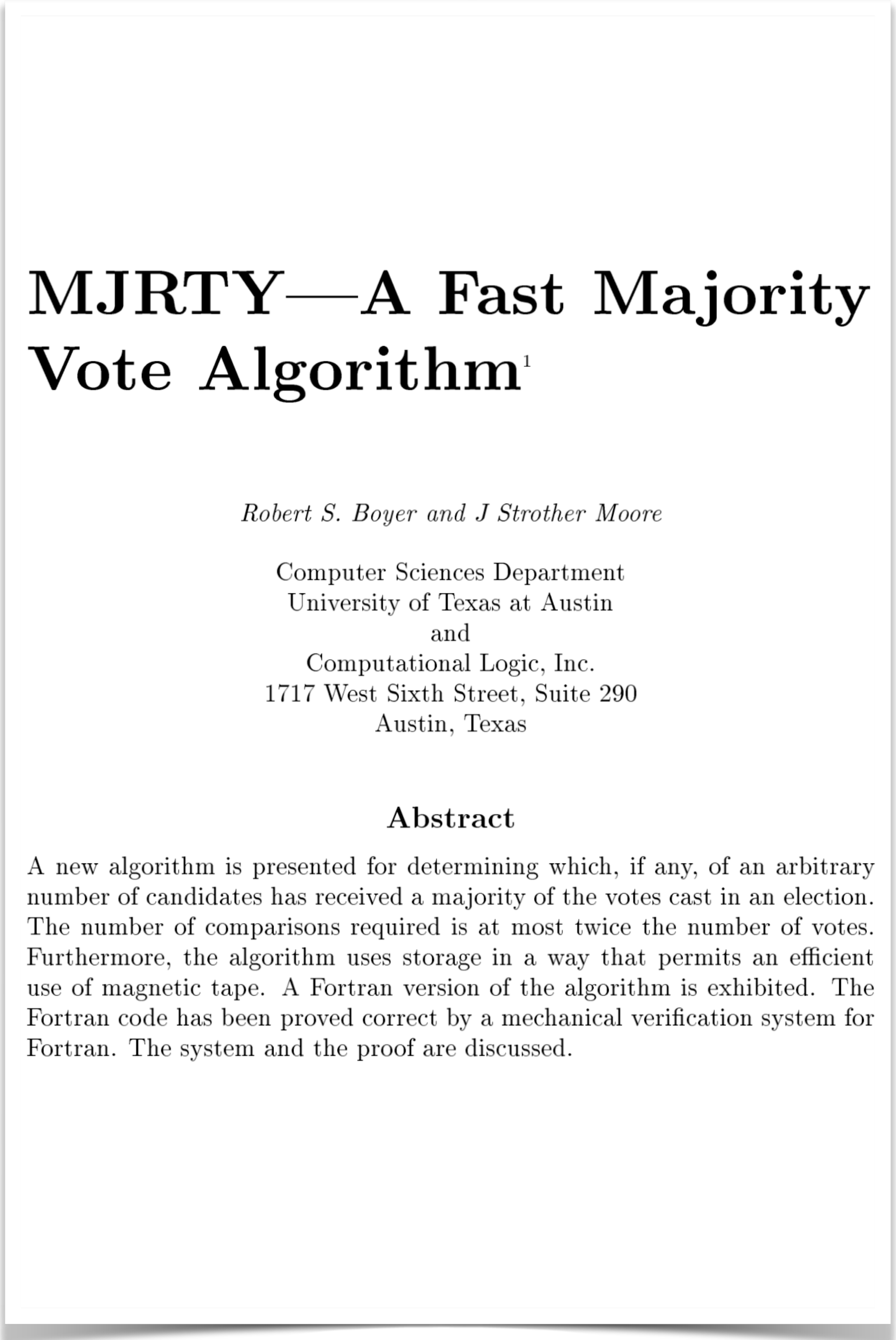
A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient 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]

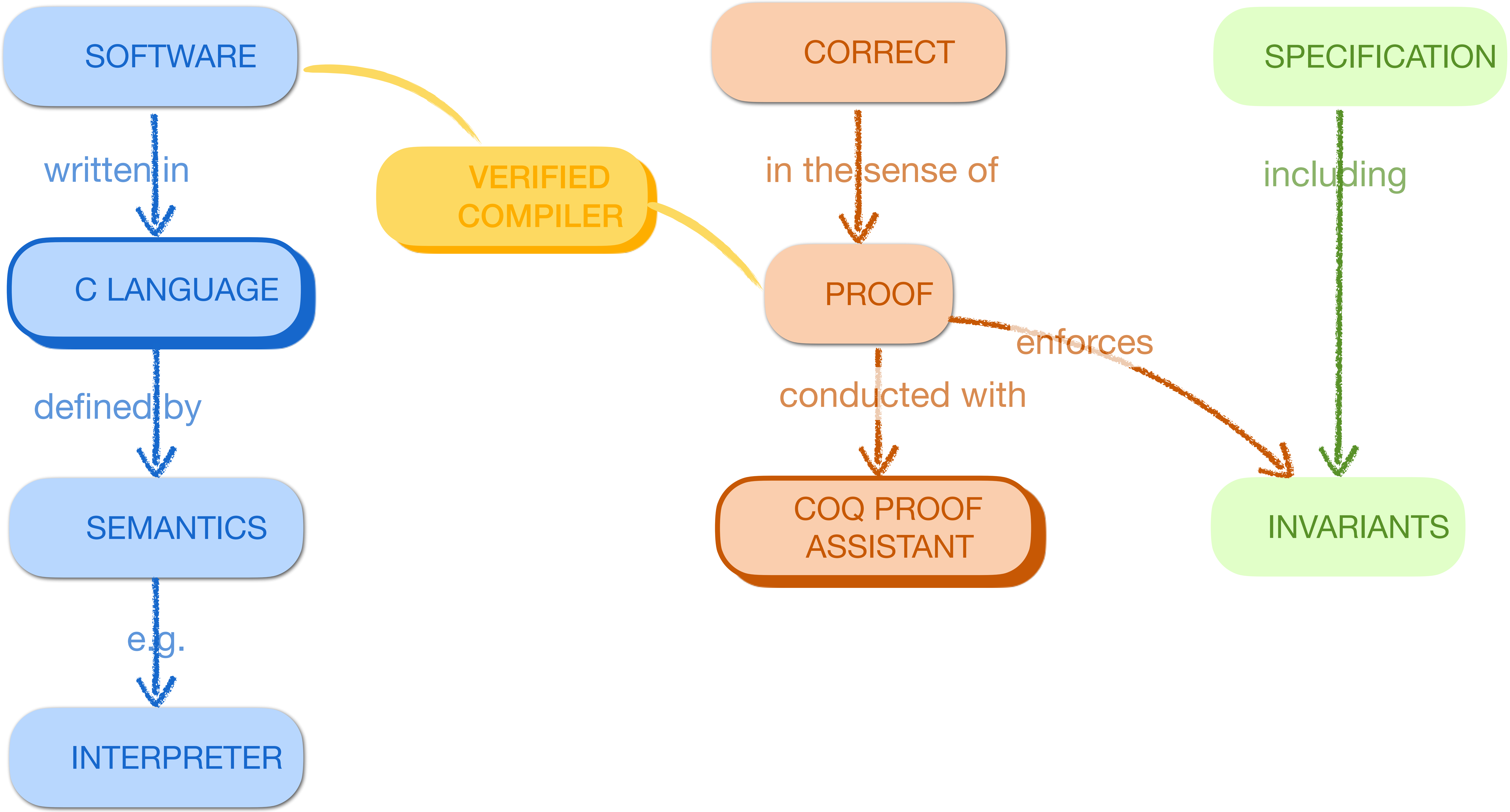
Robert S. Boyer and J Strother Moore

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

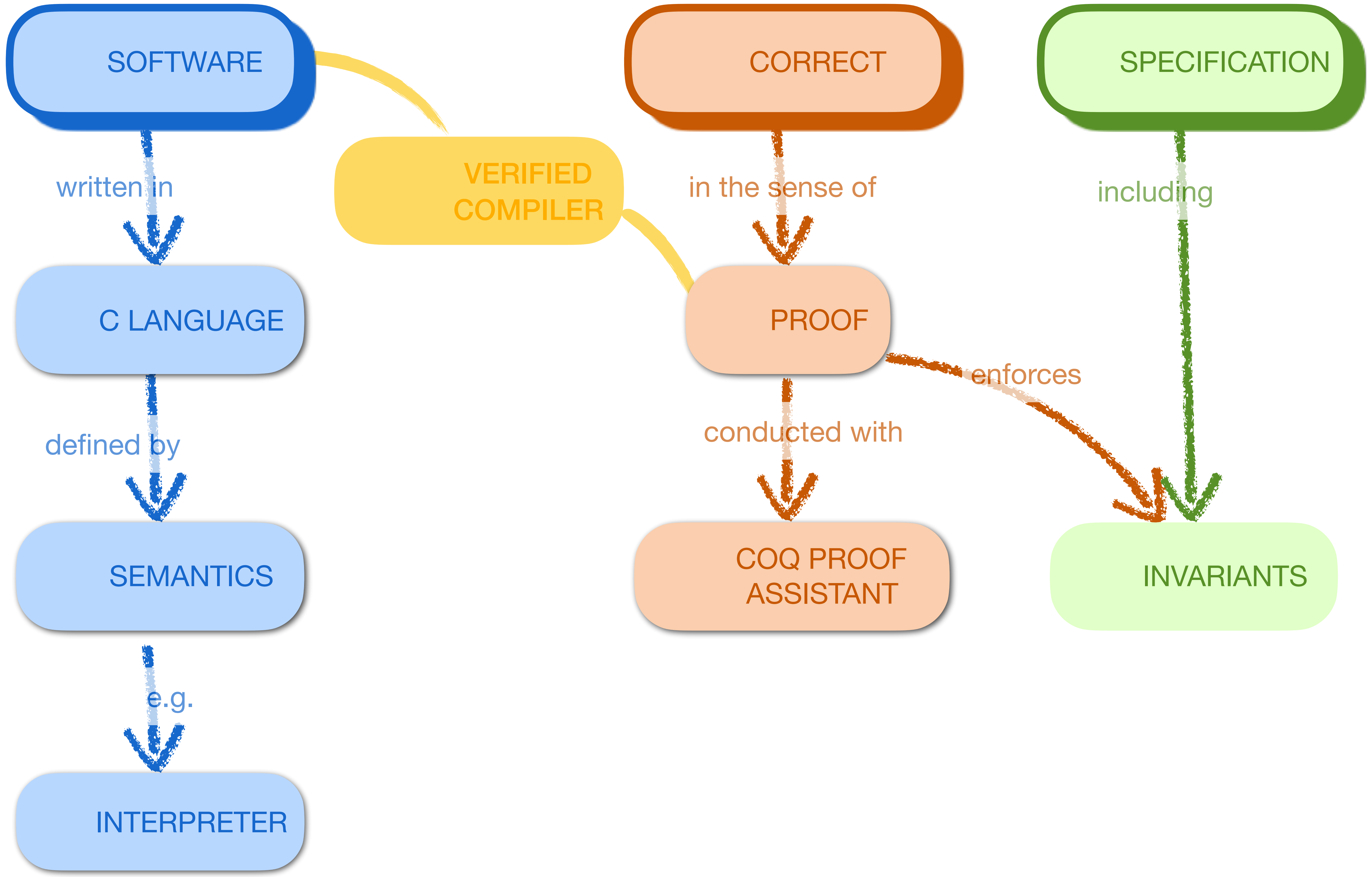
Abstract

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



Part 2:
basics of
verified compilation



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.

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 to Masters students

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

semantics (eval, exec)

compiler (compile)

```
type exp = Nb of int | Id of string | Plus of exp * exp
```

```
type state = string → int
```

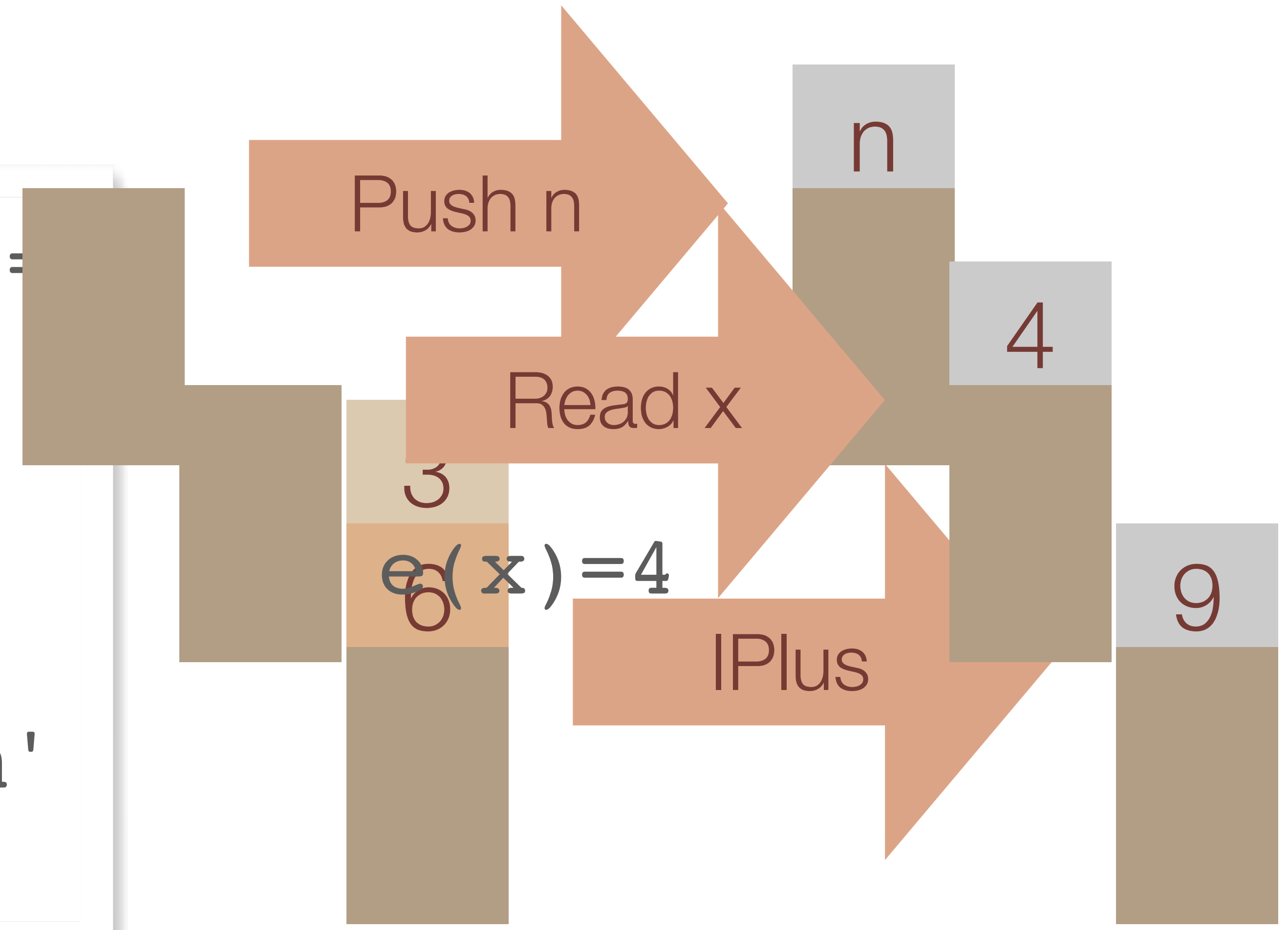
```
let rec eval (e:state)(a:exp): int =
match a with
   | Nb n → n
   | Id x → e x
   | Plus (a1,a2) → (eval e a1)+(eval e
```

compilation

```
let rec compile (a:exp): instr list =  match a with
   | Nb n →  [ Push n ]
   | Id x →  [ Read x ]
   | Plus (a1,a2) → (compile a1)@ (compile a2)@ [IPlus]
```

```
type instr = Push of int | Read of string  | IPlus
```

```
let rec exec (e:state)(stack: int list)(pgm: instr list): int list =
  match (pgm, stack) with
   | ([], _) → stack
   | (Push n :: pgm', _) → exec e (n :: stack) pgm'
   | (Read x :: pgm', _) → exec e (e x :: stack) pgm'
   | (IPlus :: pgm', n:: m :: stack') → exec e ((m+n) :: stack') pgm'
   | (_ :: pgm', _) → exec e stack pgm'
```



Proving a property with the Coq software

ACM SIGPLAN Programming Languages Software award 2013
ACM Software System award 2013 coq.inria.fr

```
Theorem toy-compiler-correct:
  forall e a,
  exec e [] (compile a) = [eval e a].
```

semantics
(**eval**, **exec**)

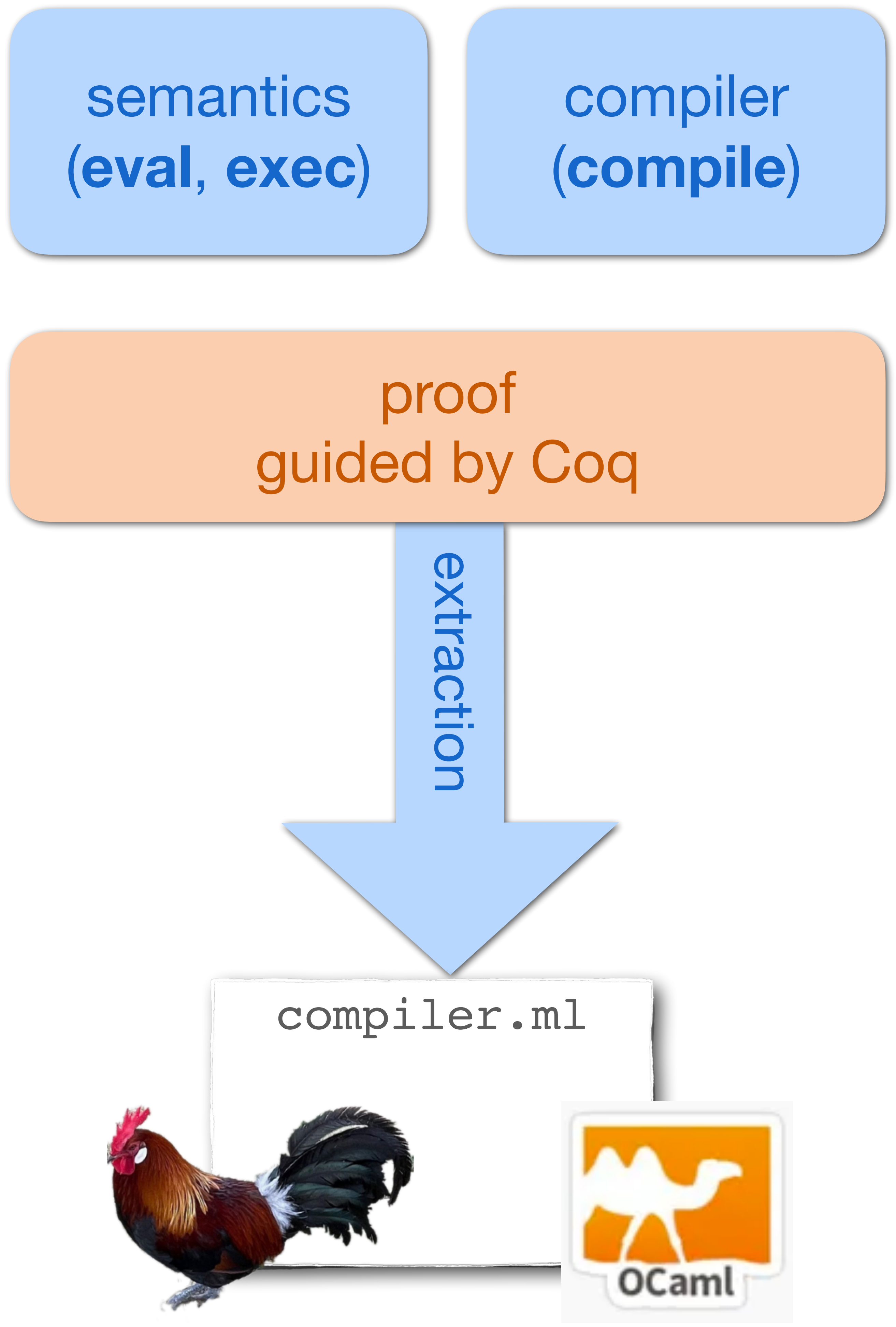
compiler
(**compile**)

Proving a property with the Coq software

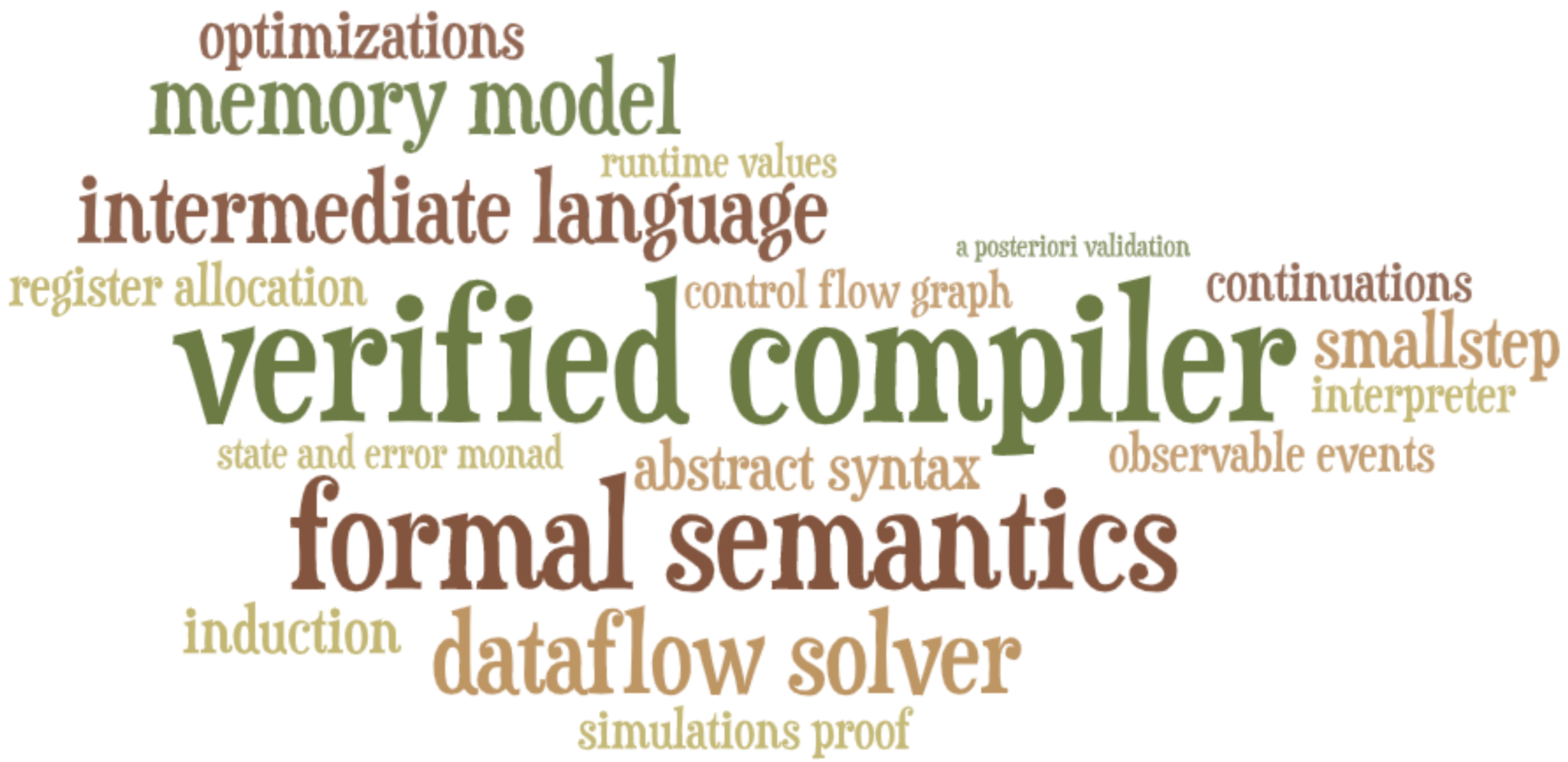
ACM SIGPLAN Programming Languages Software award 2013
ACM Software System award 2013 coq.inria.fr

```
Theorem toy-compiler-correct:
  forall e a,
  exec e [] (compile a) = [eval e a].
Proof.
  intros;
  … (* not shown here *)
Qed.
```

```
Extraction compile.
```



Part 3
How to turn CompCert from a prototype in a lab into a real-world 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)

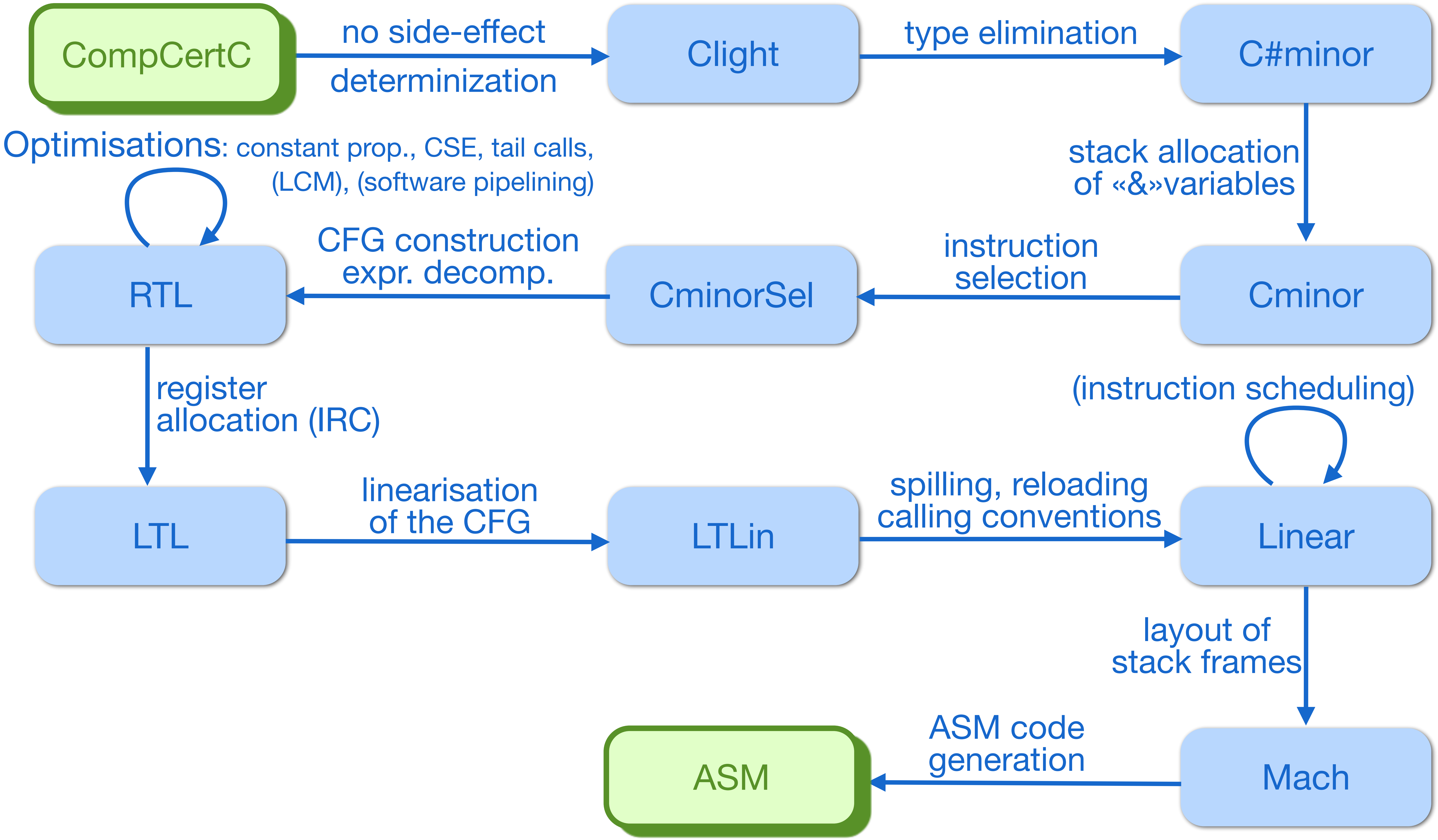
Programmed and verified using the Coq proof assistant

Shared infrastructure for ongoing research

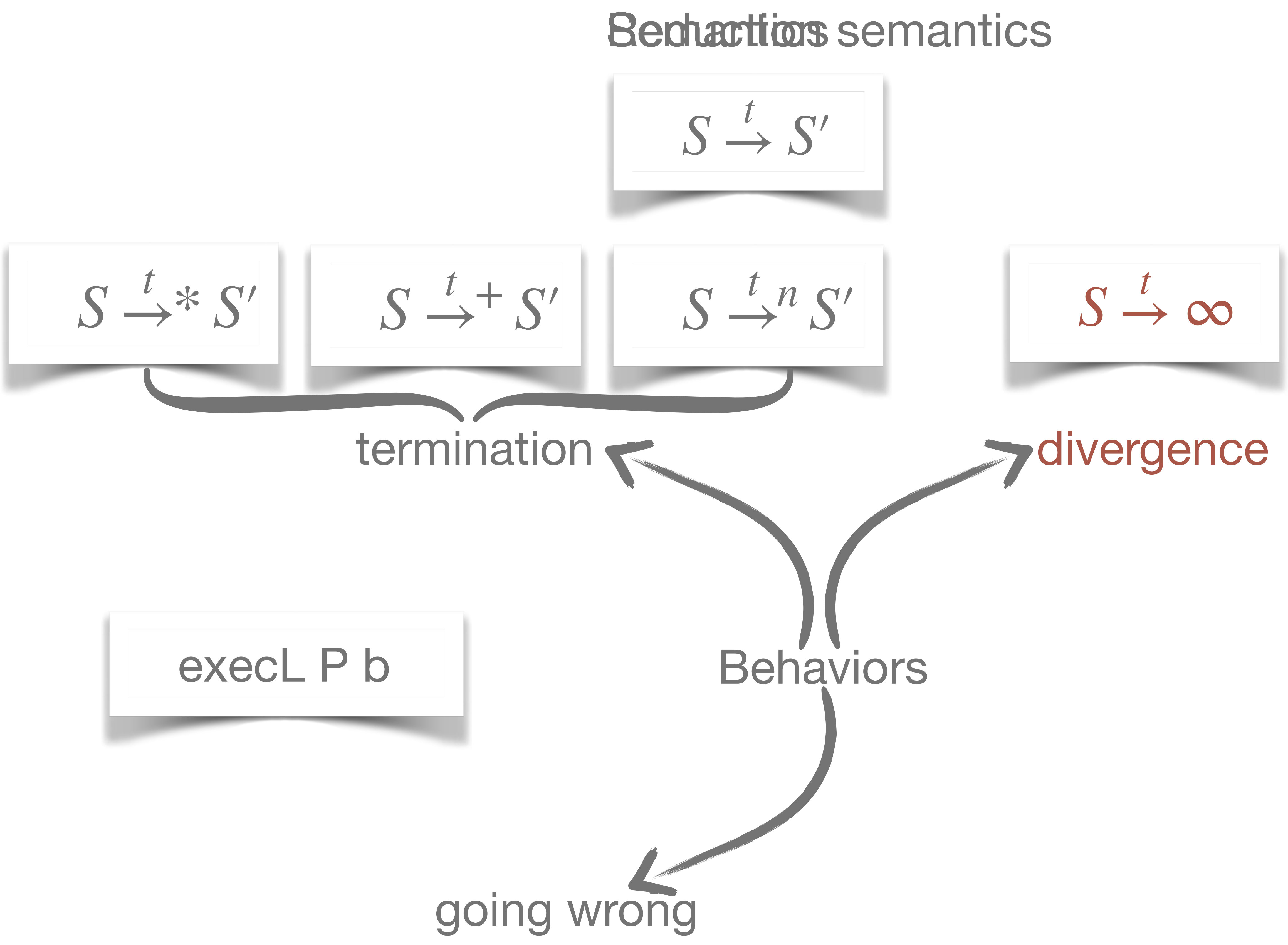
Used in commercial settings (for emergency power generators and flight control navigation algorithms) and for software certification - AbsInt company
Improved performances of the generated code while providing proven traceability information

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

CompCert compiler: 11 languages, 18 passes



CompCert compiler: 11 languages, 18 passes



Proving semantics preservation: the simulation approach

semantics (**execCompCertC**, **execASM**)

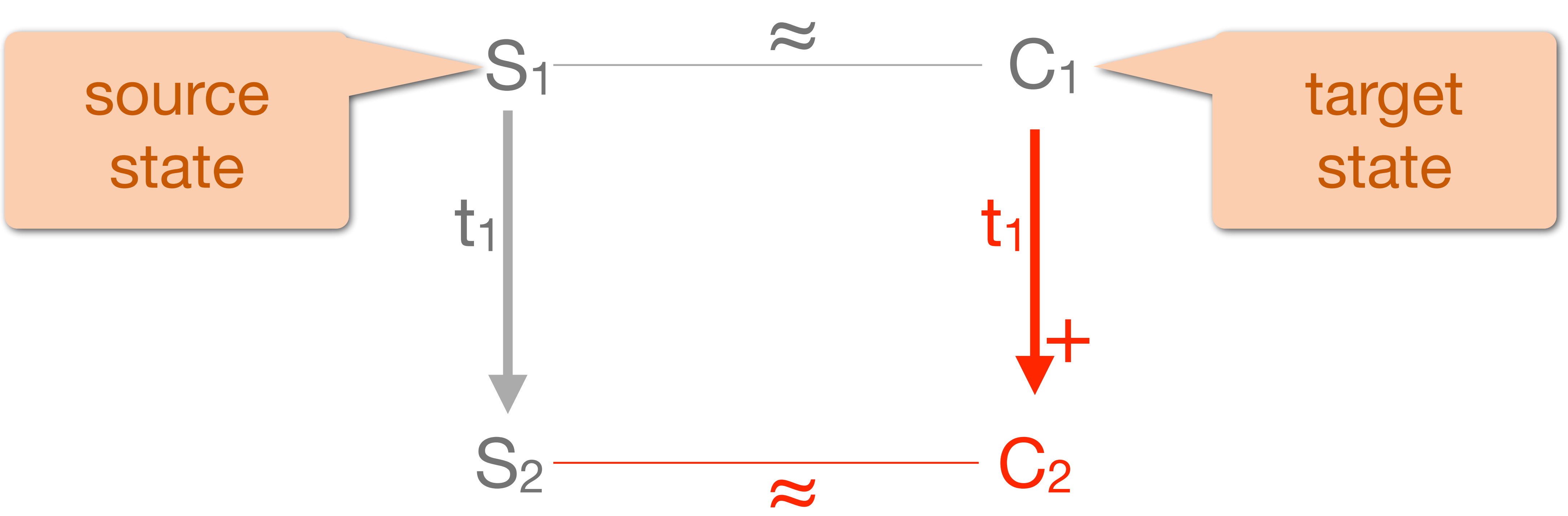
compiler

Preserved behaviors = termination and divergence

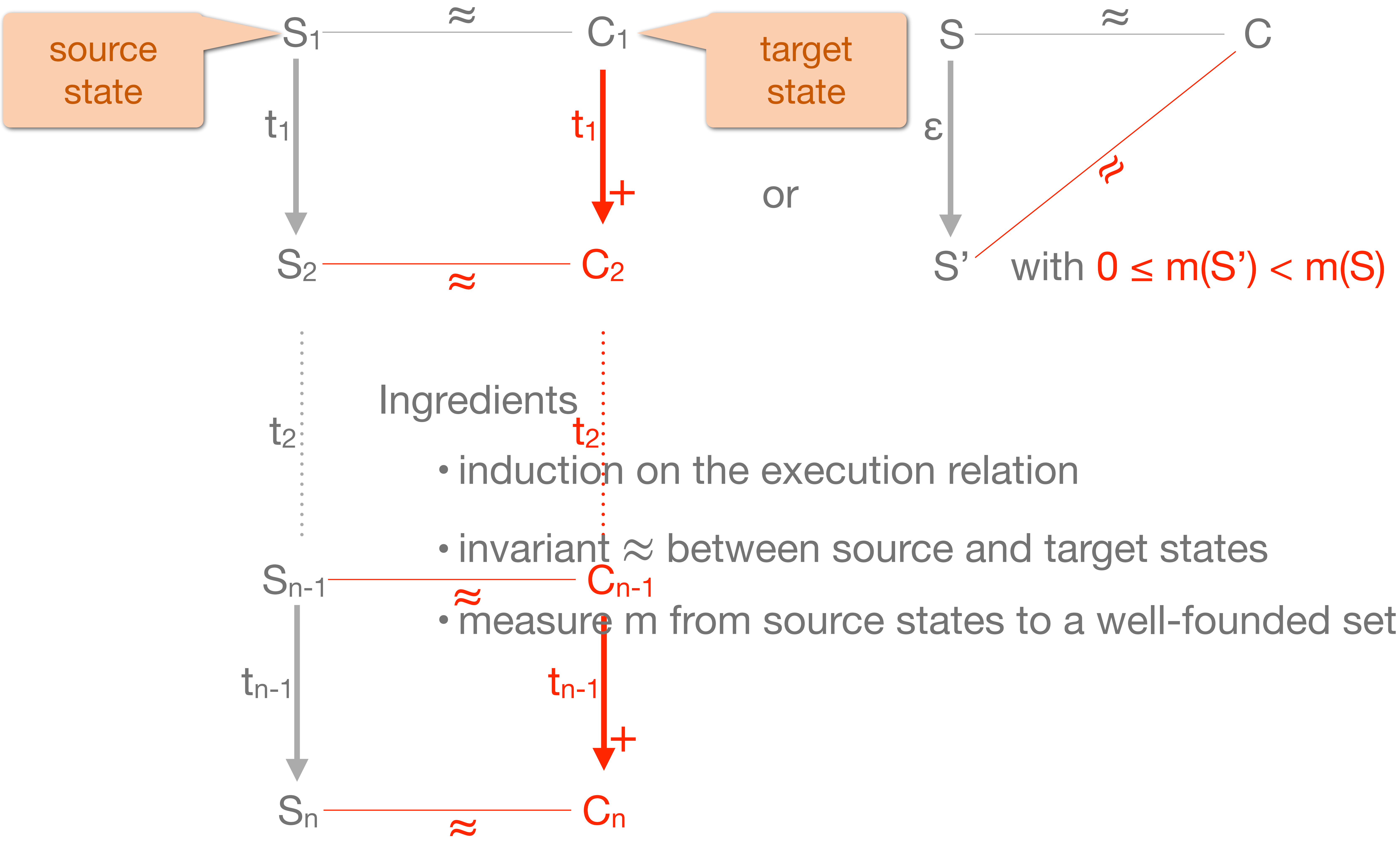
```
Theorem compiler-correct:
  ∀ S C b,
  compiler S = OK C →
  execCompCertC S b →
  execASM C b.
```

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

Proof technique: simulation diagram



Proving semantics preservation: the simulation approach



Ingredients

- induction on the execution relation
- invariant ≈ between source and target states
- measure m from source states to a well-founded set

Which operational semantics for C-like languages?

Reduction semantics to model diverging executions

i / s → i' / s'

Some rules generate instructions that do not exist in the source program.

(while b do i) / s → i; while b do i / s   when eval s b = true

(if b then i1 else i2); i / s → i1; i / s   when eval s b = true

Raises two issues when using simulation diagrams:

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

Continuation-based semantics to the rescue

[Appel, Blazy TPHOL'07]

i / k / s → i' / k' / s'

Continuation: remaining computations and their structure

No generation of new instruction: i' is always a subterm of i

(if b then i1 else i2) / k / s → i1 / k / s   when eval s b = true

New kinds of rules for dealing with continuations

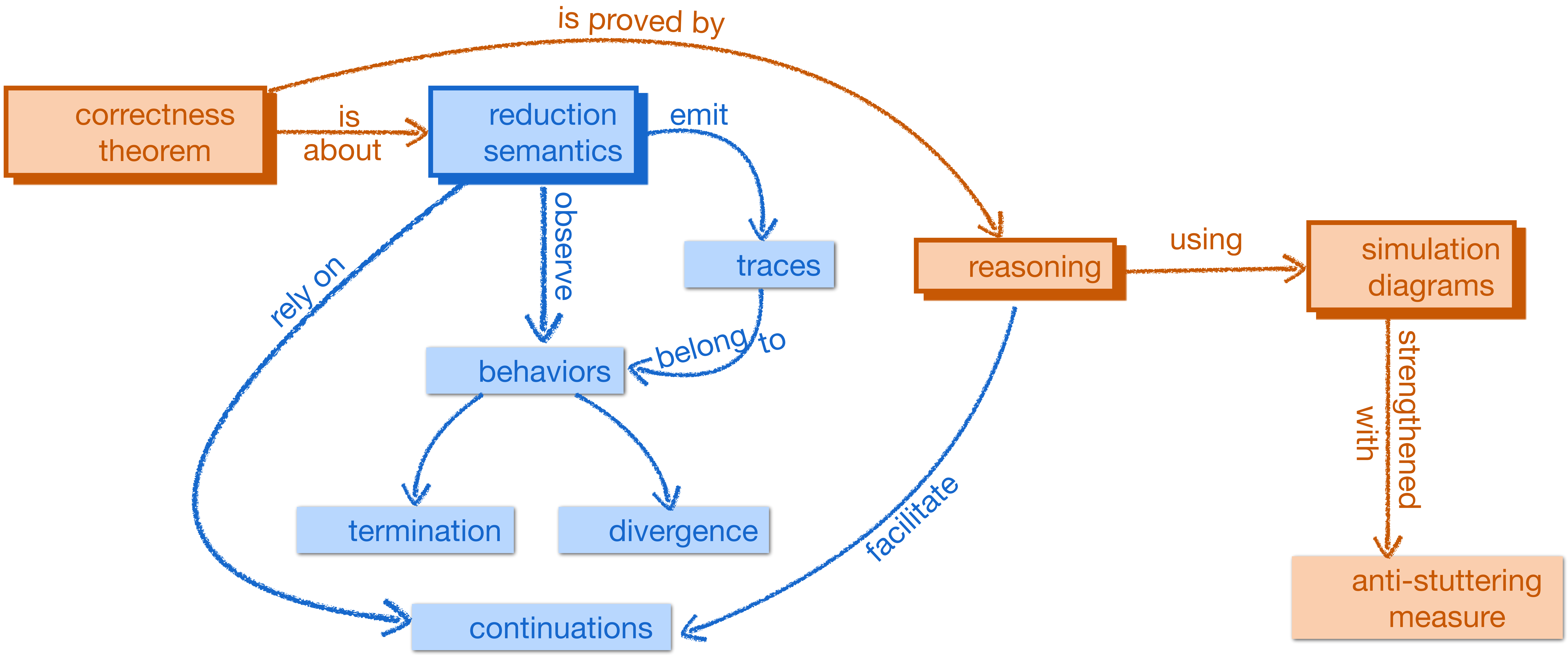
(i1;i2) / k / s → i1 / i2 • k / s

Focus (on the left of a sequence)

skip / i • k / s → i / k / s

Resume (the remaining computations)

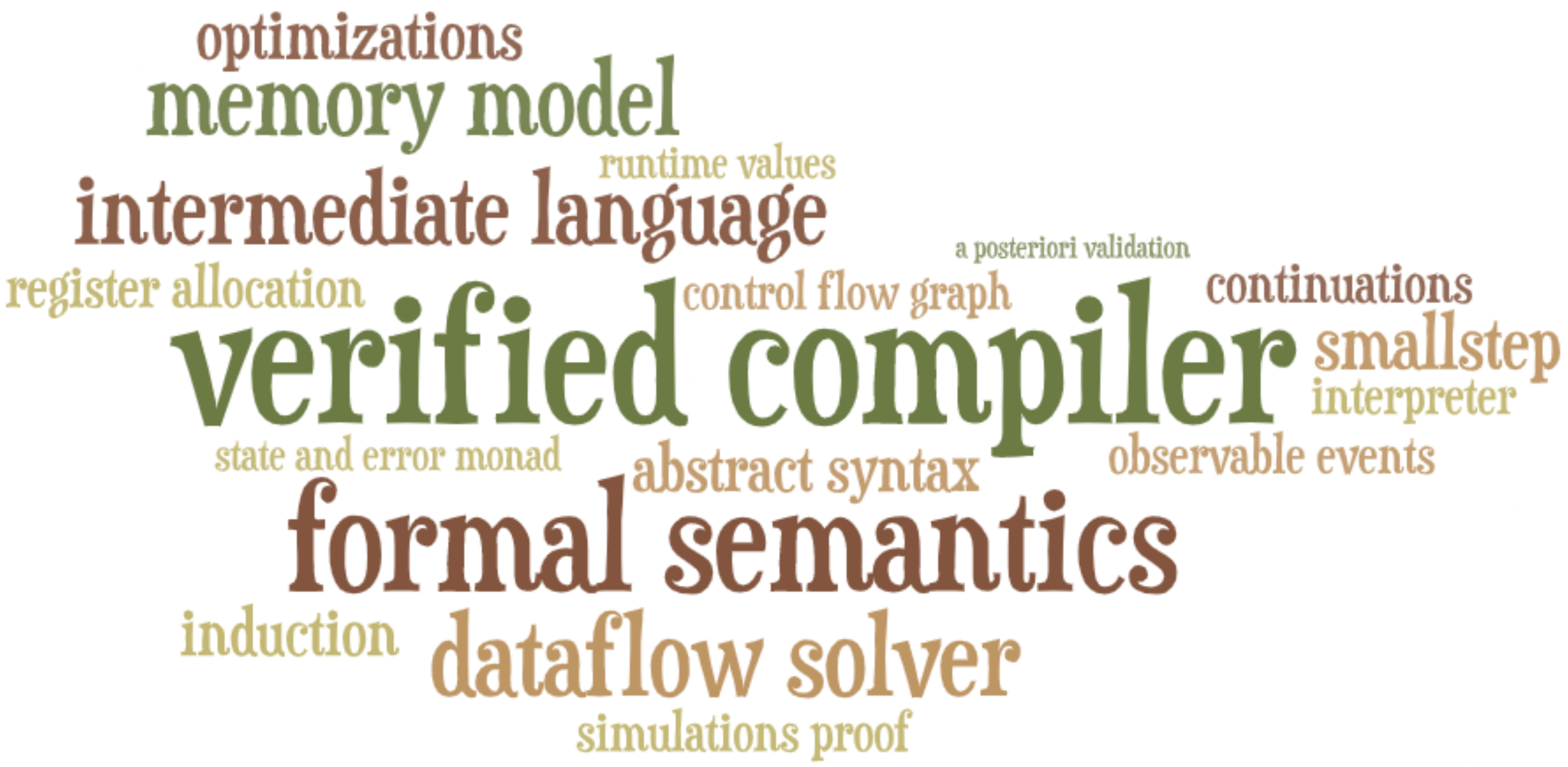
Part 3: summary



Part 4

Beyond CompCert

- secure compilation
- just-in-time compilation
- WIP



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

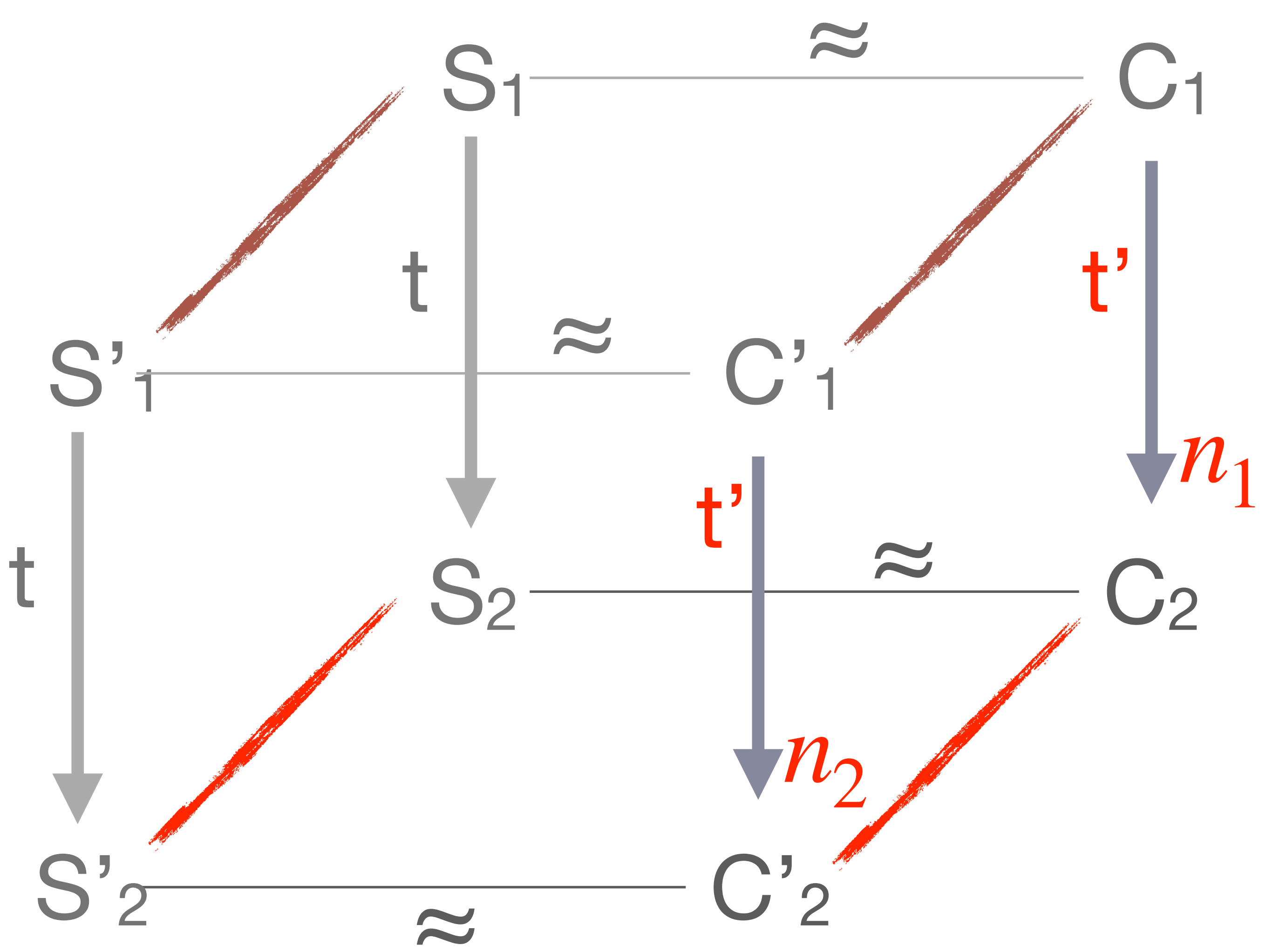
$S_1 \xrightarrow{\ell} S_2$

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

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

implies $\ell = \ell'$

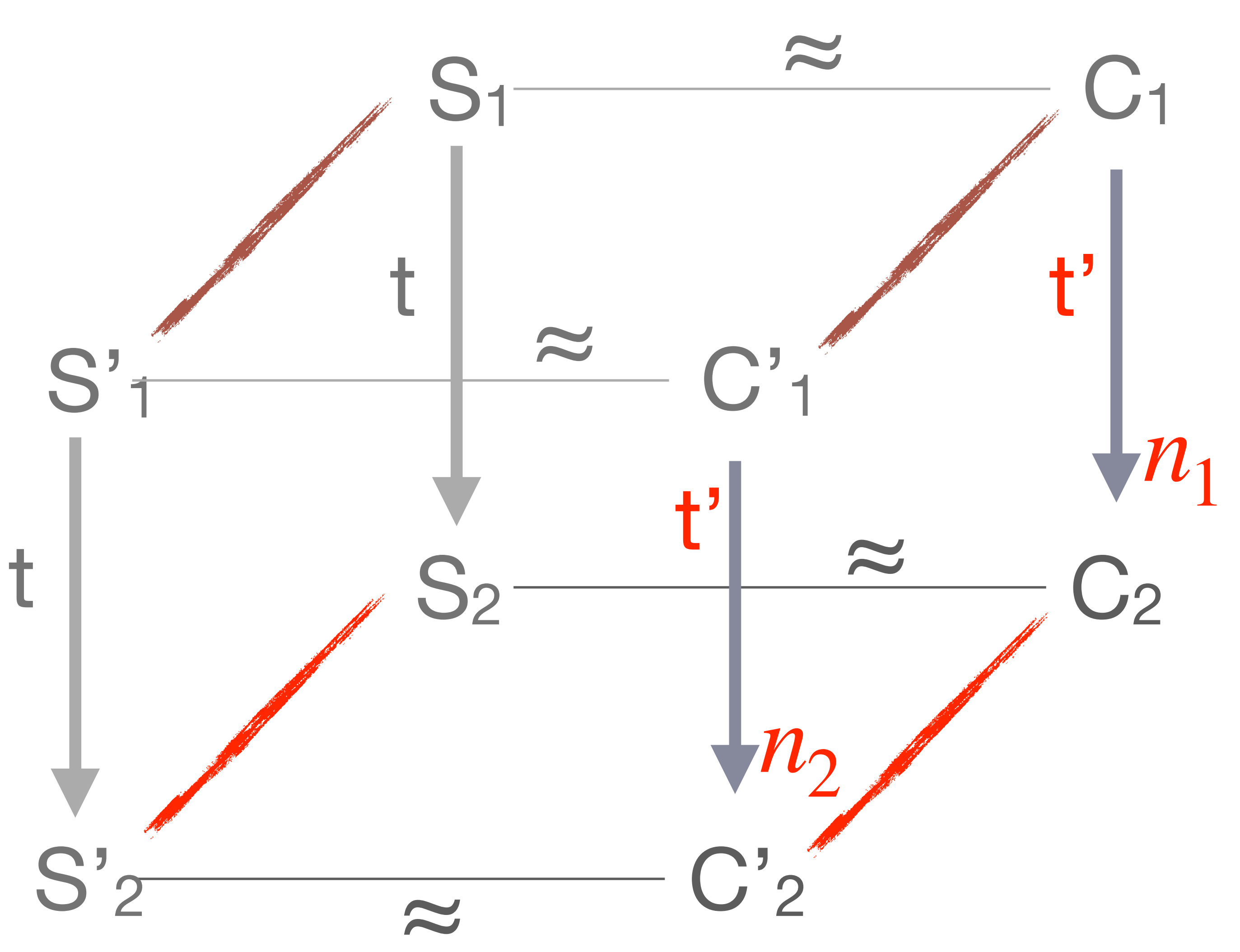
Indistinguishability property $\varphi(S_i, S'_i)$: share public values, but may differ on secret values

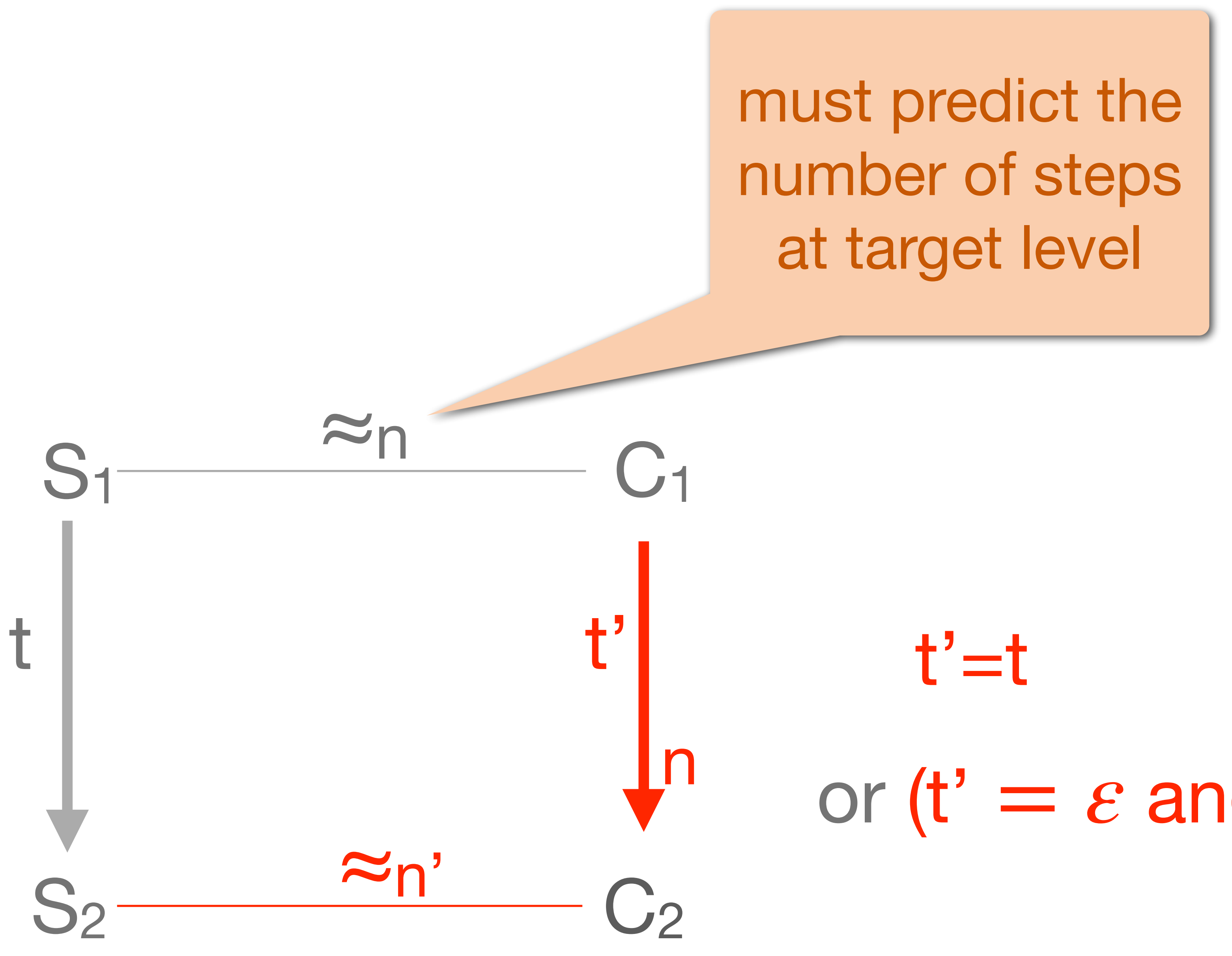


Difficulty: tricky proofs!

Proving CCT preservation: back to simulation diagrams

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



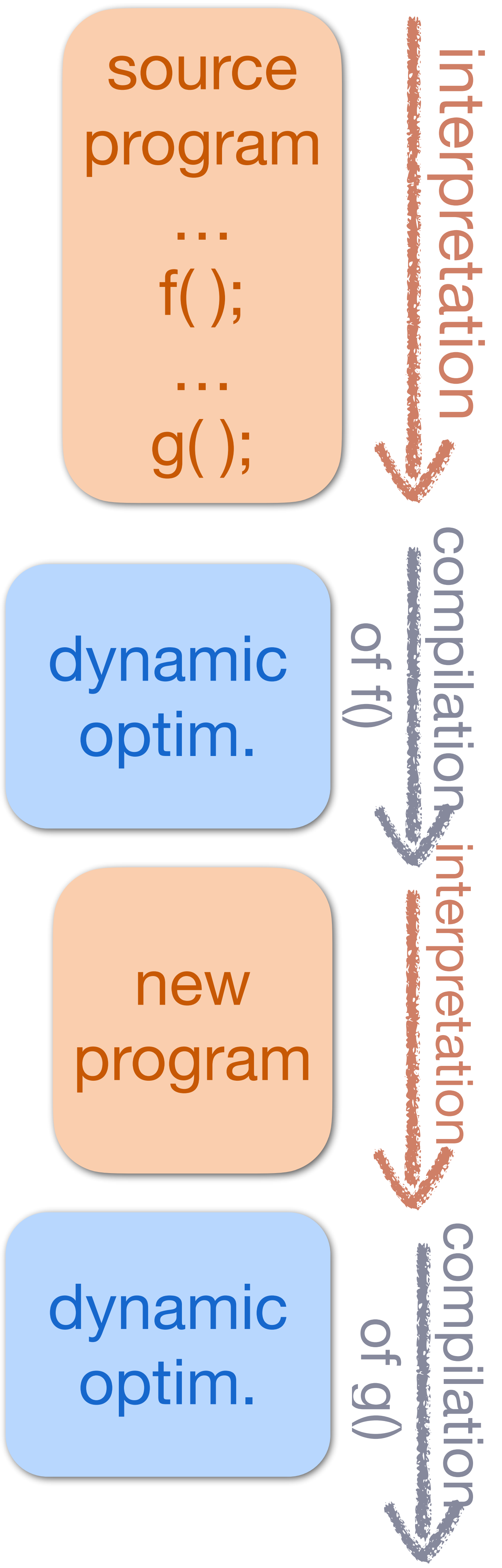


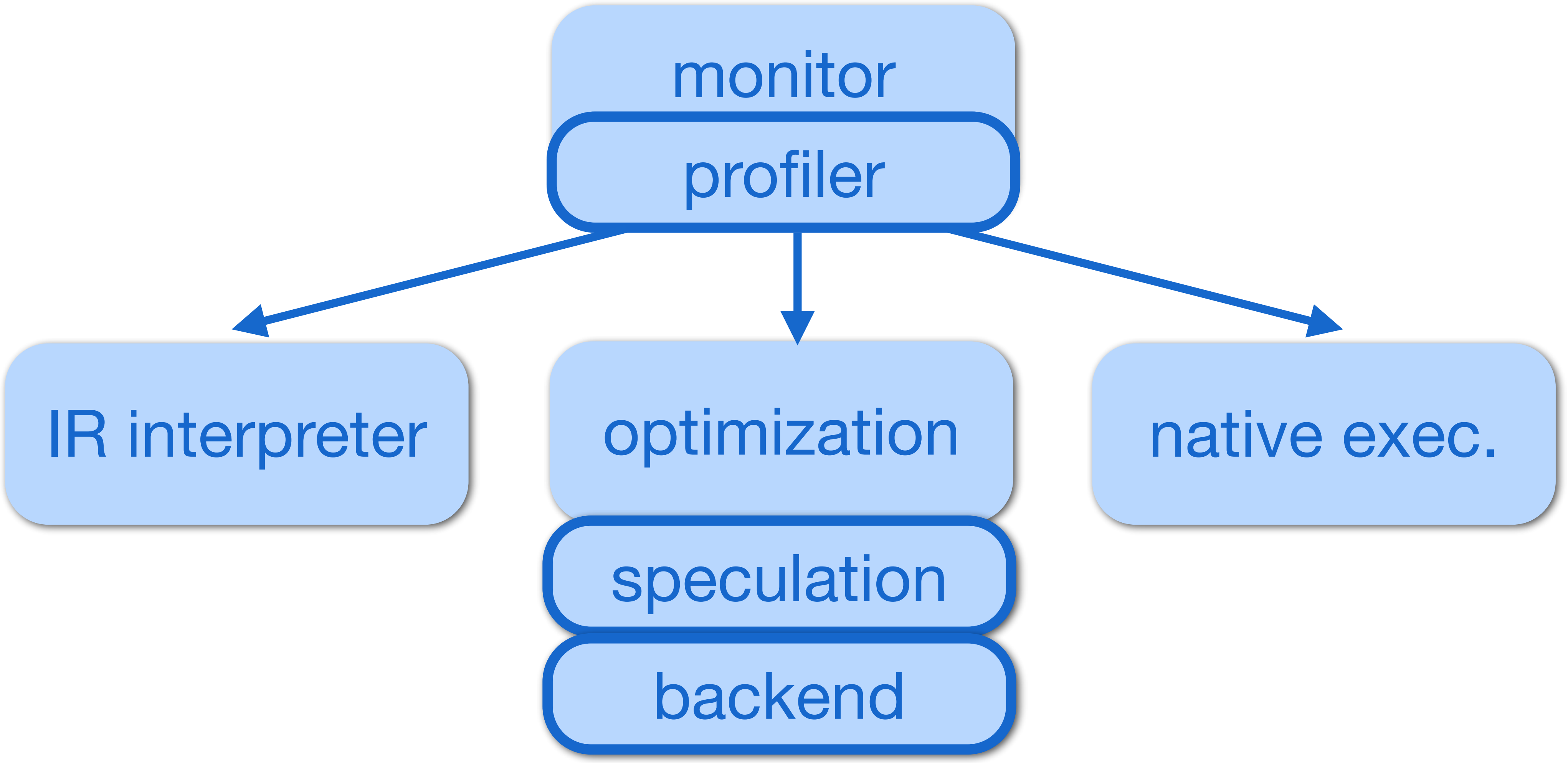
t'=t

or (t' $=$ ε 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]



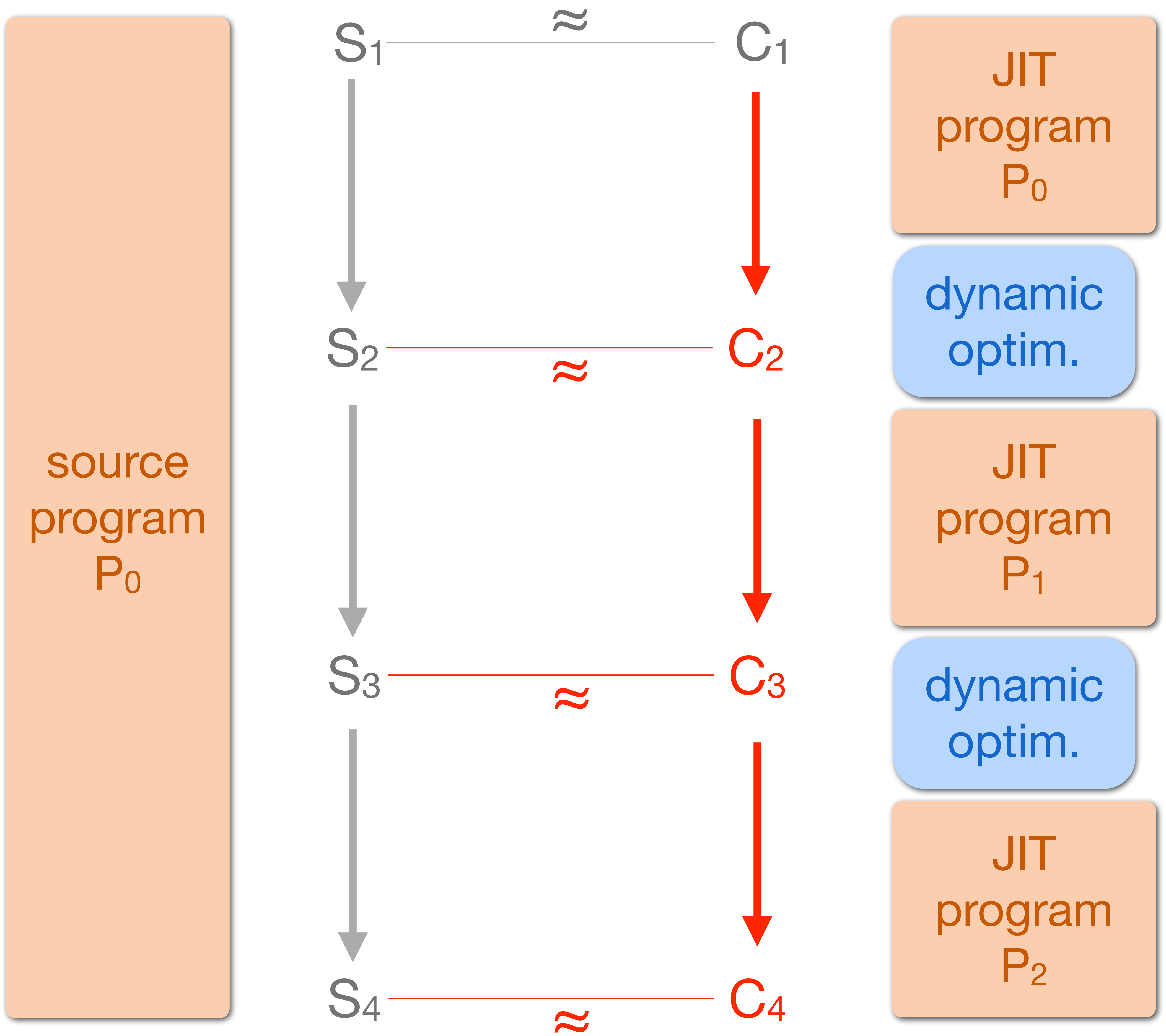


A JIT compiler **interleaves** the execution of a program with its optimizations

Dynamic speculation: specializes functions, requires deoptimization

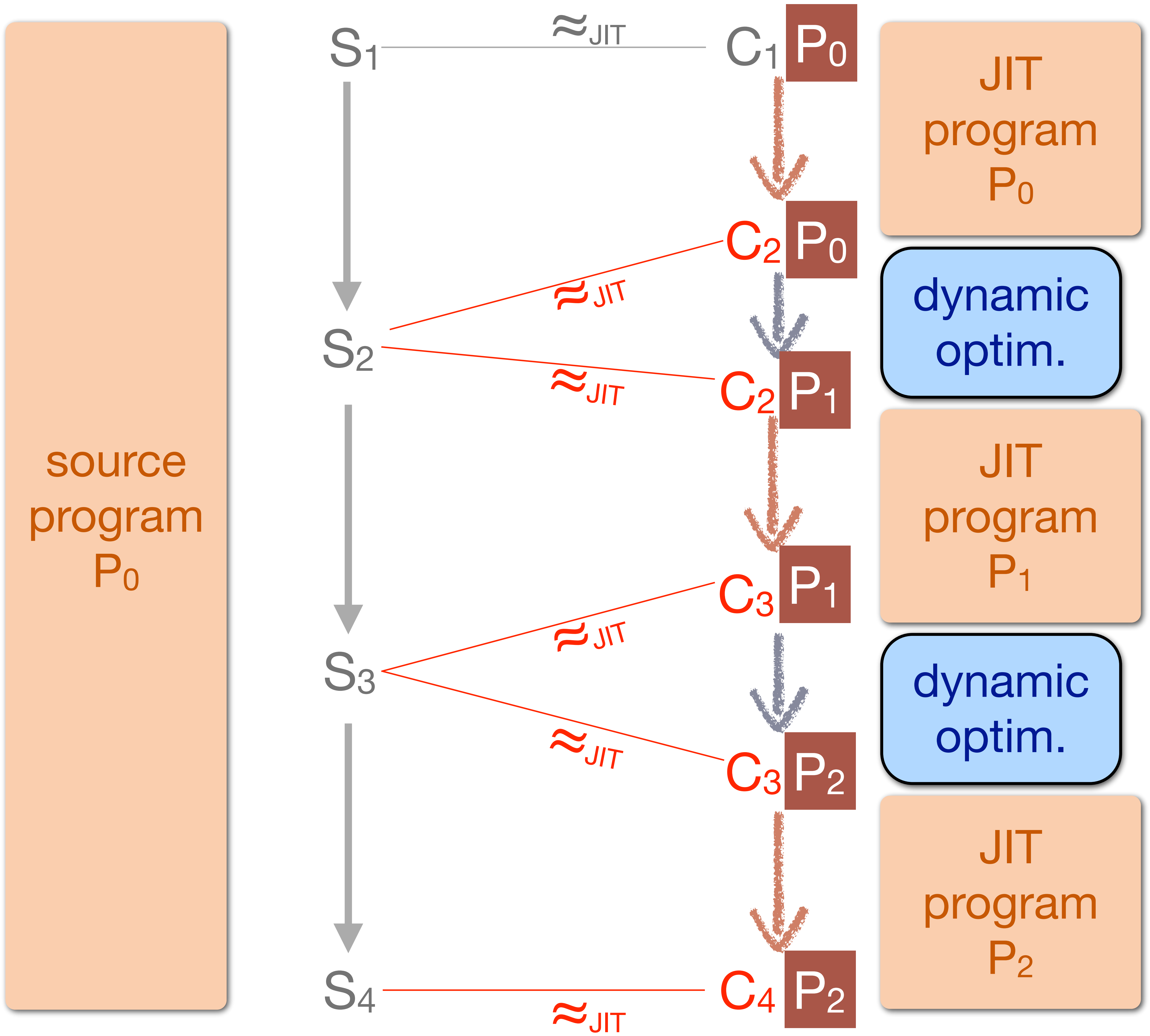
Non-deterministic semantics: either deoptimize to the source program or continue to the next instruction in the optimized program

Proving semantics preservation: the simulation approach



Both the program and the execution state are evolving

Nested simulations for JIT verification



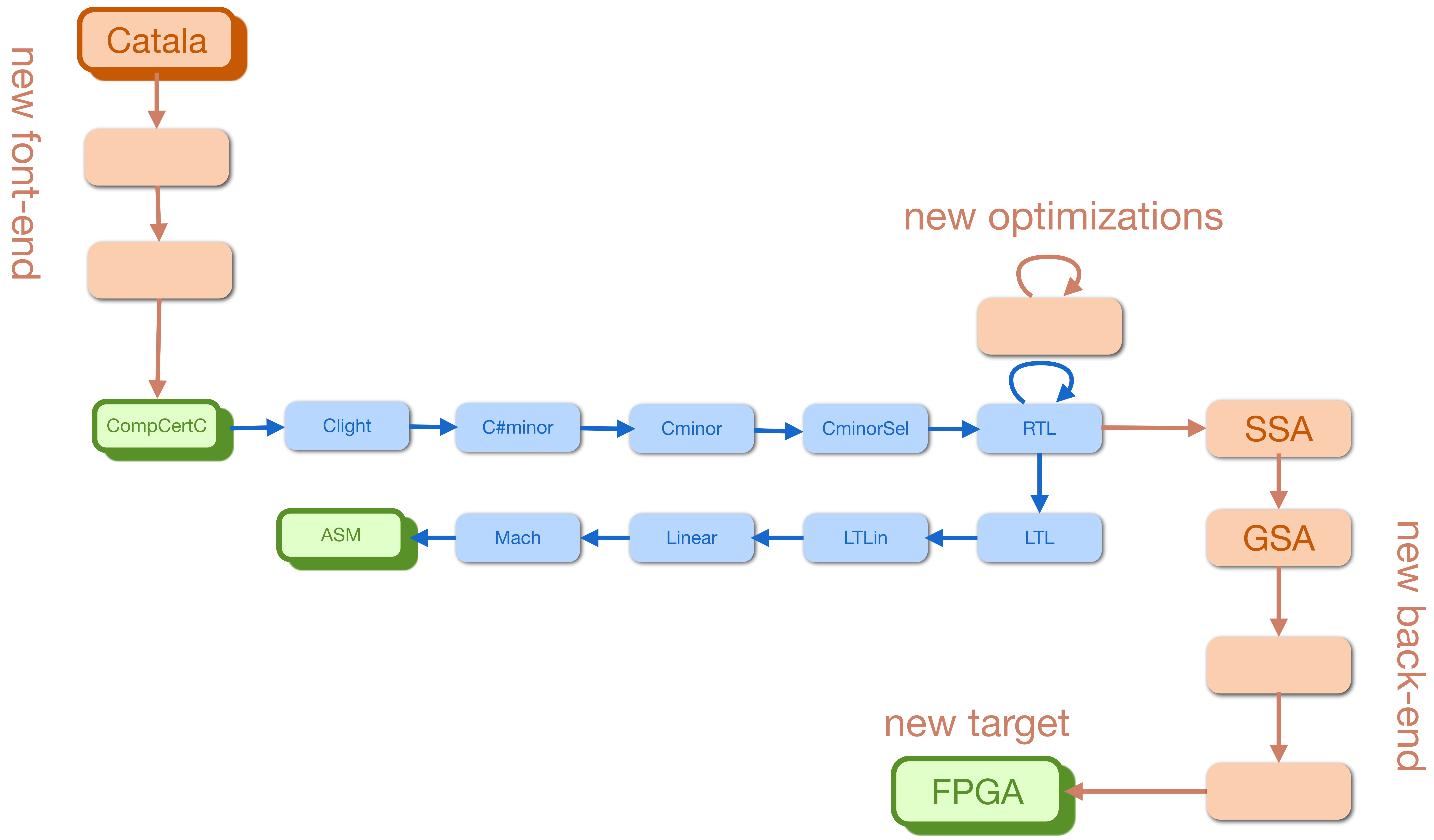
Both the program and the execution state are evolving

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 🏗



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

Thank you! Questions?