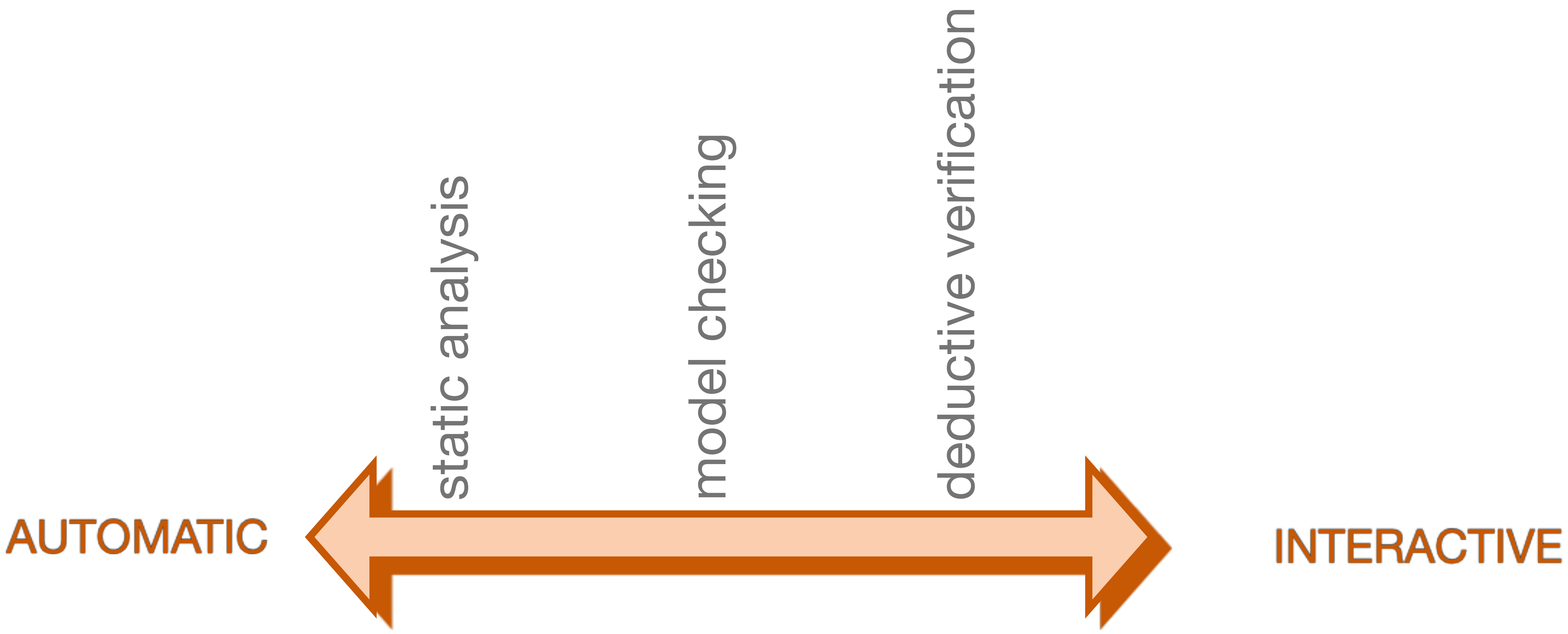
Verified compilation: towards zero-defect software

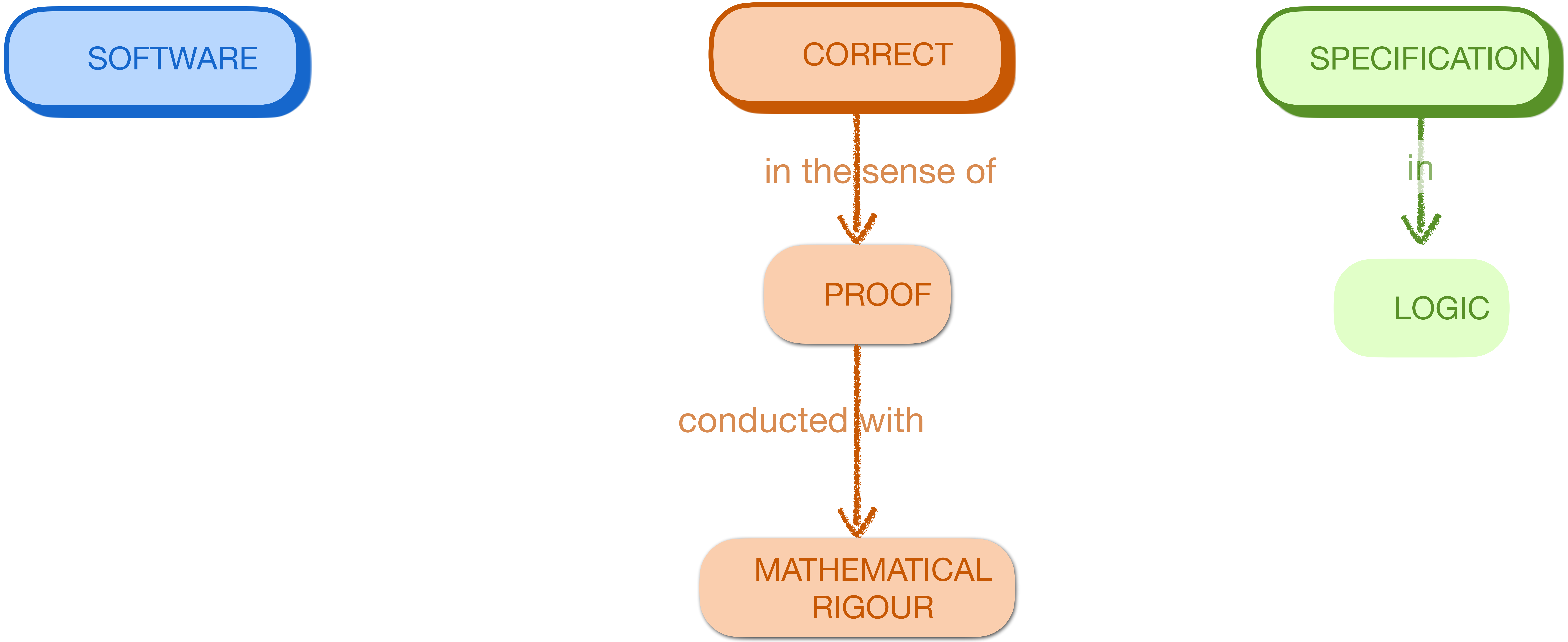
Sandrine Blazy

Colloquium d'informatique de Sorbonne Université, Paris, 2024-11-26

Formal verification of software: tool-assisted techniques



Deductive verification



SOFTWARE

CORRECT

in the sense of

PROOF

conducted with

MATHEMATICAL RIGOUR

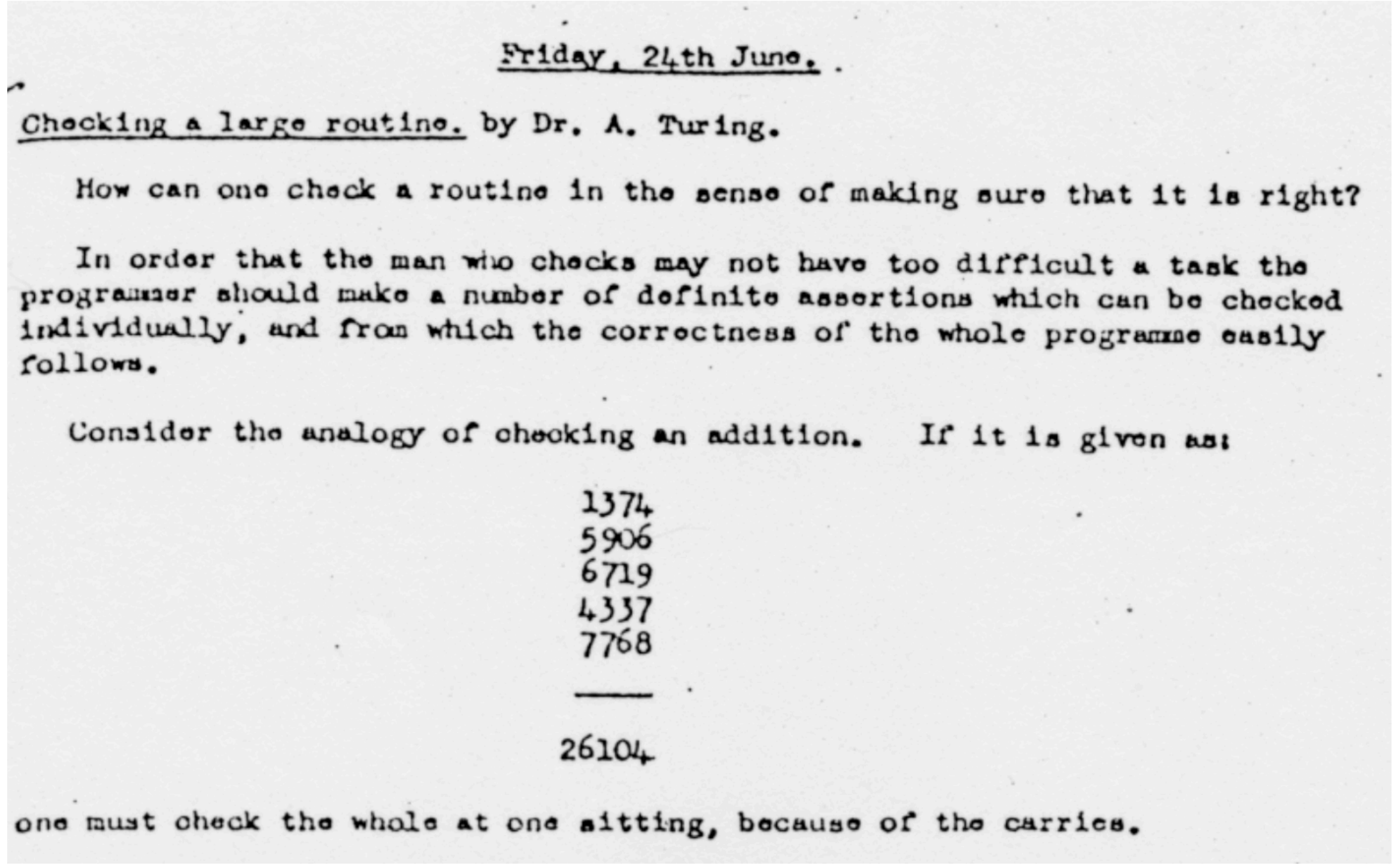
SPECIFICATION

in

LOGIC

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?

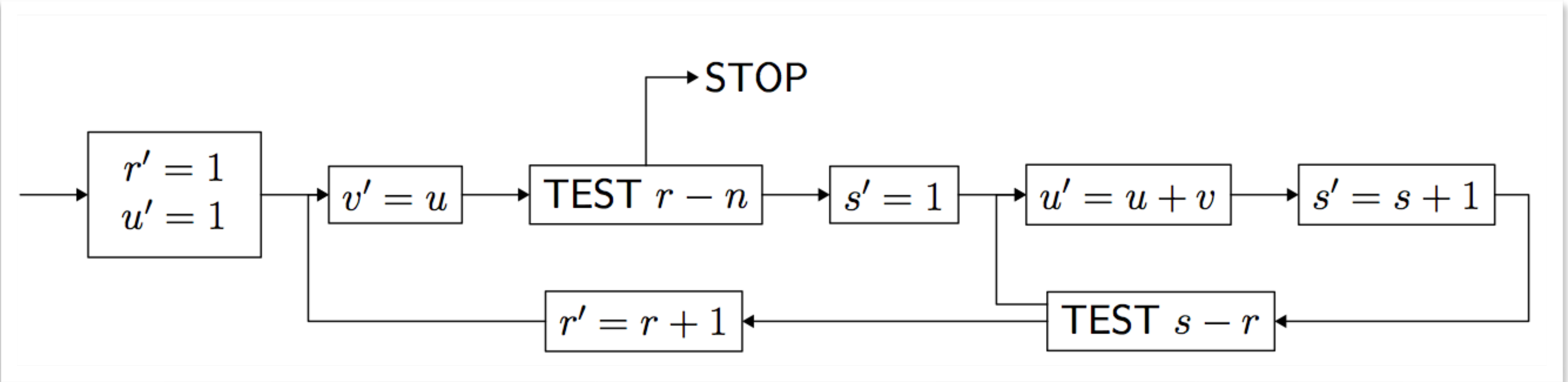
In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

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

1374
5906
6719
4337
7768

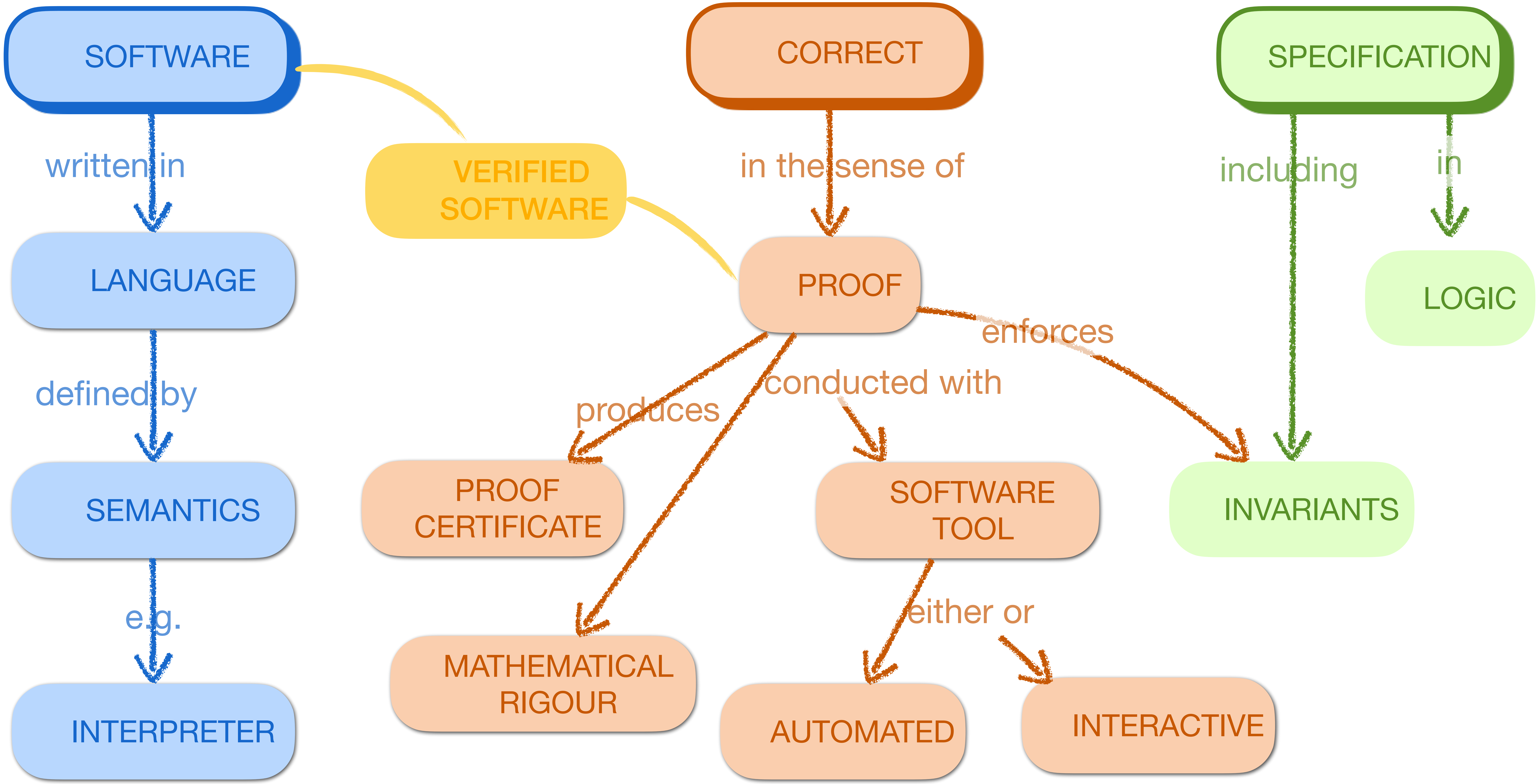
26104

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



… to deductive-verification and automated tools

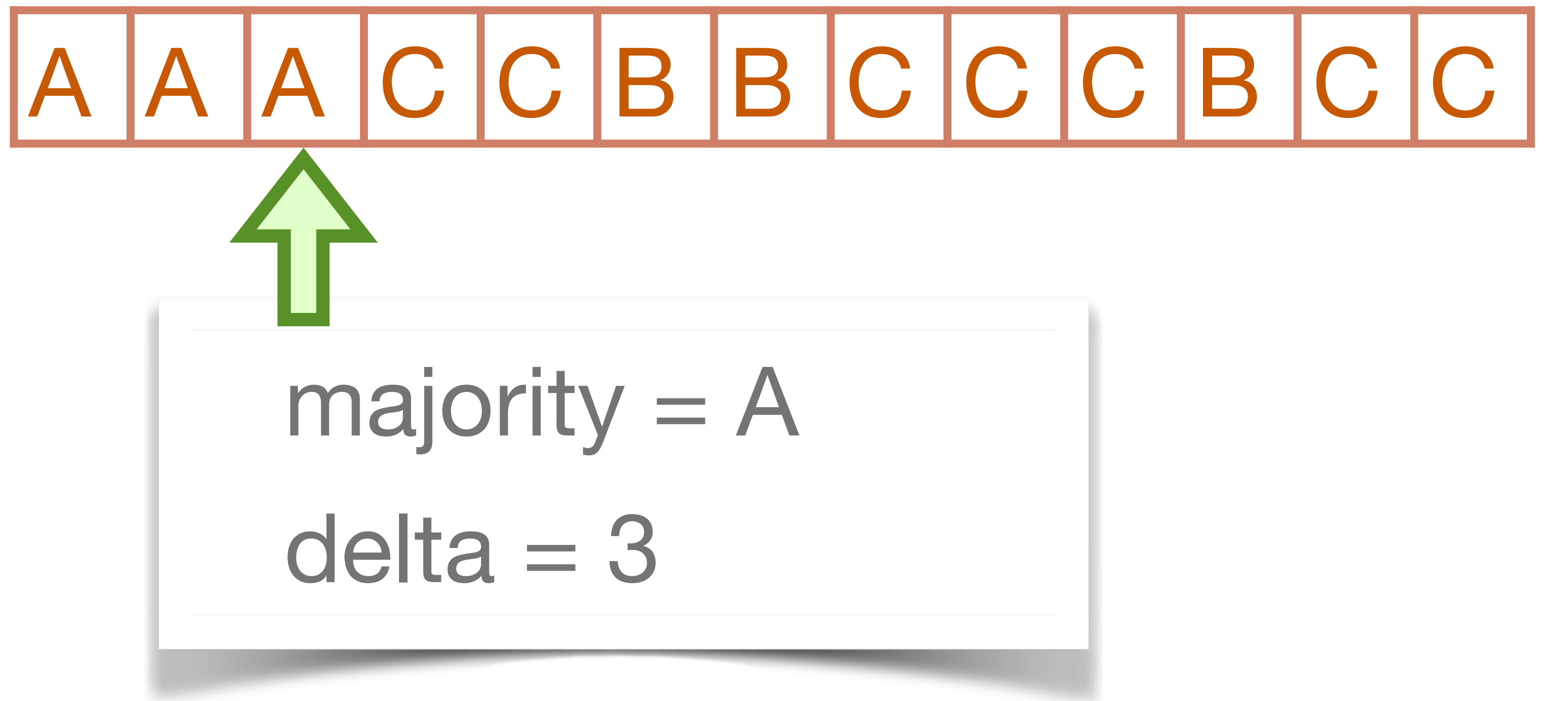
Floyd 1967, Hoare 1969



Another historical example

Boyer-Moore's majority. 1980

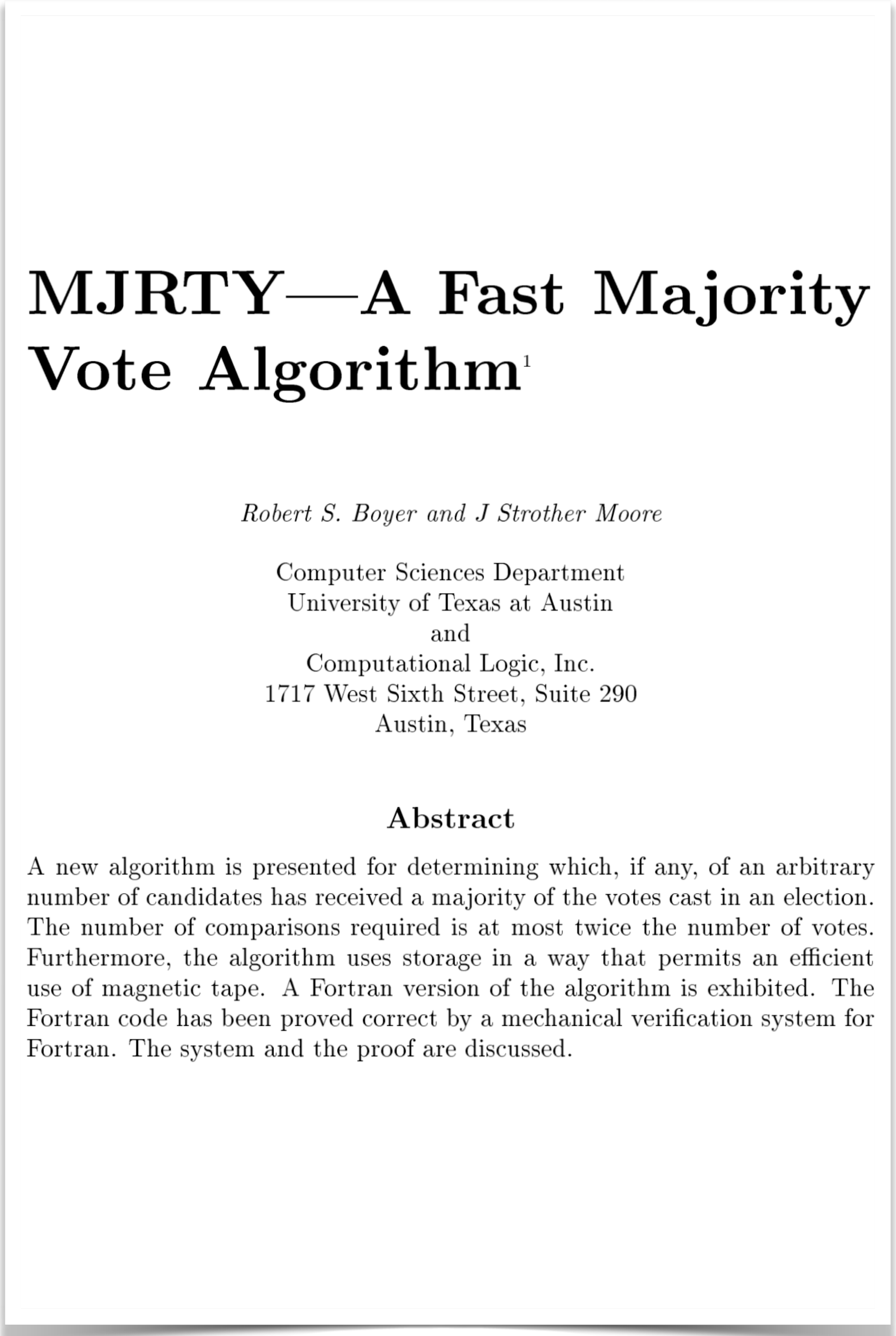
Given N votes, determine the majority if any



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

majority = A

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

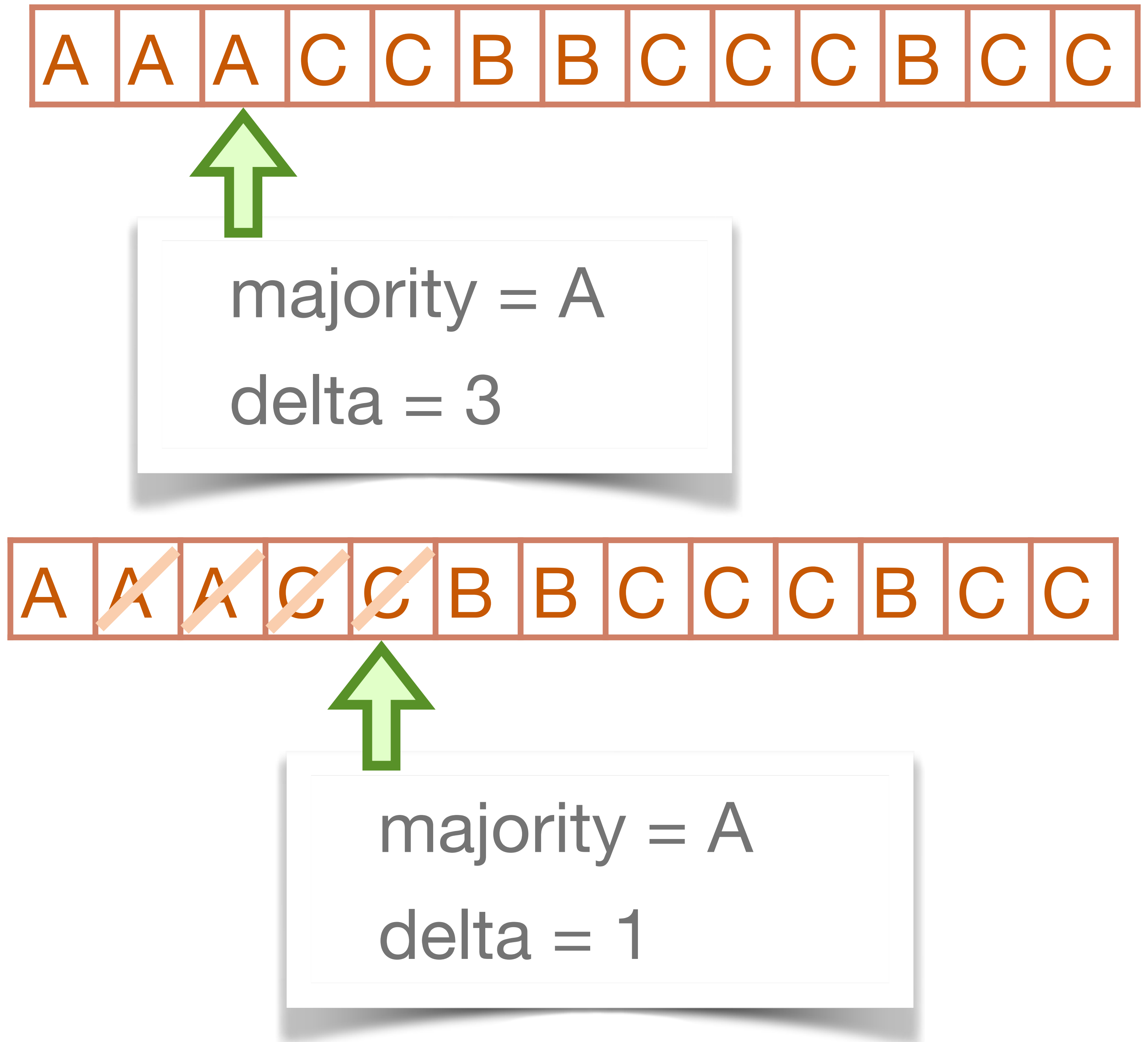
Abstract

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



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

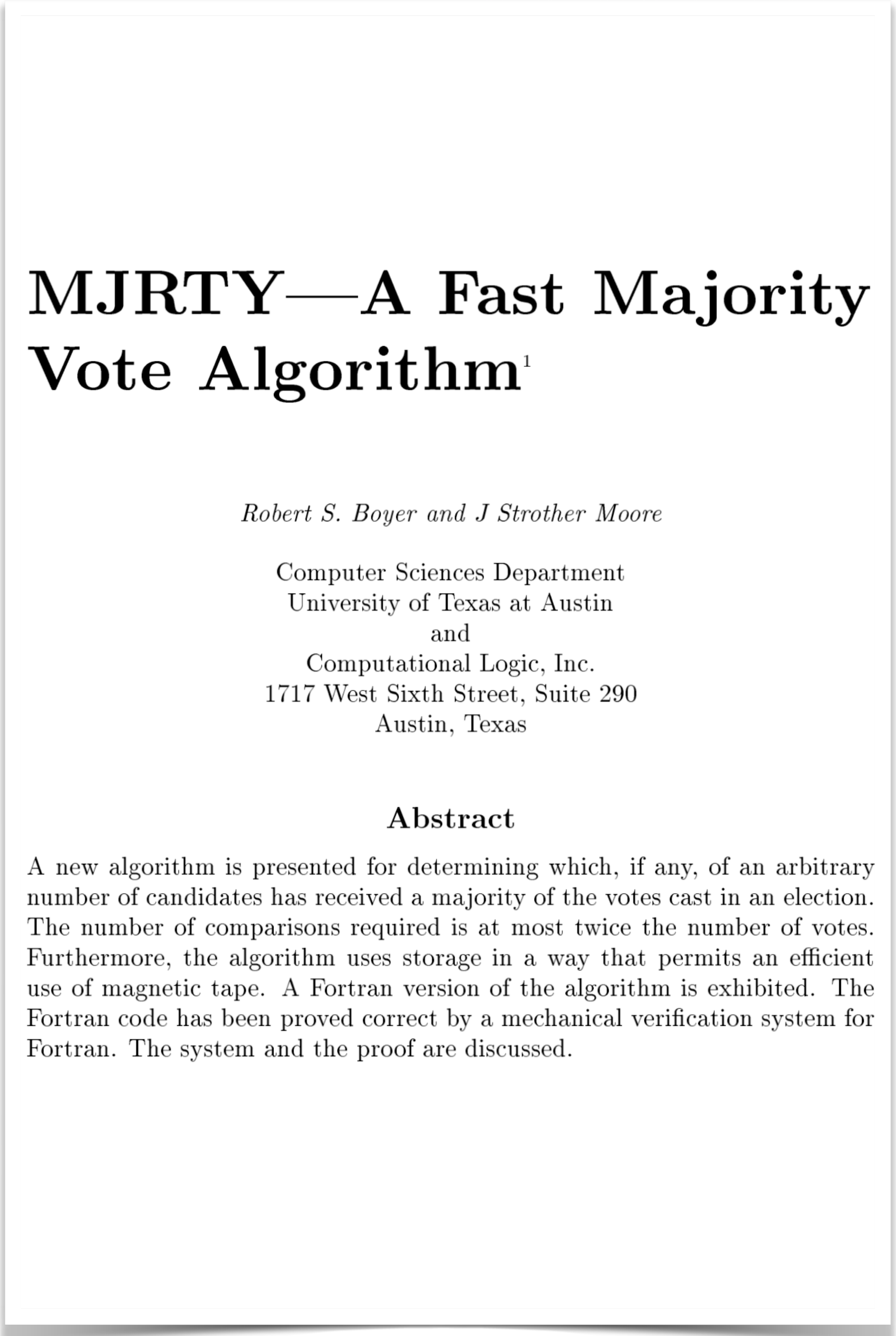
majority = A

delta = 3

A	~~A~~	~~A~~	~~C~~	~~C~~	B	B	C	C	C	B	C	C

majority = A

delta = 1



MJRTY—A Fast Majority Vote Algorithm[1]

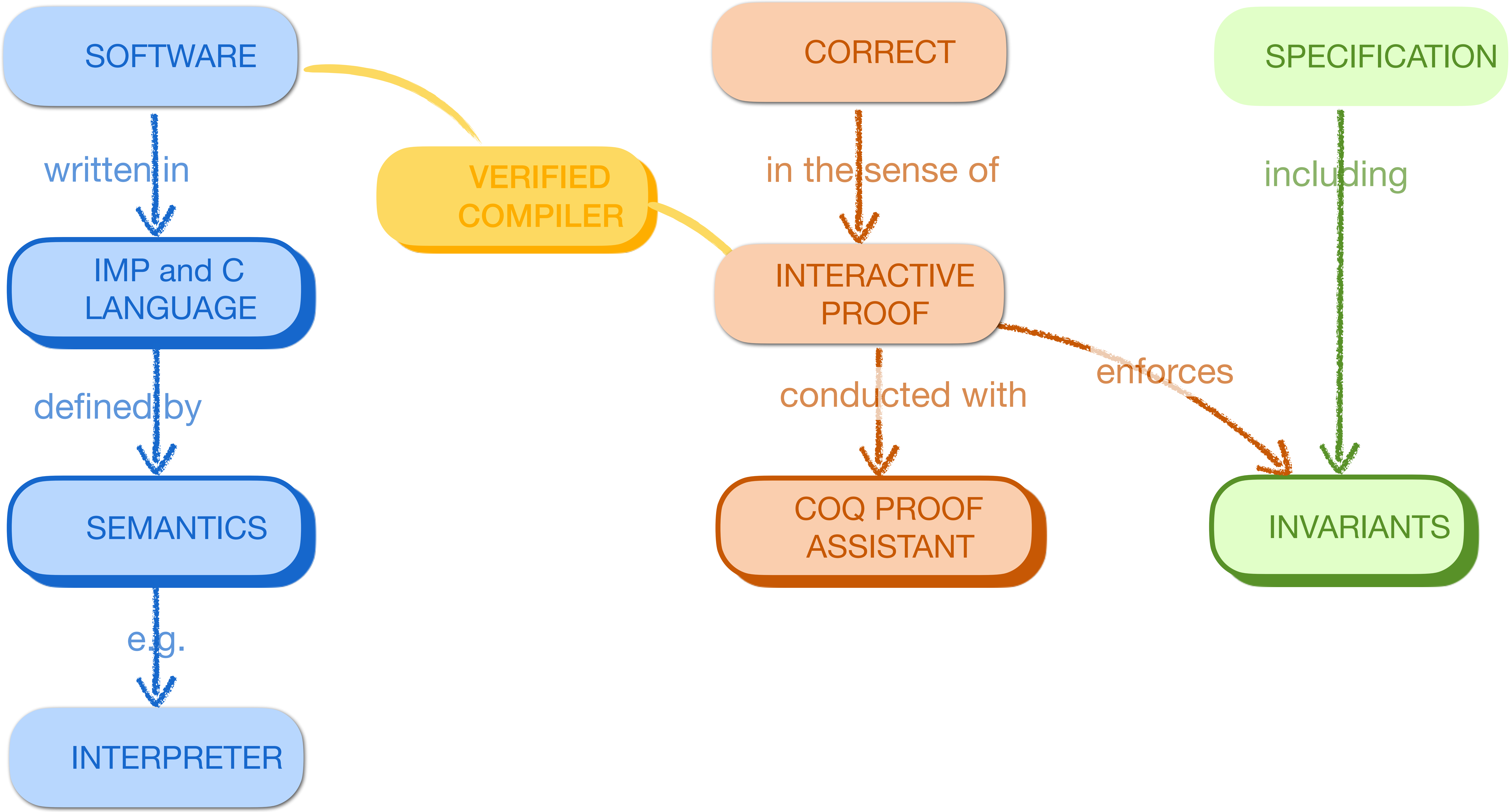
Robert S. Boyer and J Strother Moore

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

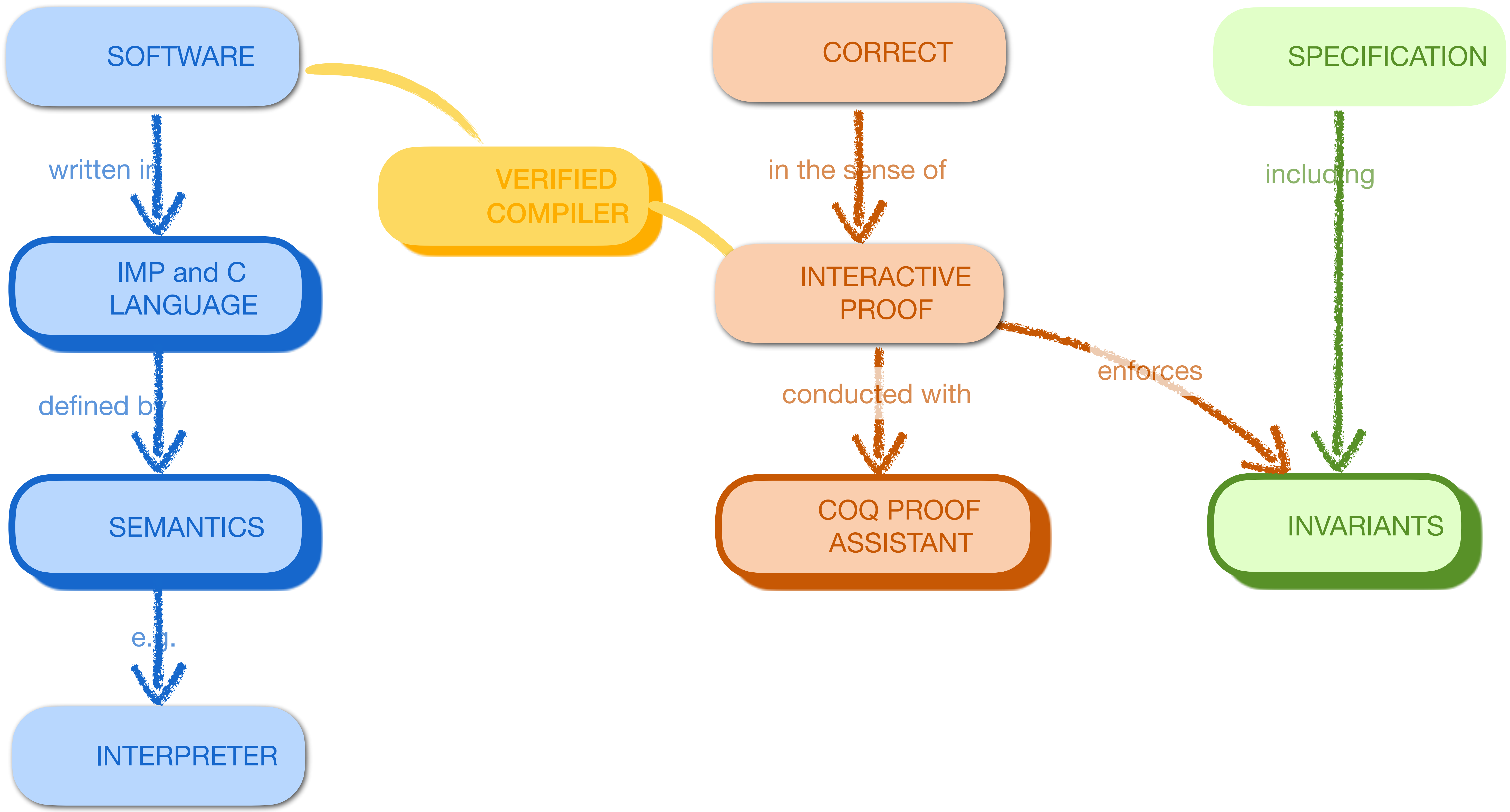
Abstract

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
Early intuitions



SOFTWARE

written in

IMP and C LANGUAGE

defined by

SEMANTICS

e.g.

INTERPRETER

VERIFIED COMPILER

CORRECT

in the sense of

INTERACTIVE PROOF

conducted with

COQ PROOF ASSISTANT

enforces

SPECIFICATION

including

INVARIANTS

Verified compilation

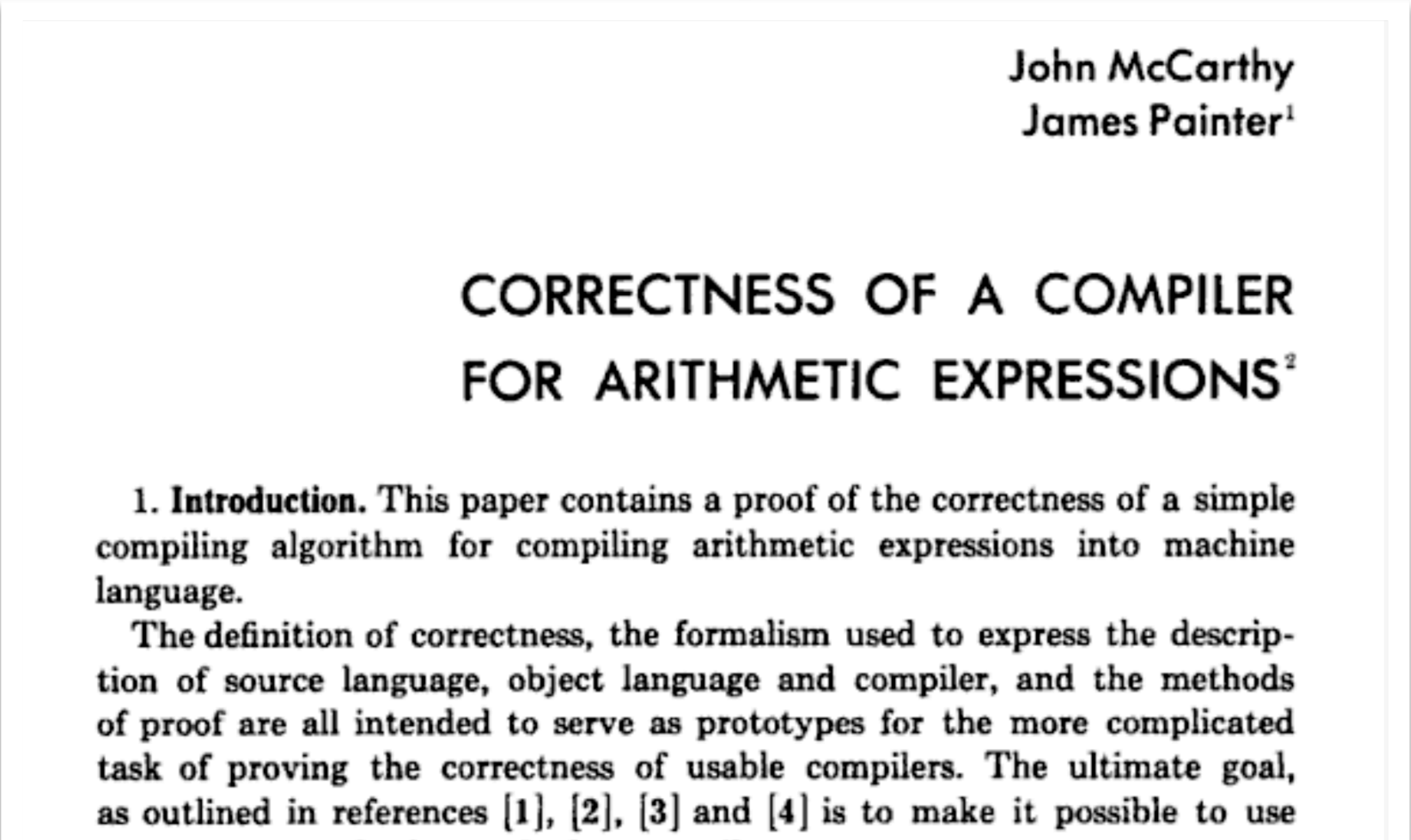
Compilers are complicated programs, but have a rather simple end-to-end specification:

> 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 …



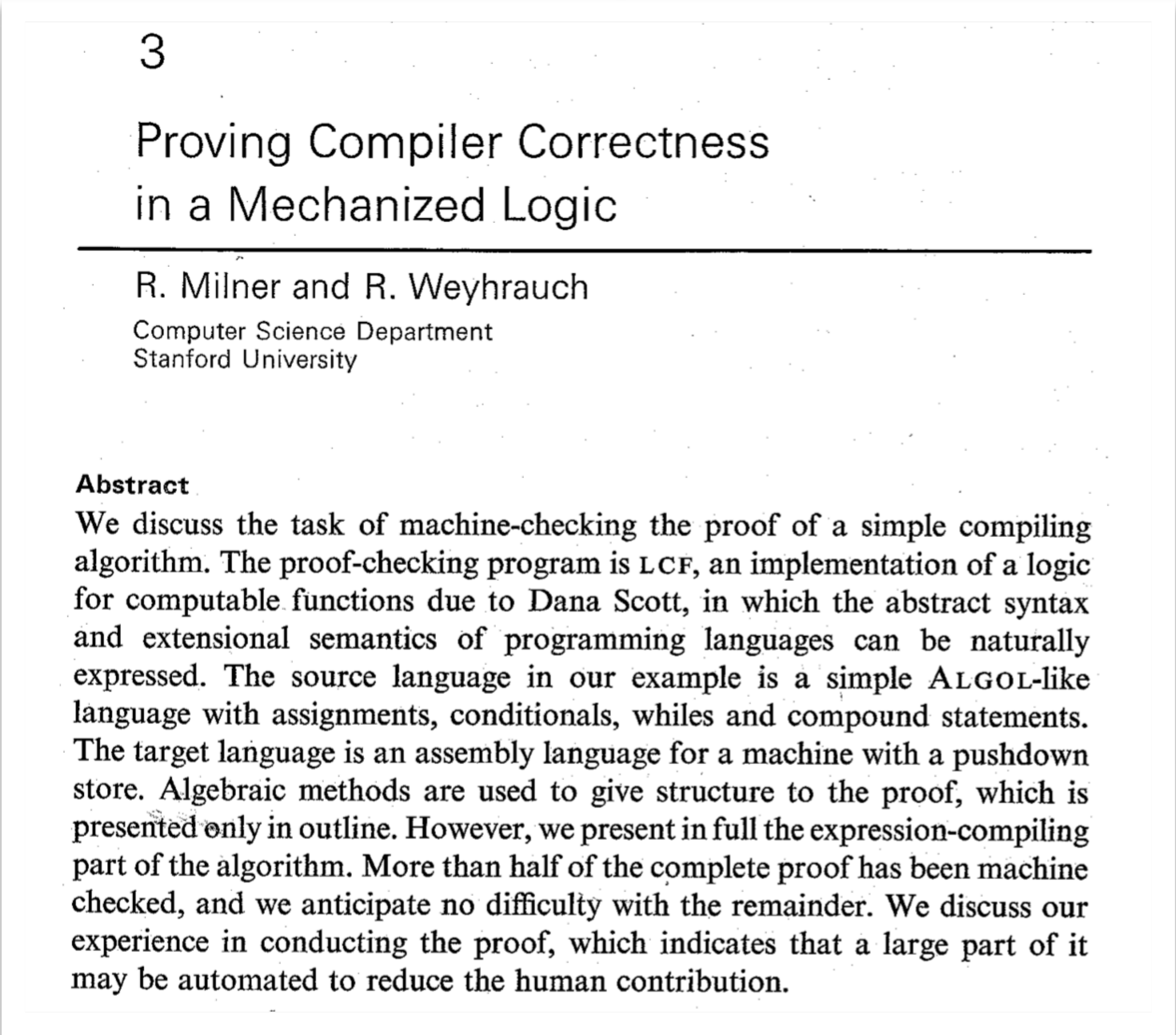
John McCarthy
James Painter[1]

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 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 int | Id string | Plus exp exp
```

```
type state = string → int
```

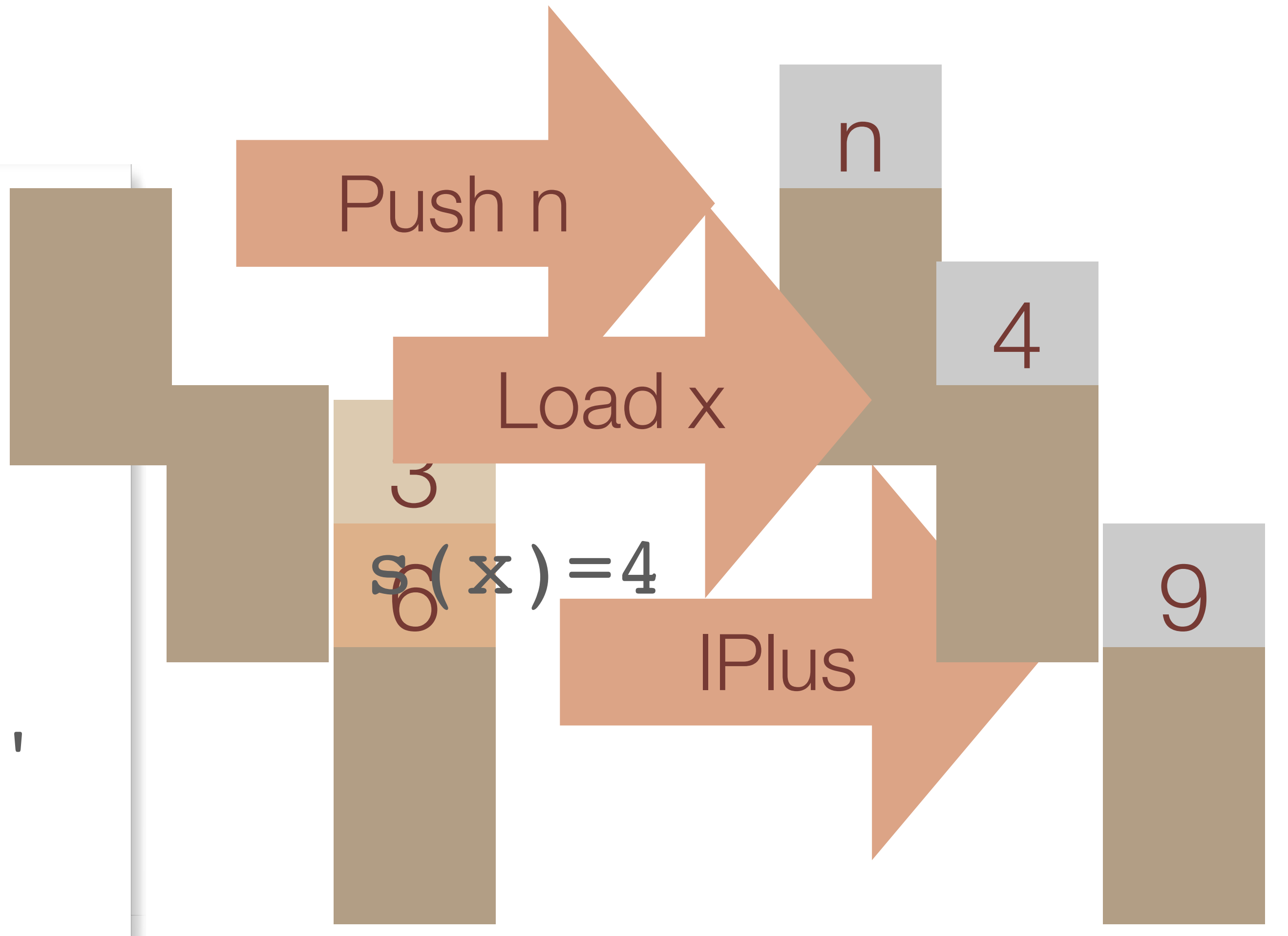
```
let rec eval (s:state)(a:exp): int =
match a with
   | Nb n → n
   | Id x → s x
   | Plus (a1,a2) → (eval s a1)+(eval s
```

compilation

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

```
type instr = Push int | Load string  | IPlus
```

```
let rec exec(s:state)(stack: int list)(pgm: instr list): int list =
  match (pgm, stack) with
  | ([], _) → stack
  | (Push n :: pgm', _) → exec s (n :: stack) pgm'
  | (Load x :: pgm', _) → exec s (s x :: stack) pgm'
  | (IPlus :: pgm', n:: m :: stack') → exec s ((m+n) :: stack') pgm'
  | (_ :: pgm', _) → exec s 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 s a,
  exec s [] (compile a) = [eval s a].
```

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

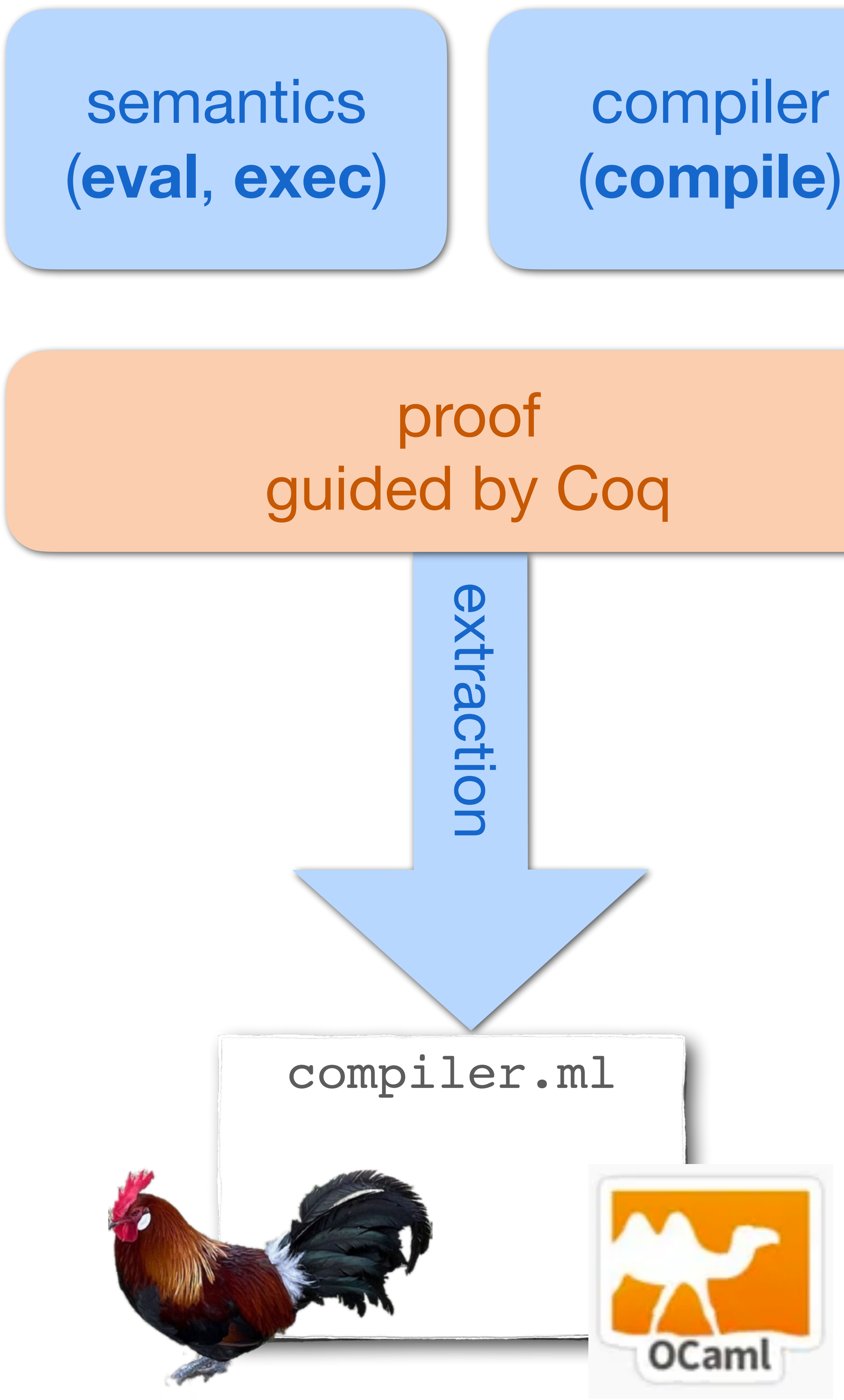
compiler
(**compile**)

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 s a,
  exec s [] (compile a) = [eval s a].
Proof.
  intros;
  … (* not shown here *)
Qed.
```

```
Extraction compile.
```



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

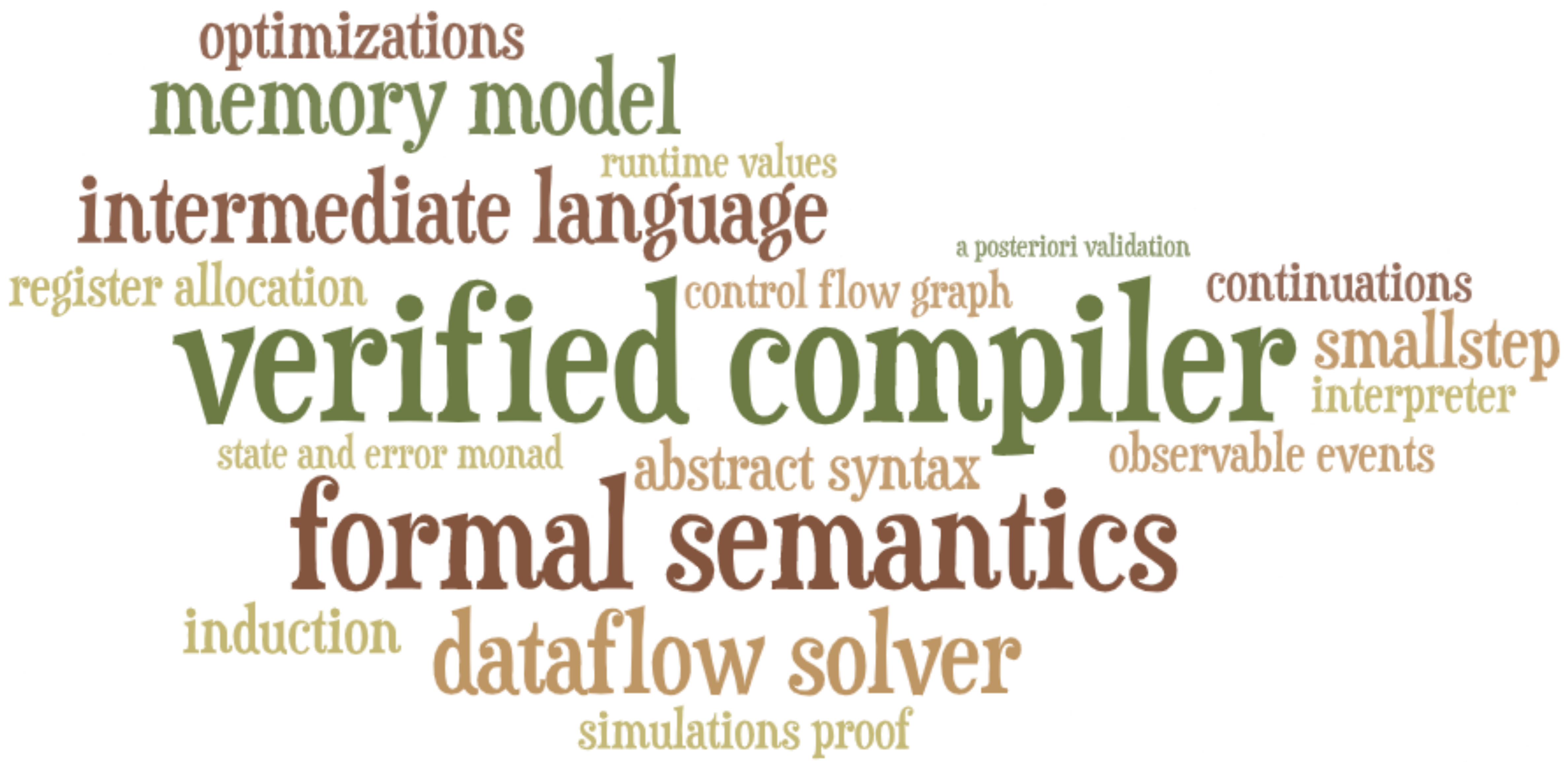
compiler
(**compile**)

proof
guided by Coq

extraction

compiler.ml

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



A selection of formally verified compilers

CompCert C compiler (Coq) [Leroy, POPL'06]

CakeML ML bootsrapped compiler (HOL)
[Kumar, Myreen, Norrish, Owens, POPL'14]

CertiCoq Gallina compiler (Coq) [Appel et al., CoqPL'17]

Jasmin language and compiler for cryptographic implementations (Coq)
[Almeida et.al, CCS'17]

The CompCert formally verified compiler

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

A moderately optimizing C compiler

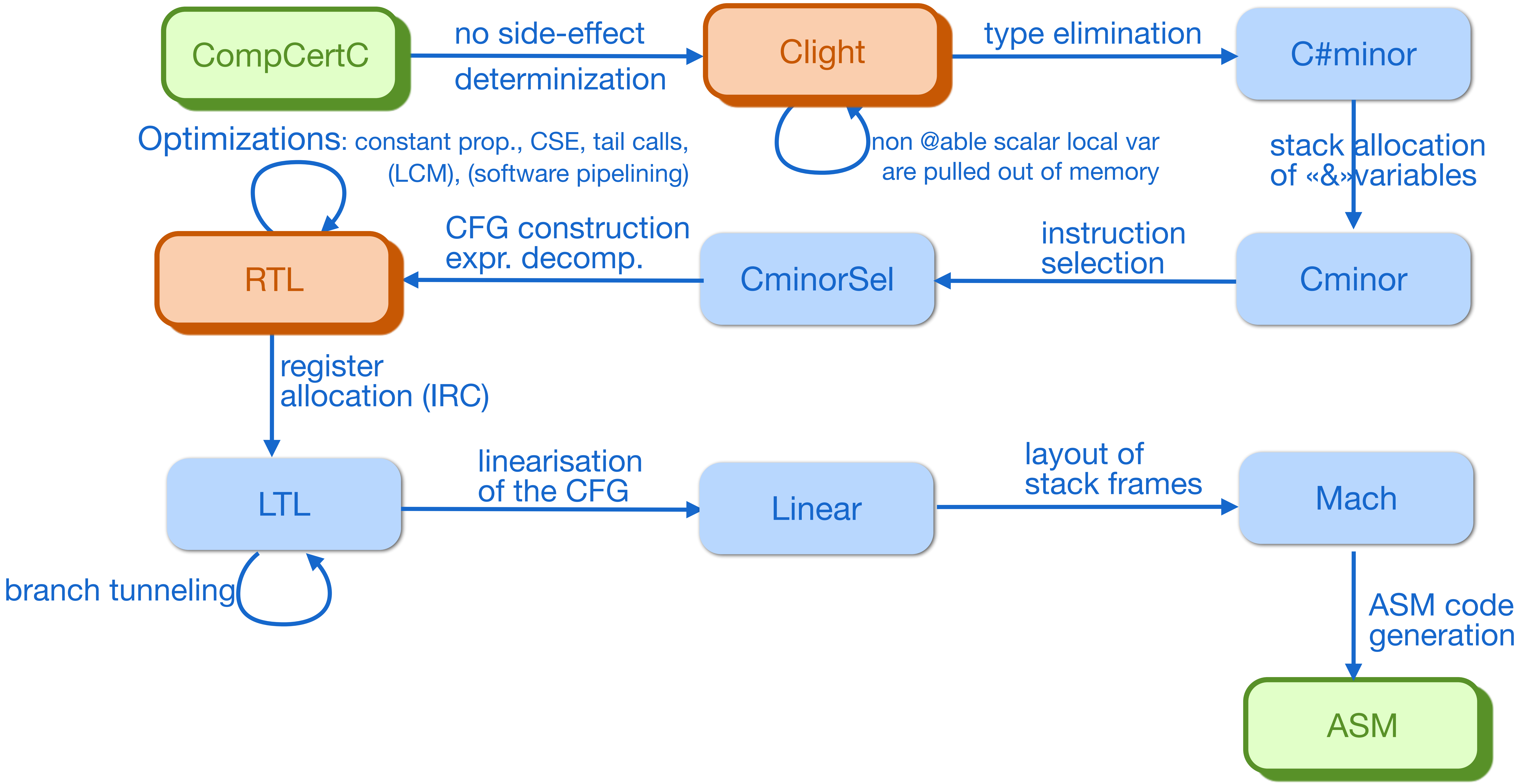
Targets several architectures (PowerPC, ARM, RISC-V and x86)

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: 10 languages, 18 passes



CompCertC

no side-effect

determinization

Clight

type elimination

C#minor

non @able scalar local var
are pulled out of memory

stack allocation
of «&» variables

Cminor

instruction
selection

CminorSel

CFG construction
expr. decomp.

RTL

Optimizations: constant prop., CSE, tail calls,
(LCM), (software pipelining)

register
allocation (IRC)

LTL

branch tunneling

linearisation
of the CFG

Linear

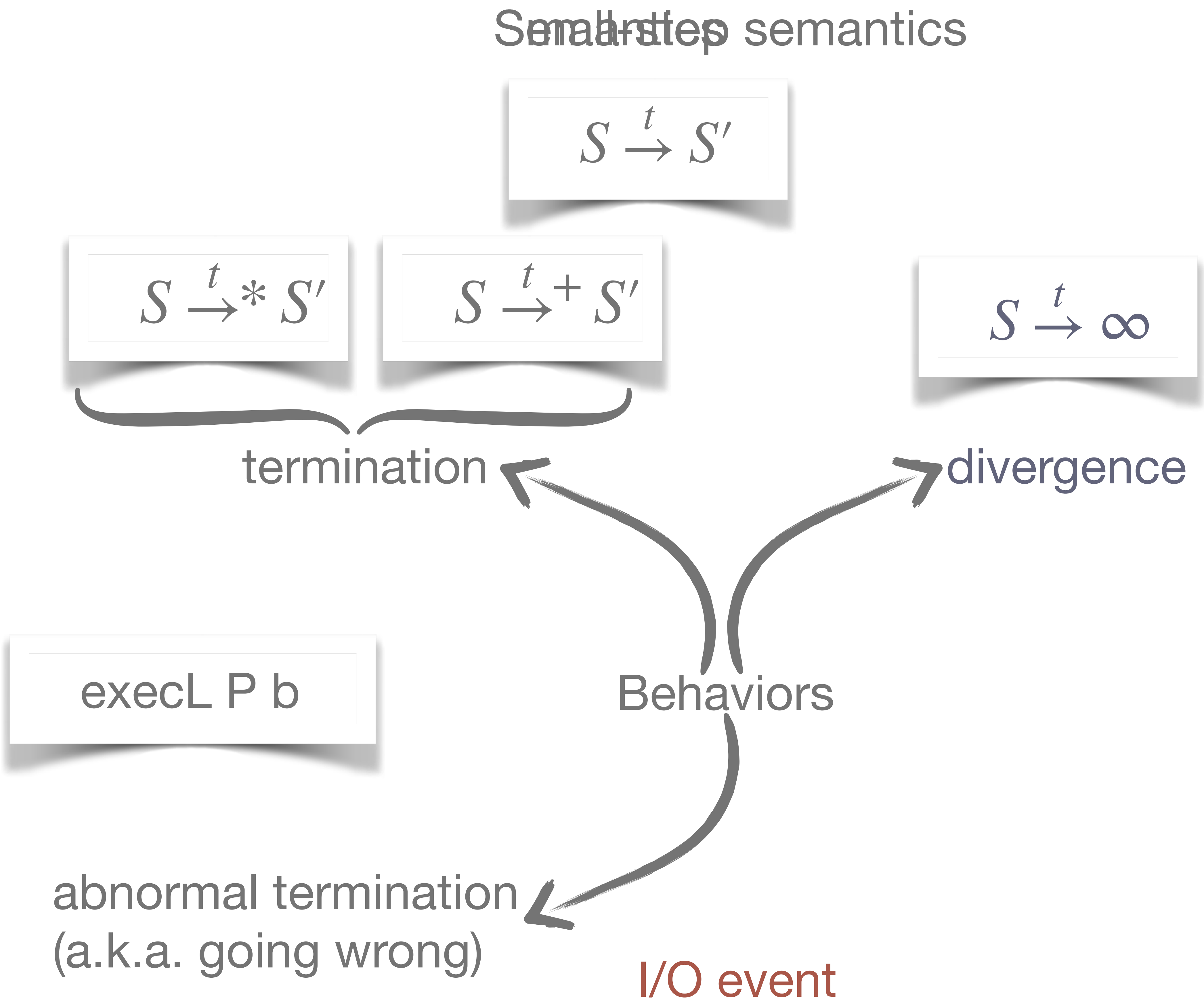
layout of
stack frames

Mach

ASM code
generation

ASM

CompCert compiler: 10 languages, 18 passes



Semantics

Small-step semantics

$S \xrightarrow{t} S'$

$S \xrightarrow{t}{}^{*} S'$

$S \xrightarrow{t}{}^{+} S'$

$S \xrightarrow{t} \infty$

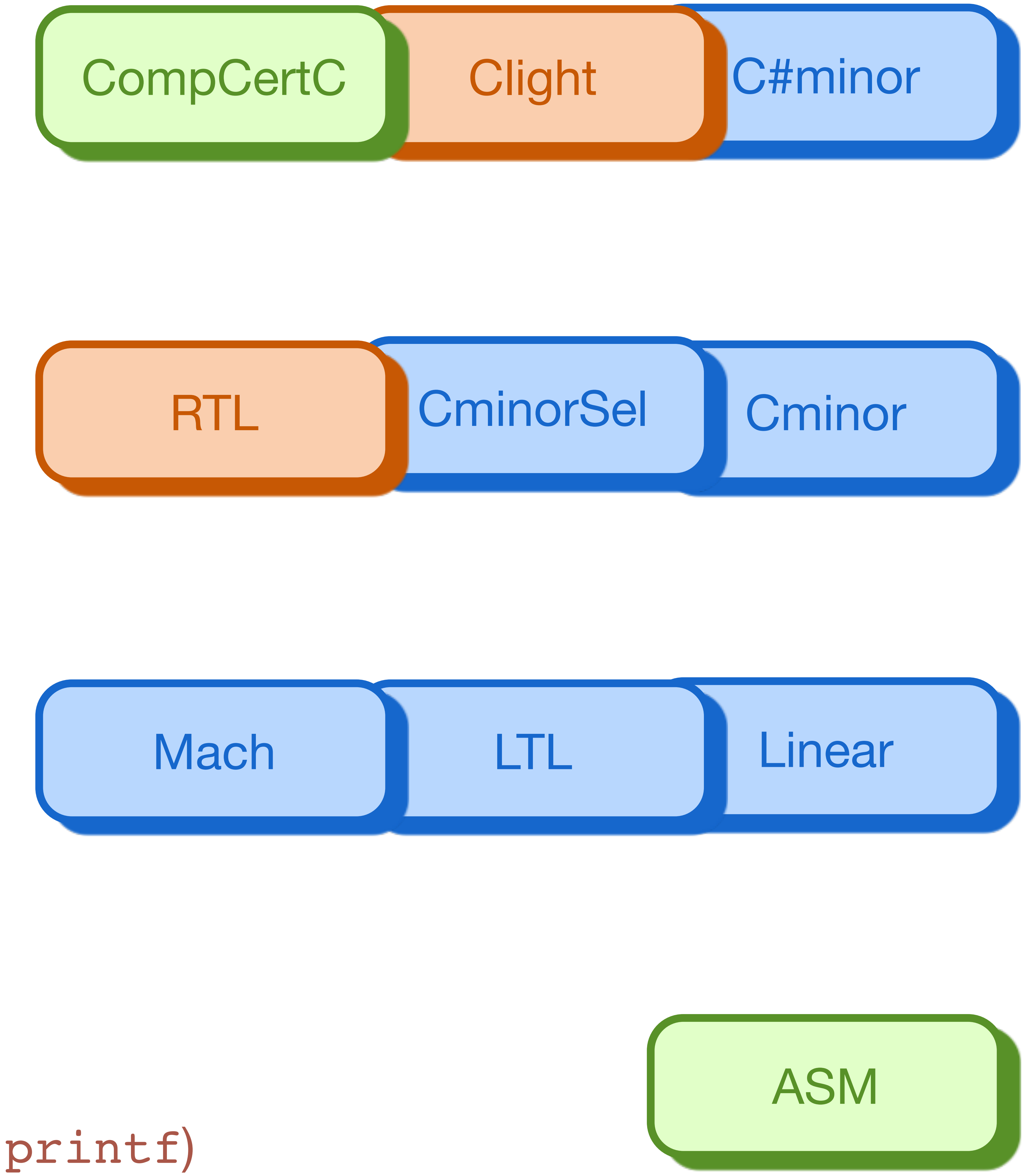
termination

divergence

execL P b

Behaviors

abnormal termination
(a.k.a. going wrong)



CompCertC

Clight

C#minor

RTL

CminorSel

Cminor

Mach

LTL

Linear

ASM

I/O event

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

Proving semantics preservation: the simulation approach

semantics
(**execSource**, **execTarget**)

compiler

Preserved behaviors = termination and divergence

```
Theorem compiler-correct:
  ∀ S C b,
  compiler S = OK C →
  execSource S b →
  execTarget C b.
```

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

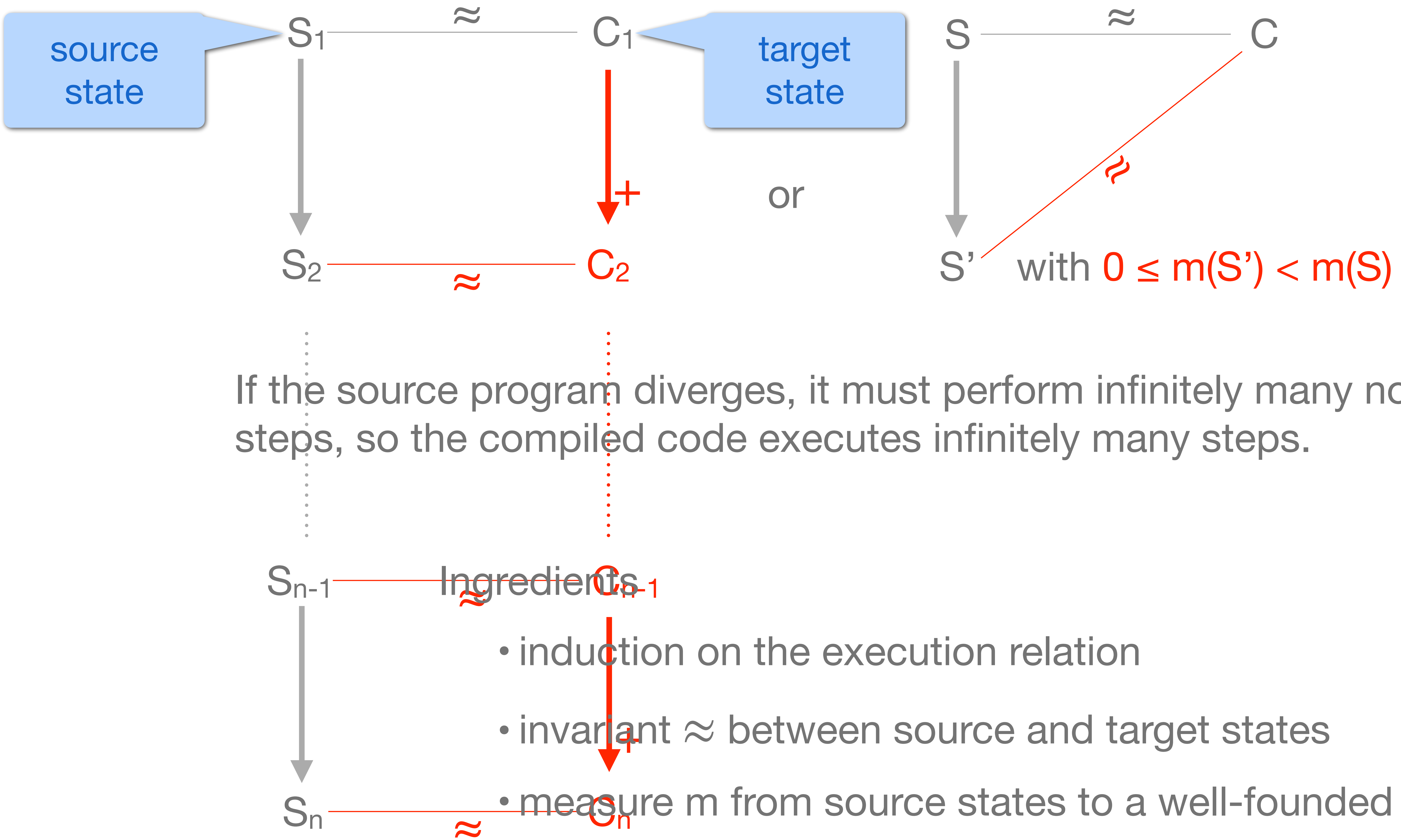
Proof technique: simulation diagram

source state

target state

$$\begin{tikzcd}
S_1 \arrow[r, dash, "\approx"] \arrow[d, "t_1"'] & C_1 \arrow[d, "t_1"', "+"] \\
S_2 \arrow[r, dash, "\approx"'] & C_2
\end{tikzcd}$$

Proving semantics preservation: the simulation approach

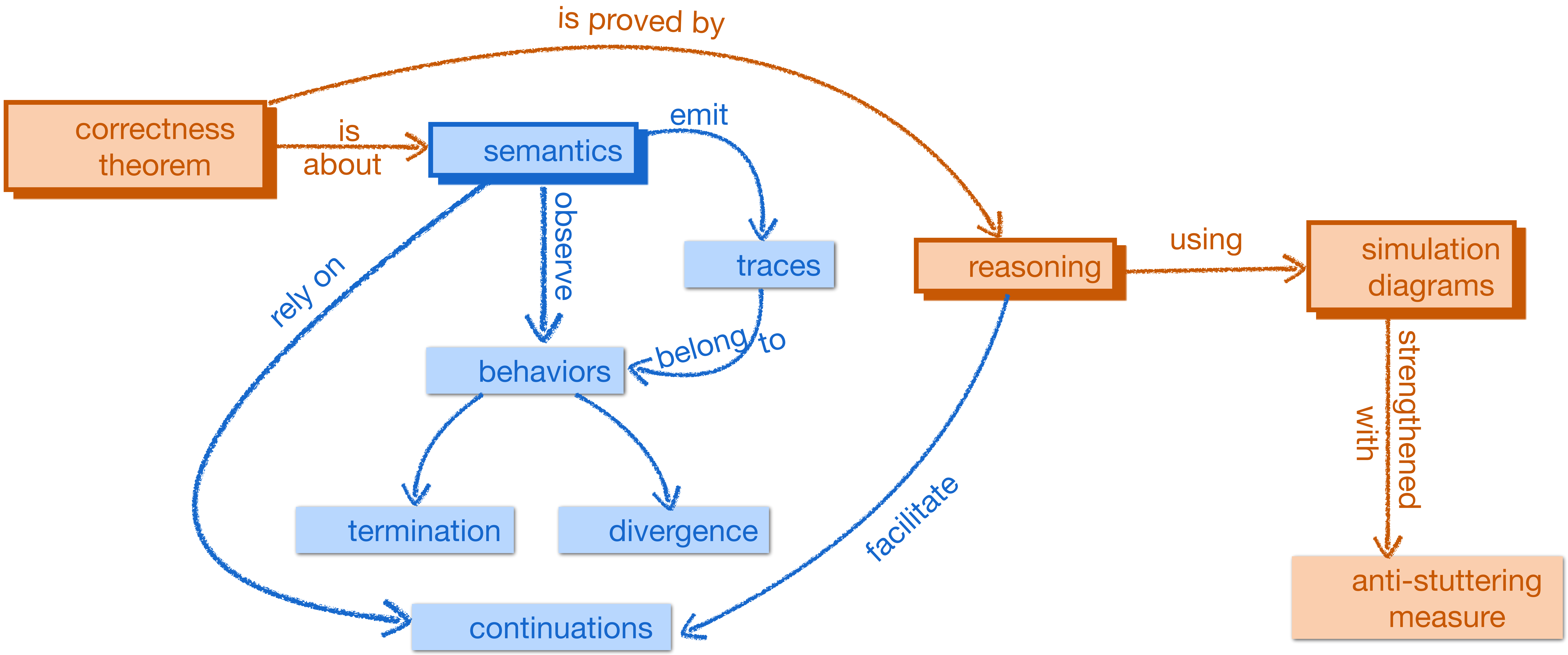


If the source program diverges, it must perform infinitely many non-stuttering steps, so the compiled code executes infinitely many steps.

Ingredients:

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

Semantic reasoning for compiler correctness: summary



correctness theorem

is about

semantics

is proved by

emit

observe

traces

rely on

reasoning

using

simulation diagrams

behaviors

belong to

strengthened with

termination

divergence

facilitate

anti-stuttering measure

continuations

CompCert verified compiler: main ingredients

Compilation

Source and target languages

Intermediate language

Optimizations

Data-flow analysis

Register allocation

Other passes

Formal semantics

Observable behaviors

Traces of ext. I/O events

Small-step style

Continuations

Memory model

Deductive verification

Proof assistant

Semantic preservation theorem

Simulation diagram

Anti-stuttering measure

A posteriori validation

Part 4

Beyond CompCert:

- secure compilation
- just-in-time compilation

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?

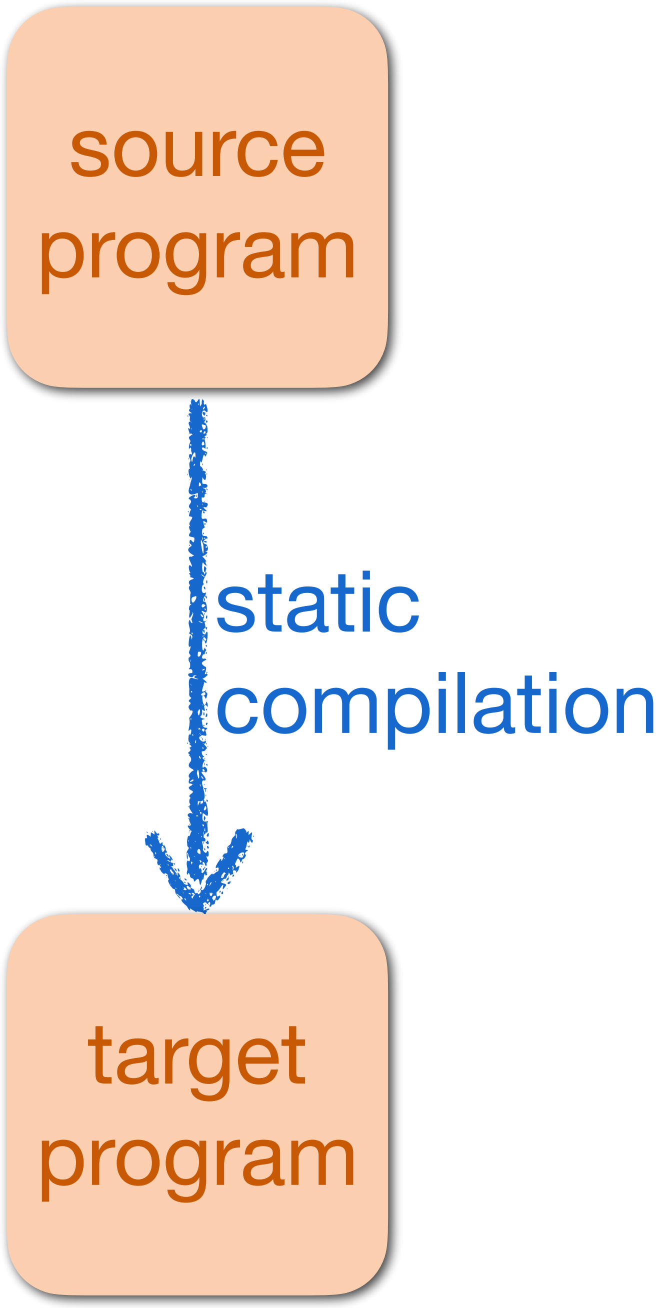
```
Theorem compiler-correct:
  ∀ S C b,
  compiler S = OK C →
  execCompCertC S b →
  execASM C b.
```

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

observe
program leakages (boolean guards
and memory accesses)

2 executions of S from 2
indistinguishable states (only share
public values)

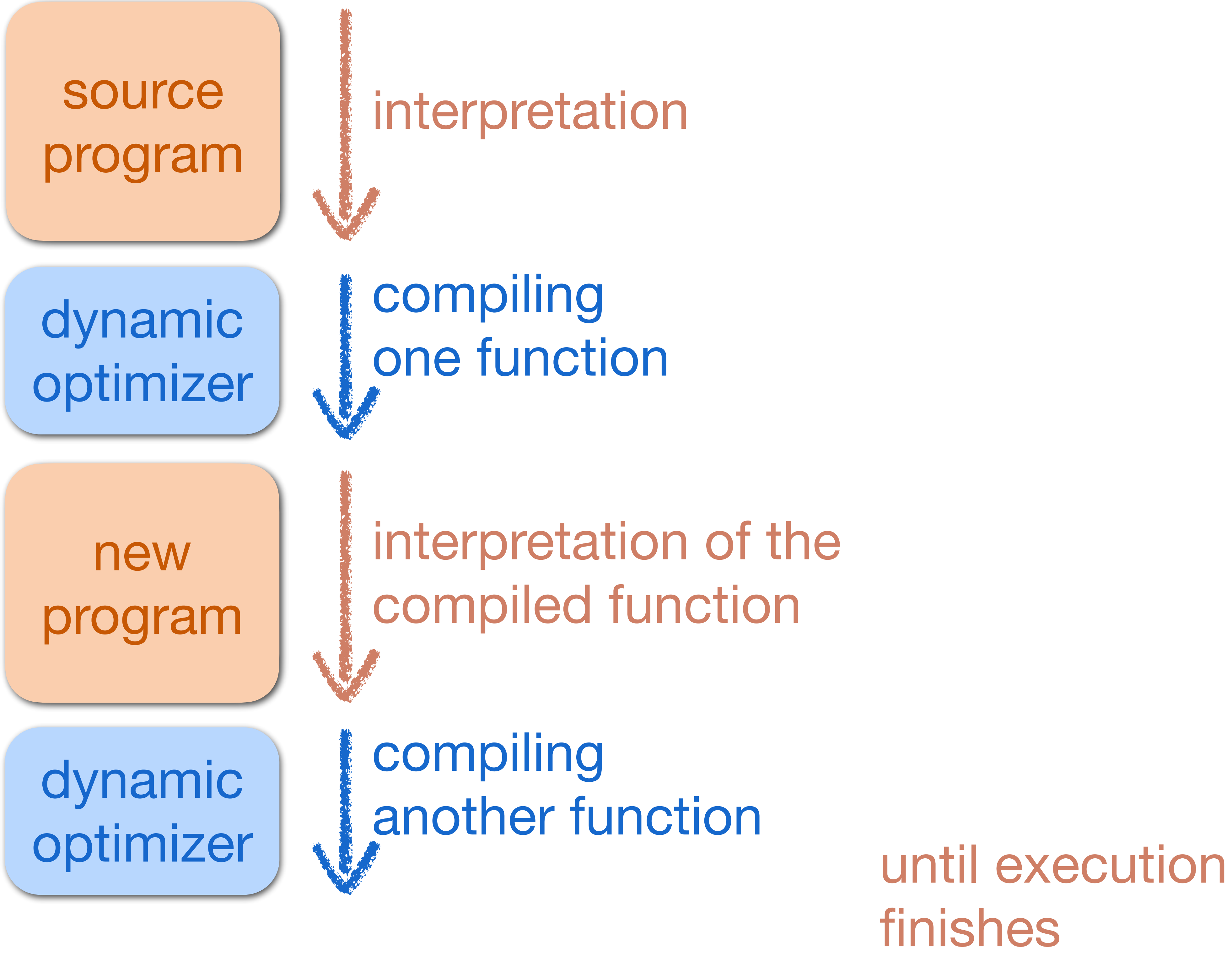
Just-in-time (JIT) compilation vs. static compilation


source program

static compilation

target program

A JIT interleaves execution and optimization.


source program

interpretation

dynamic optimizer

compiling one function

new program

interpretation of the compiled function

dynamic optimizer

compiling another function

until execution finishes

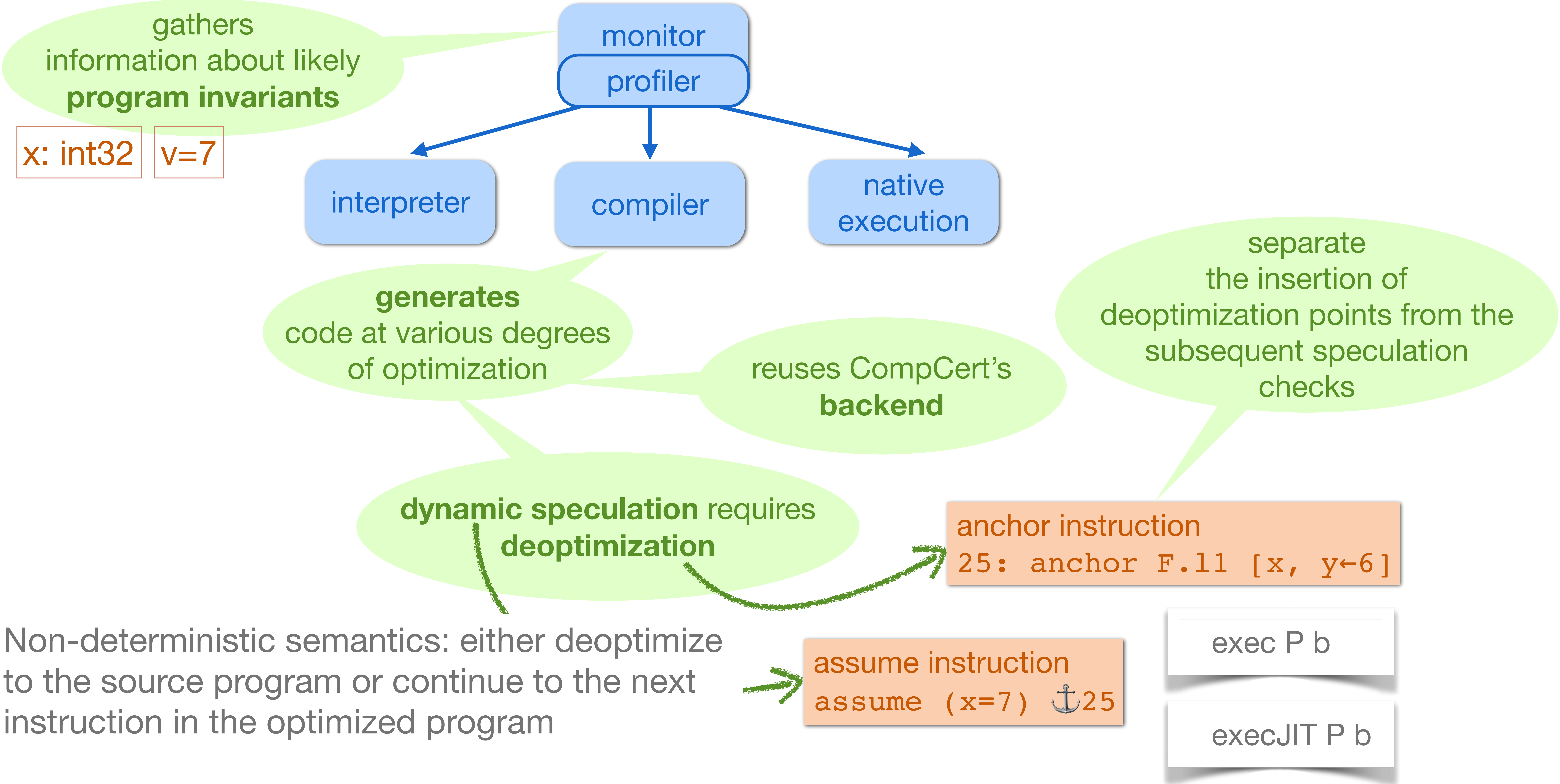
Dynamic speculation generates specialized functions

Deoptimization requires the JIT to synthesize interpreter stackframes in the middle of a function

Verifying just-in-time (JIT) compilation: FM JIT

[Aurèle Barrière's PhD 12/2022]

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



gathers information about likely **program invariants**

x: int32 v=7

generates code at various degrees of optimization

reuses CompCert's **backend**

separate the insertion of deoptimization points from the subsequent speculation checks

dynamic speculation requires **deoptimization**

anchor instruction

`25: anchor F.l1 [x, y←6]`

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

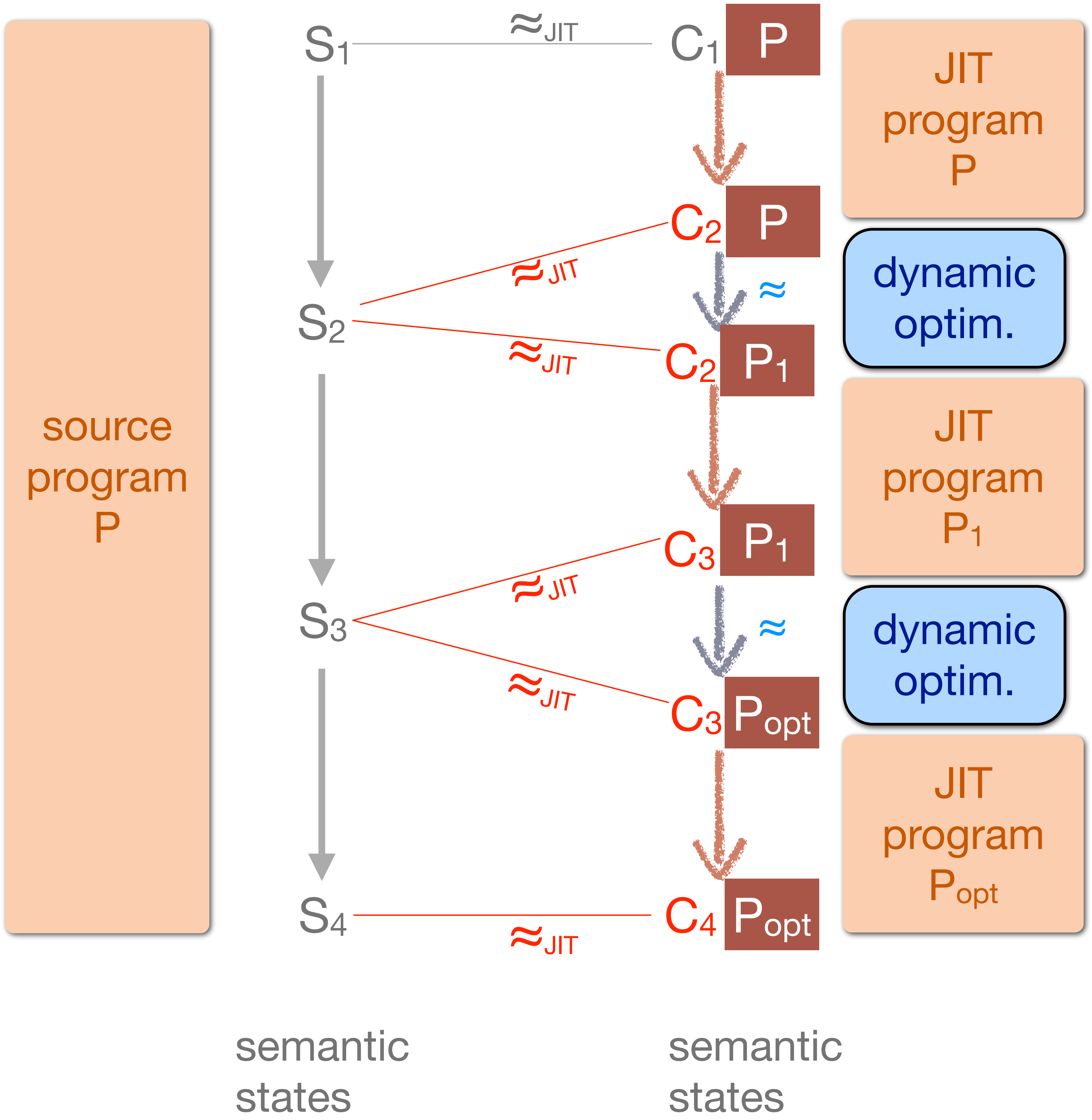
assume instruction

`assume (x=7) ⚓25`

exec P b

execJIT P b

Nested simulations for JIT verification



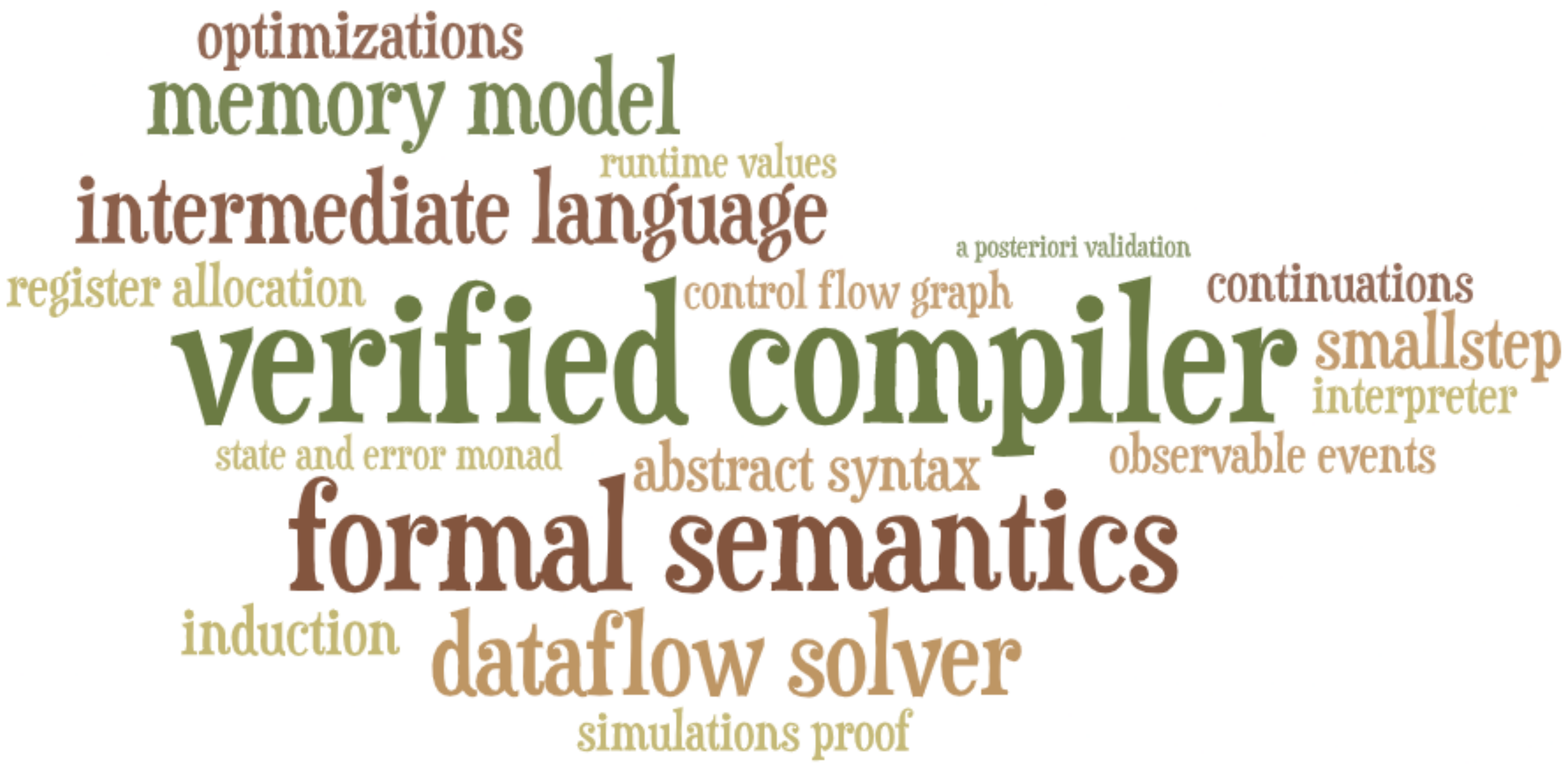
```
Theorem JITcompiler-correct:
  ∀ P Popt b,
  JITcompiler P = Popt →
  exec P b →
  execJIT Popt b.
```

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

- C_i correspond to S_i
- P_i is **equivalent** to P

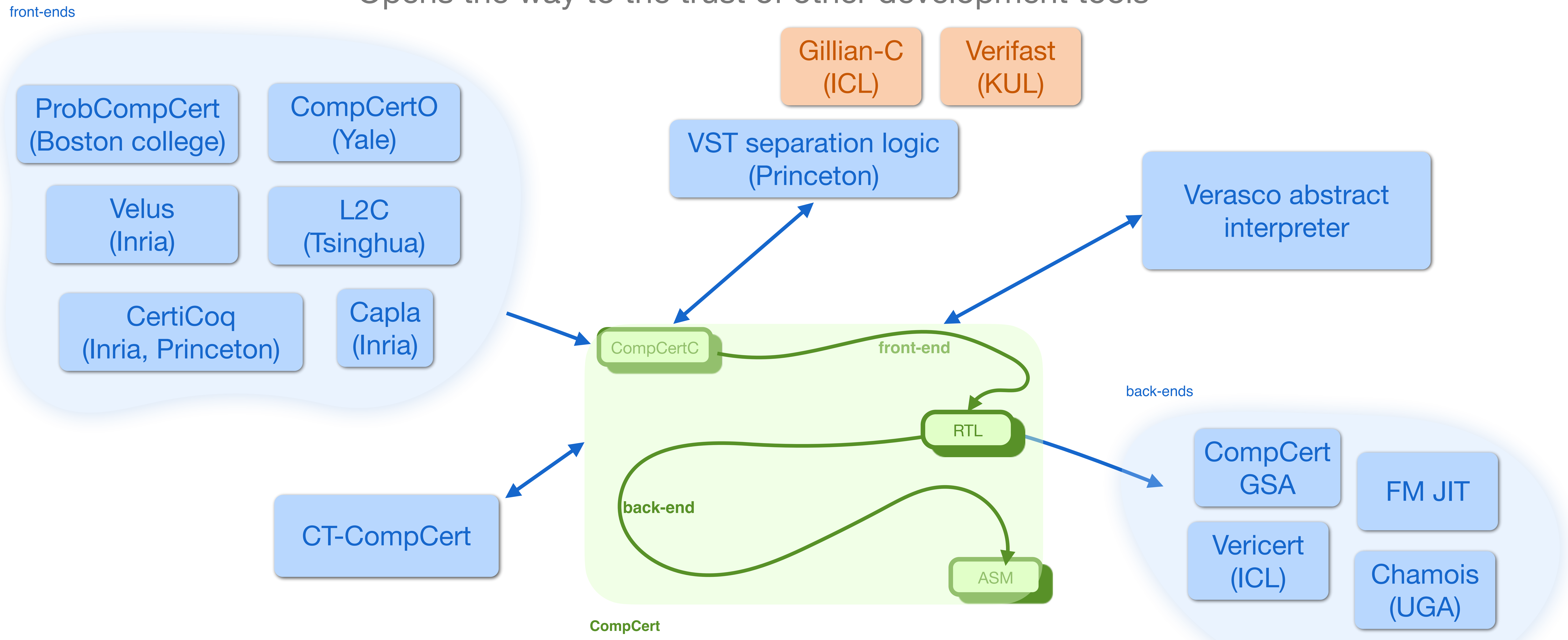
Nested simulation: this **equivalence** is expressed with another (internal) simulation $\approx$ between compiled programs

Conclusion


optimizations
memory model
runtime values
intermediate language
a posteriori validation
register allocation
control flow graph
continuations
verified compiler
smallstep
interpreter
state and error monad
abstract syntax
observable events
formal semantics
induction
dataflow solver
simulations proof

CompCert, an open infrastructure for research

Opens the way to the trust of other development tools



front-ends

ProbCompCert (Boston college)

CompCertO (Yale)

Velus (Inria)

L2C (Tsinghua)

CertiCoq (Inria, Princeton)

Capla (Inria)

Gillian-C (ICL)

Verifast (KUL)

VST separation logic (Princeton)

Verasco abstract interpreter

CompCertC

front-end

RTL

back-end

ASM

CompCert

CT-CompCert

back-ends

CompCert GSA

FM JIT

Vericert (ICL)

Chamois (UGA)

Mechanized semantics are the shared basis for verified compilers, sound program logics, and sound static analyzers

Thank you!

Questions?