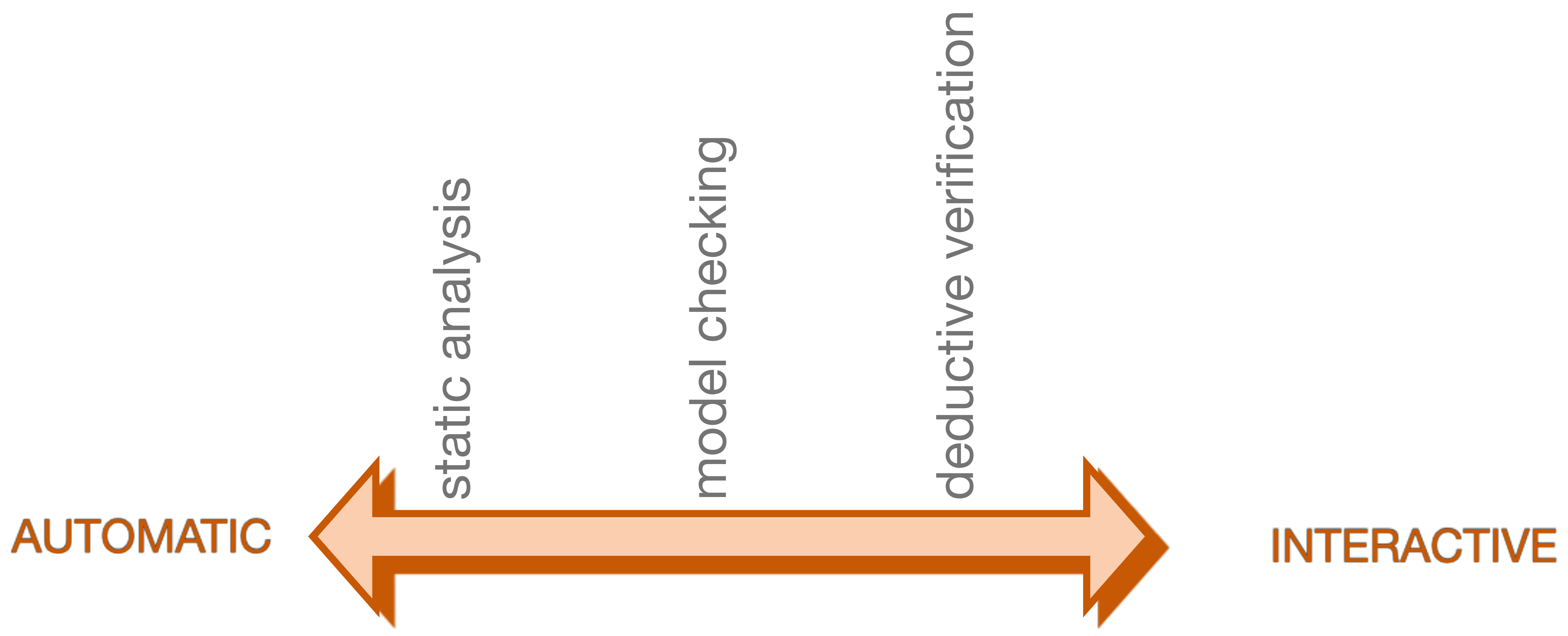
Verified compilation: towards zero-defect software

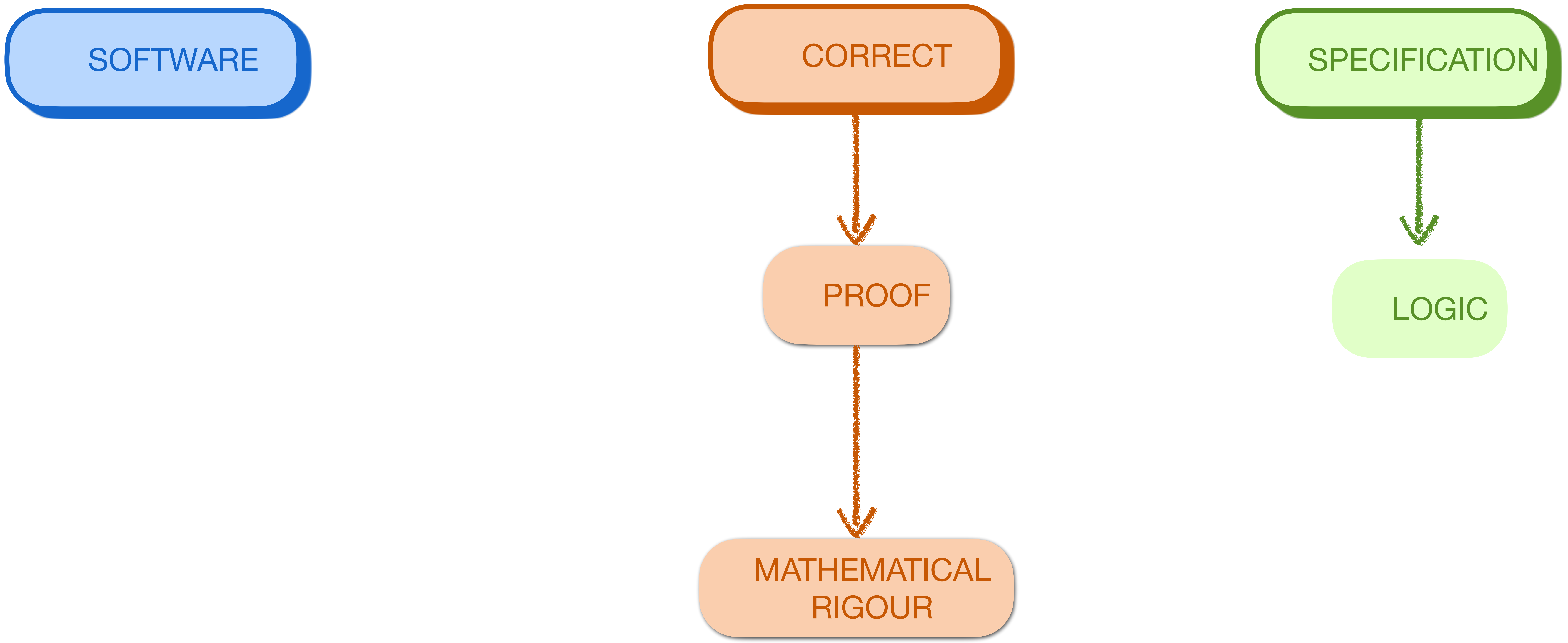
Sandrine Blazy

GDR Sécurité Informatique, Caen, 2025-06-23

Formal verification of software: tool-assisted techniques

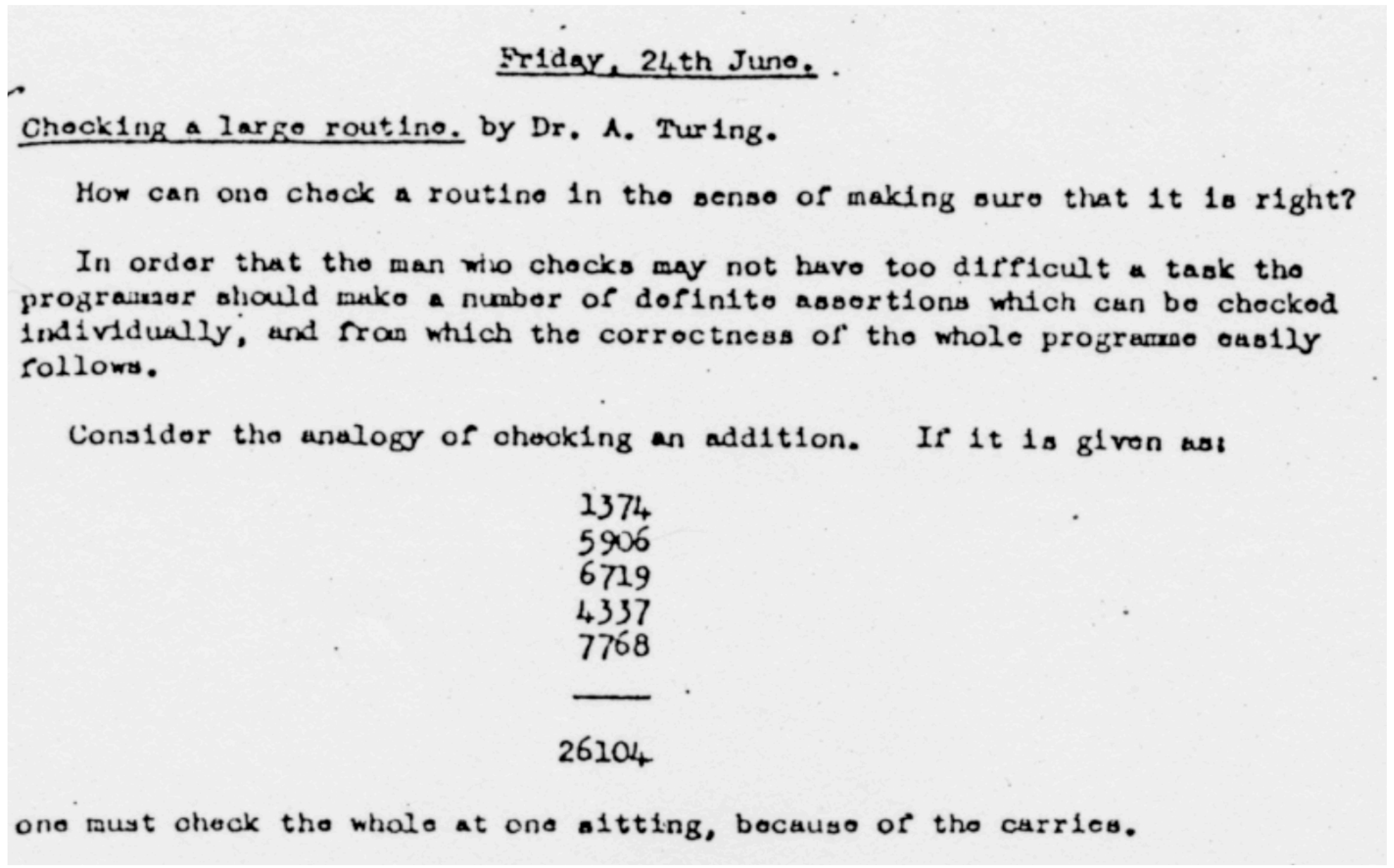


Deductive verification



From early intuitions …

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



Friday, 24th June.

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

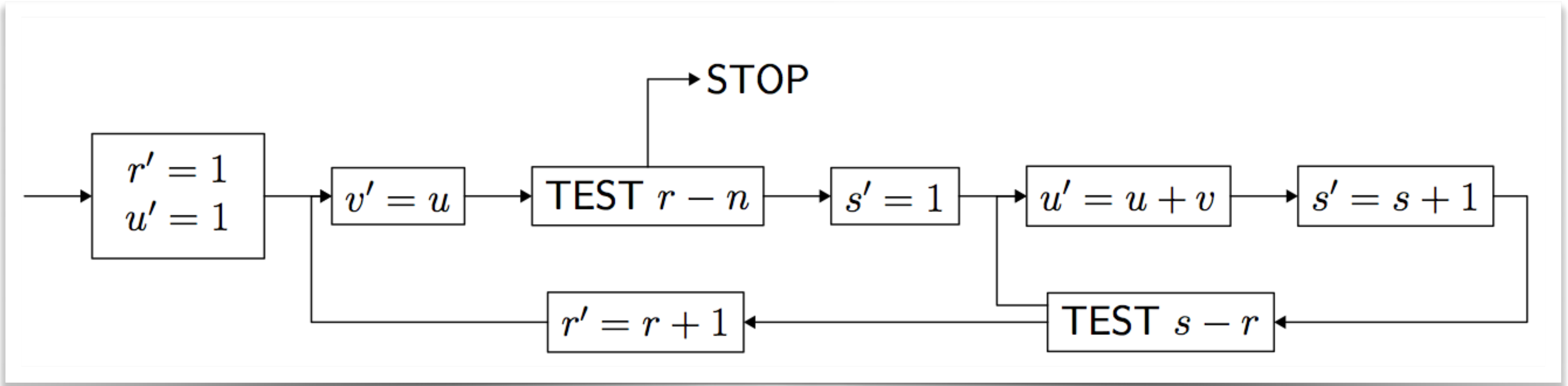
How can one check a routine in the sense of making sure that it is right?

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
```

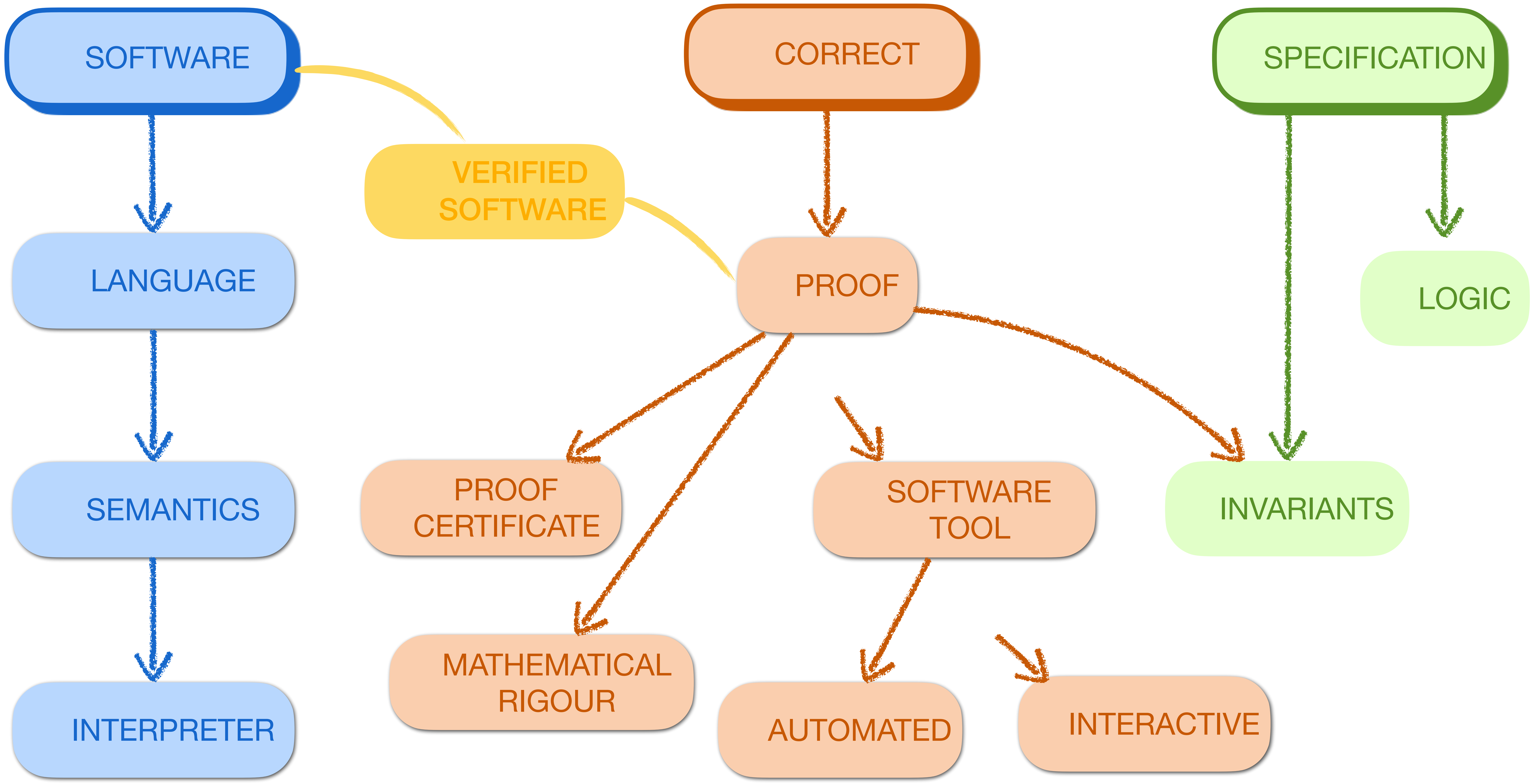
one must check the whole at one sitting, because of the carries.



```
$u \leftarrow 1$
for $r = 0$ to $n - 1$ do
    $v \leftarrow u$
    for $s = 1$ to $r$ do
        $u \leftarrow u + v$
```

… to deductive-verification and automated tools

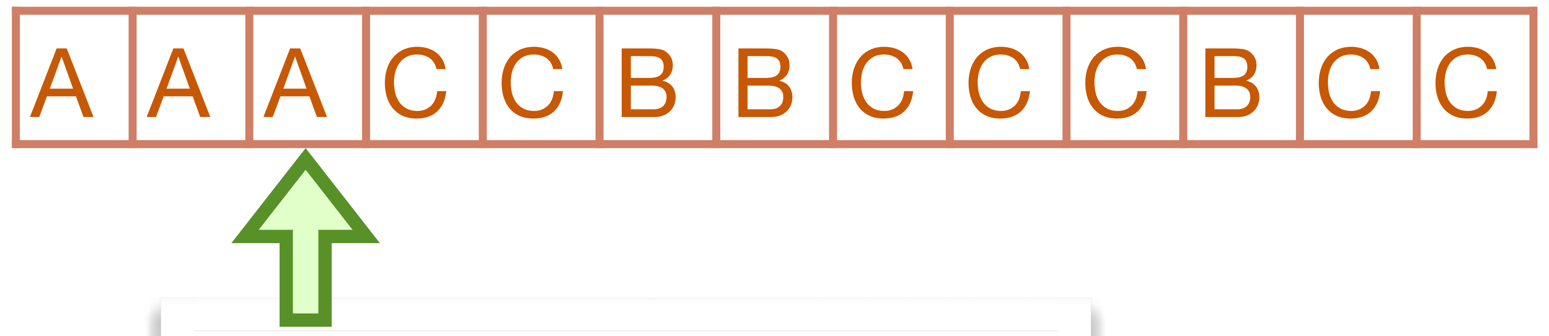
Floyd 1967, Hoare 1969



Another historical example

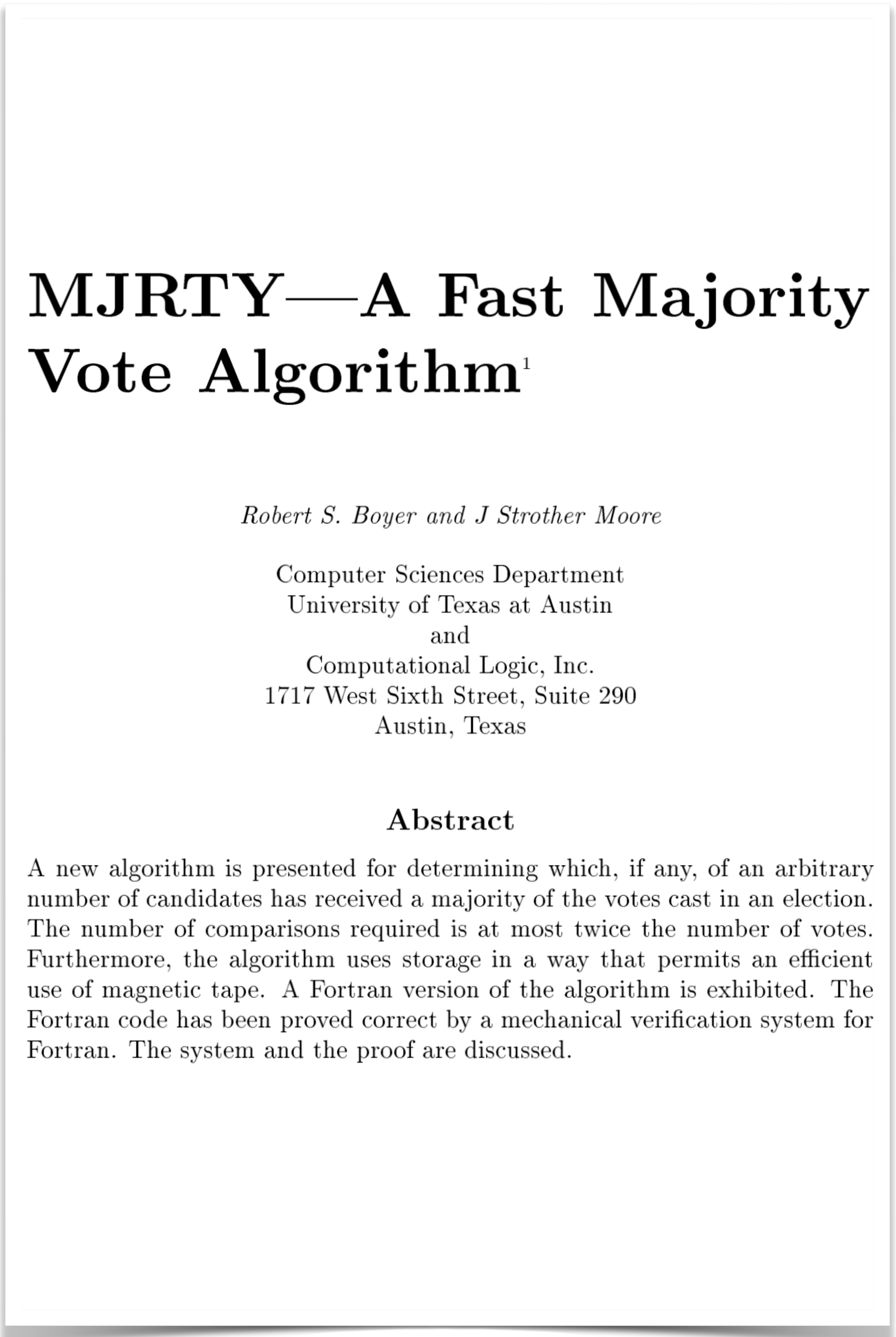
Boyer-Moore's majority. 1980

Given N votes, determine the majority if any



majority = A

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

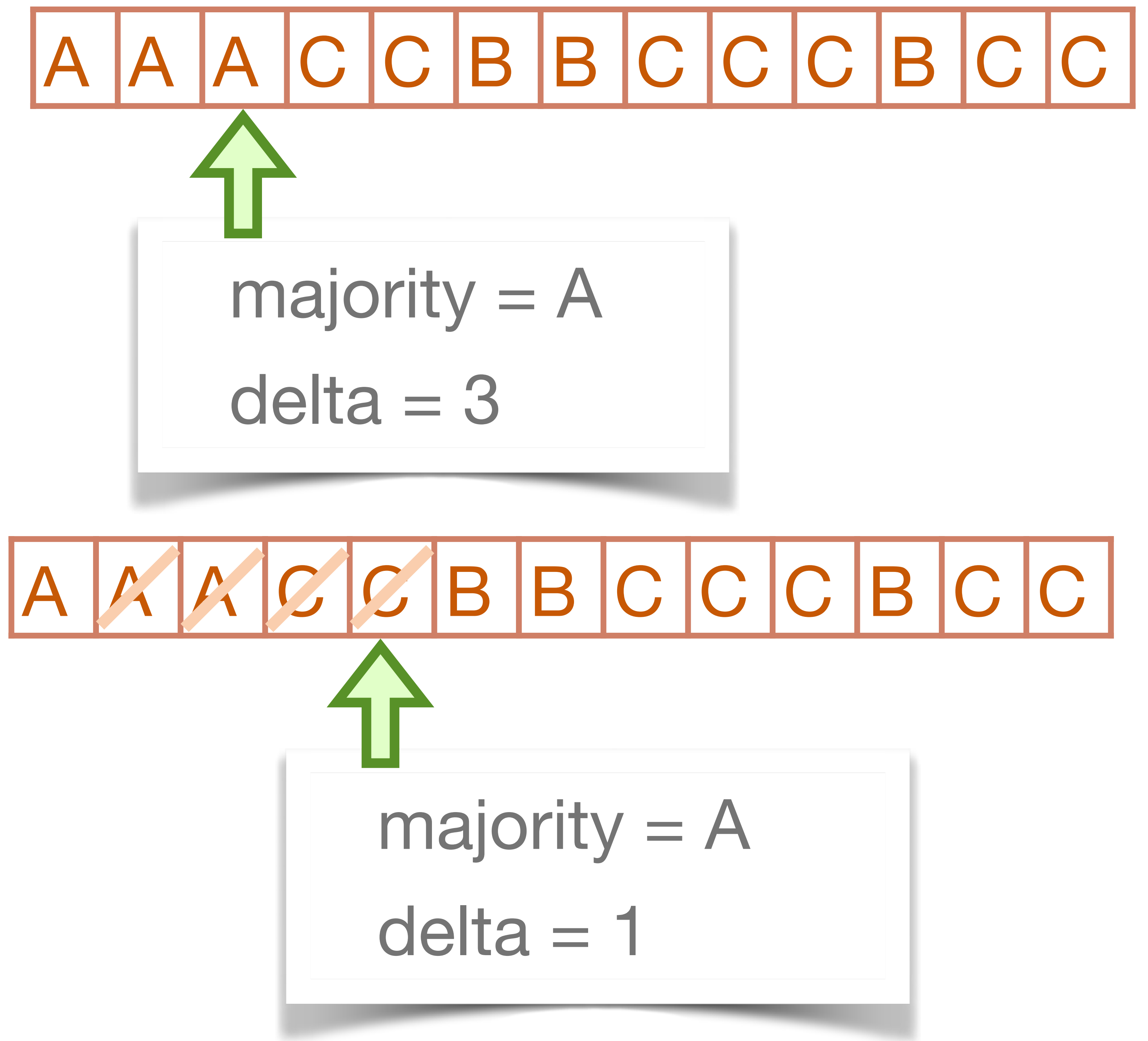
Abstract

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.

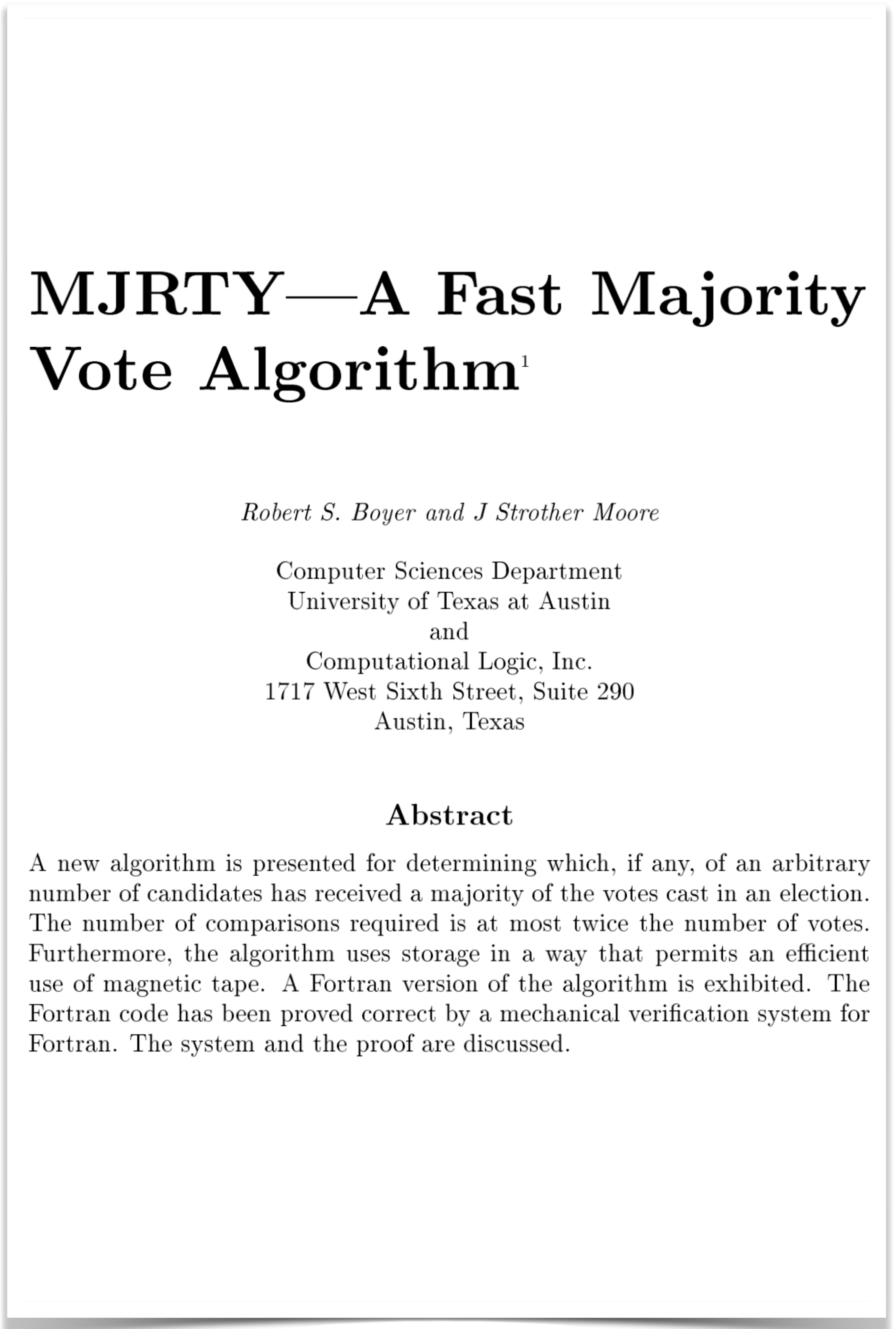
Another historical example

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]

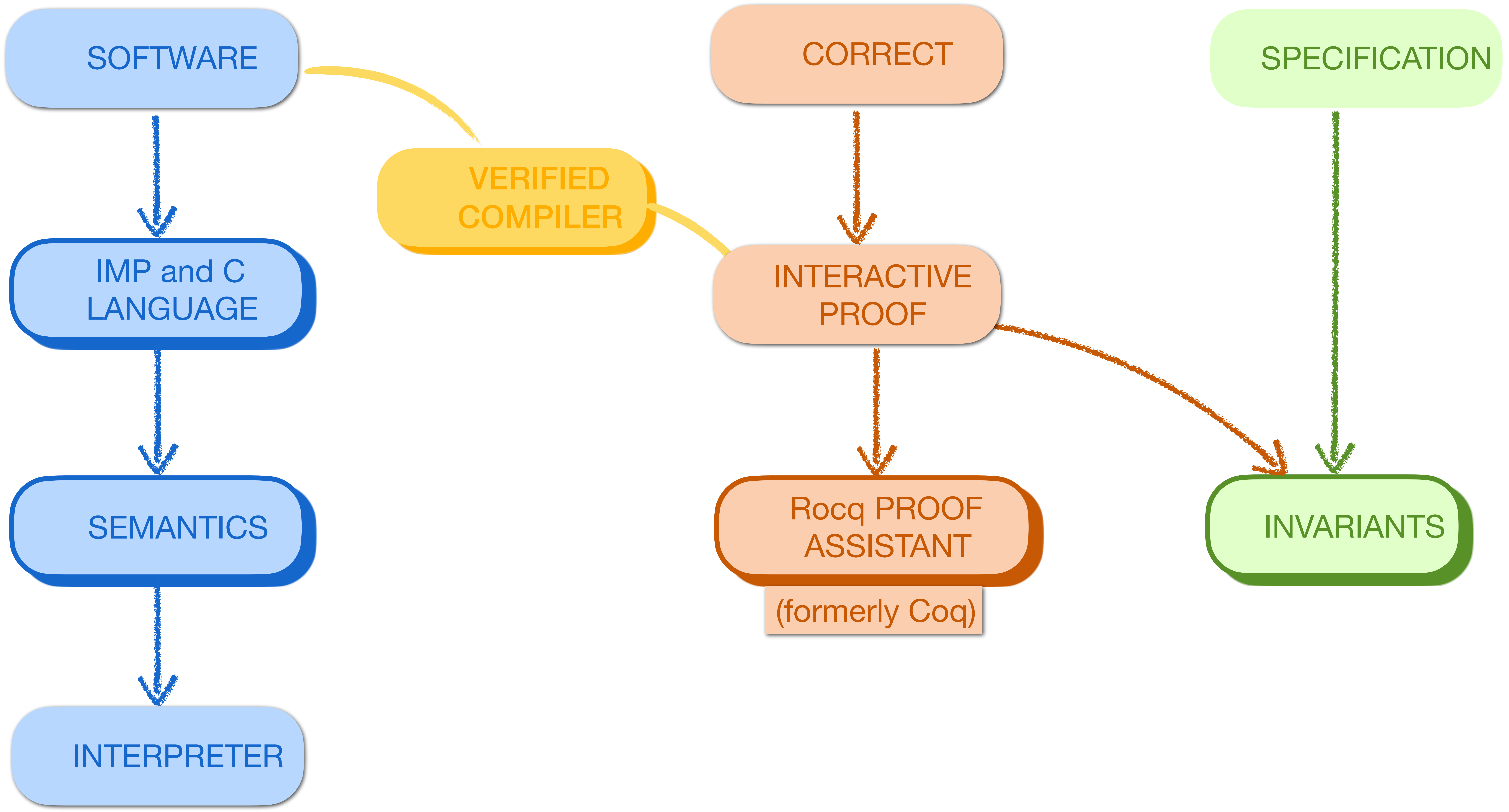
Robert S. Boyer and J Strother Moore

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

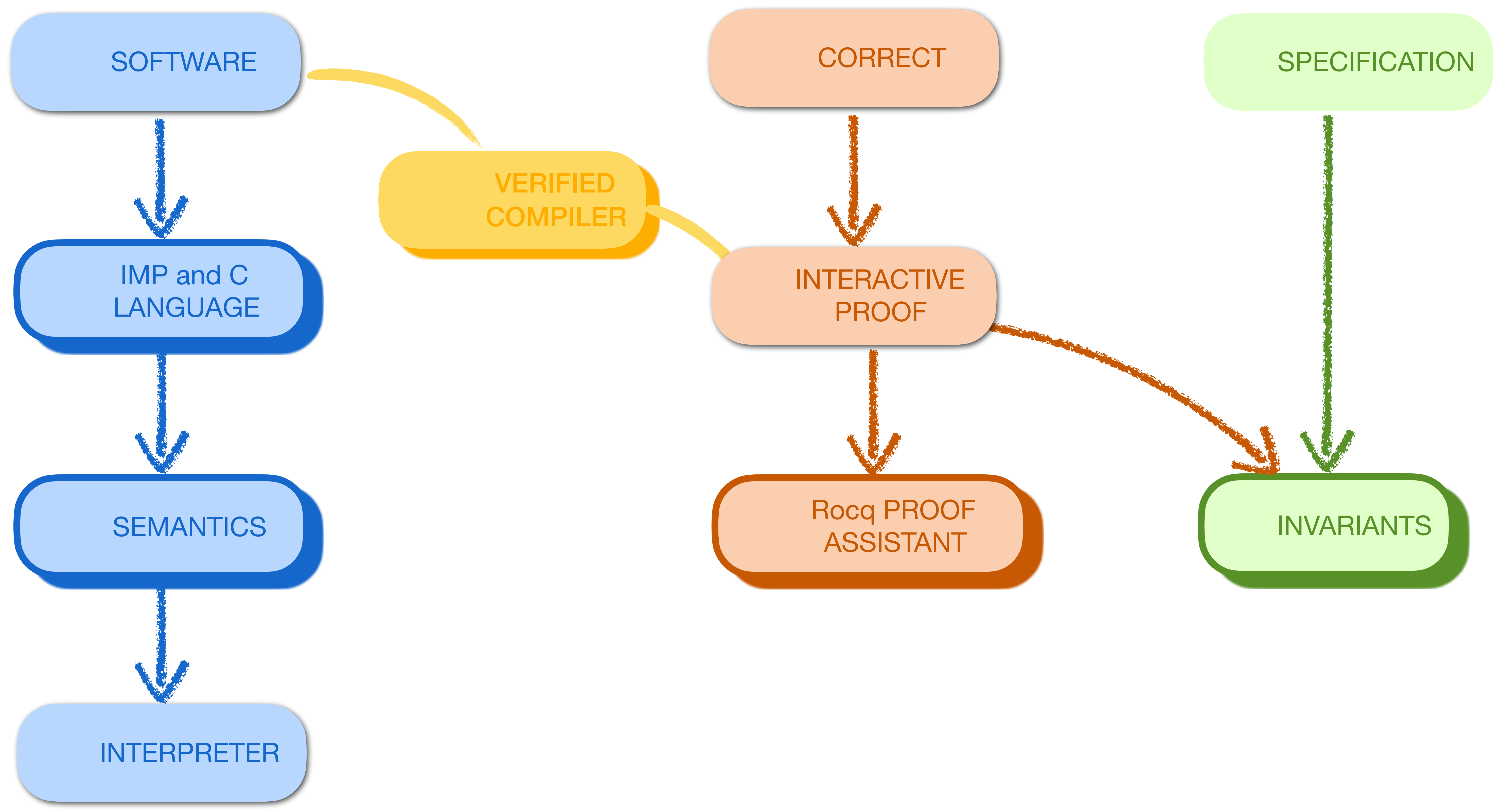
Abstract

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



Verified compilation

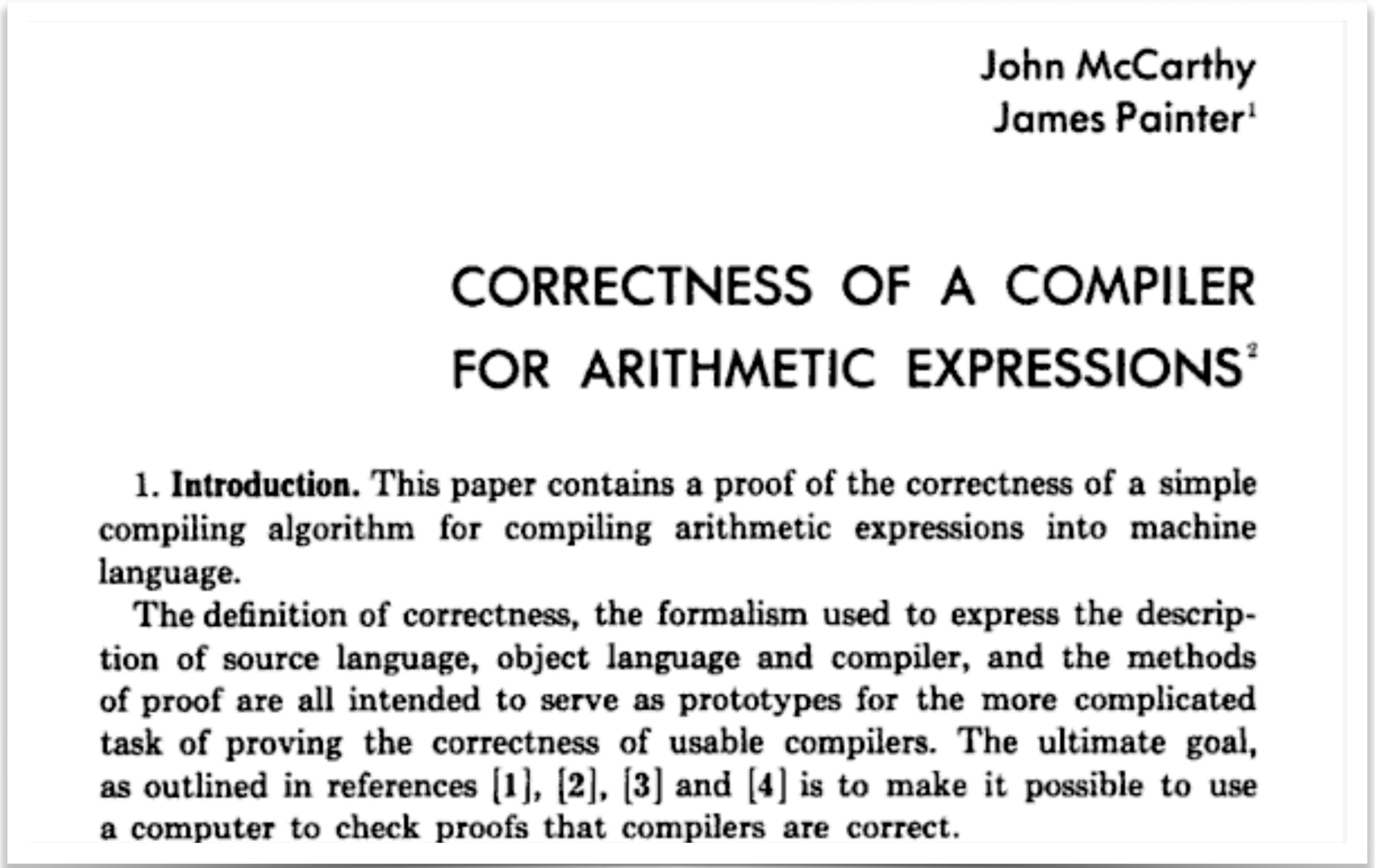
Compilers are complicated programs, but have a rather simple end-to-end specification:

> 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 …



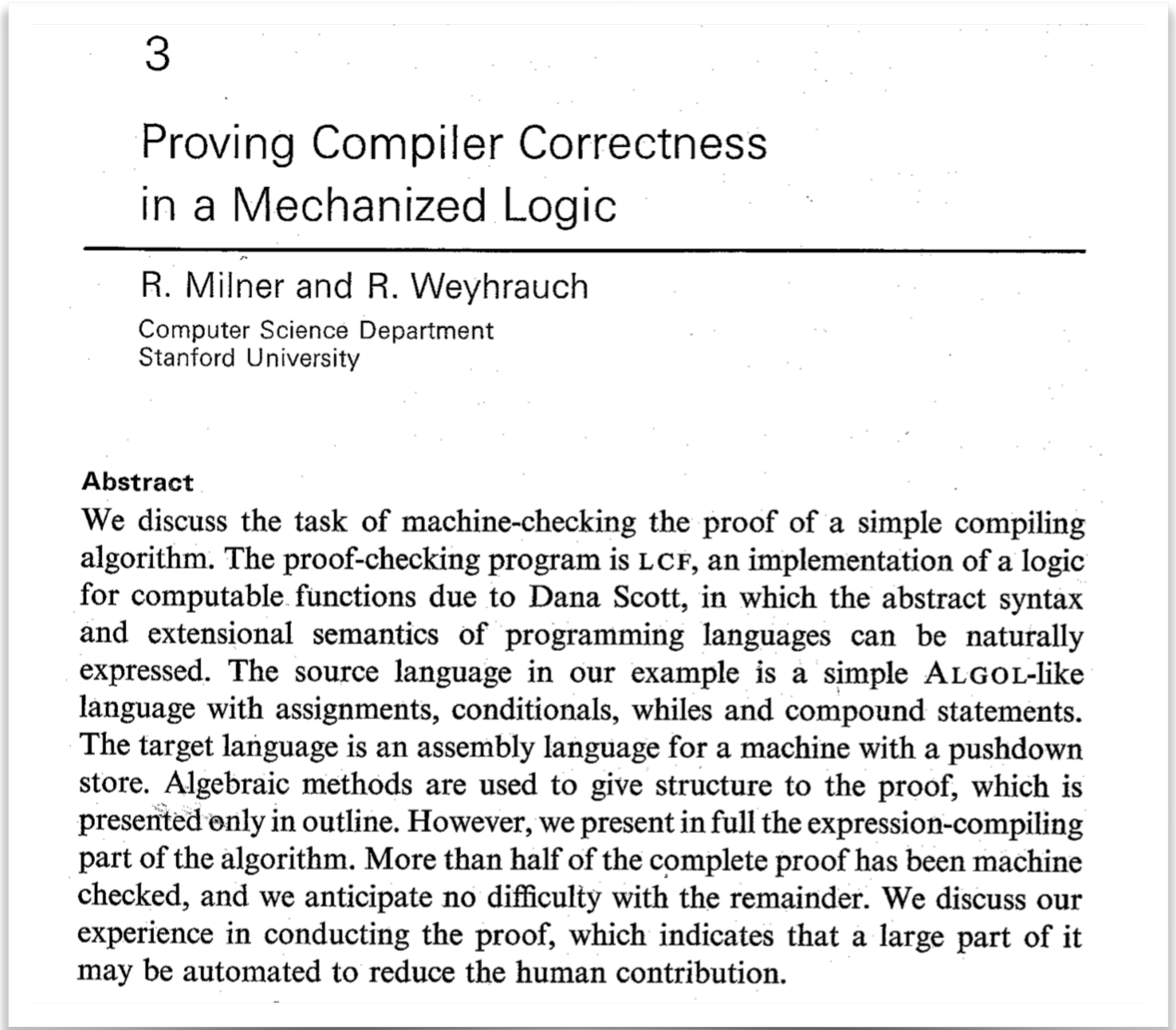
John McCarthy
James Painter[1]

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.)

```
type state = string → int
```

```
eval (s:state)(a:exp): int
```

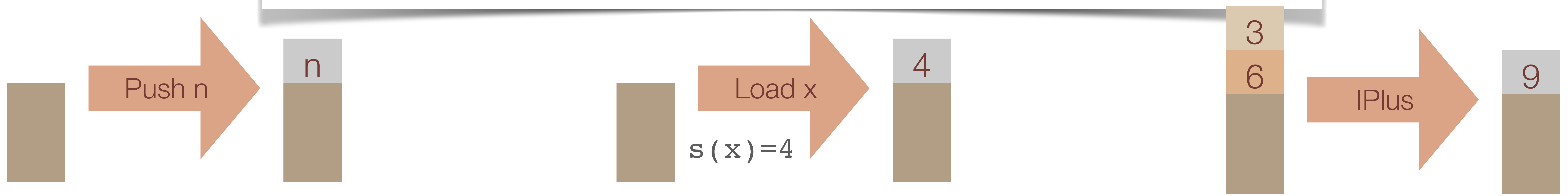
semantics
(**eval**, **exec**)

compiler
(**compile**)

compilation

```
compile (a:exp): instr list
```

```
exec(s:state)(stack: int list)(pgm: instr list): int list
```



Proving a property with the Rocq software

ACM SIGPLAN Programming Languages Software award 2013

ACM Software System award 2013 https://rocq-prover.org/

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

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

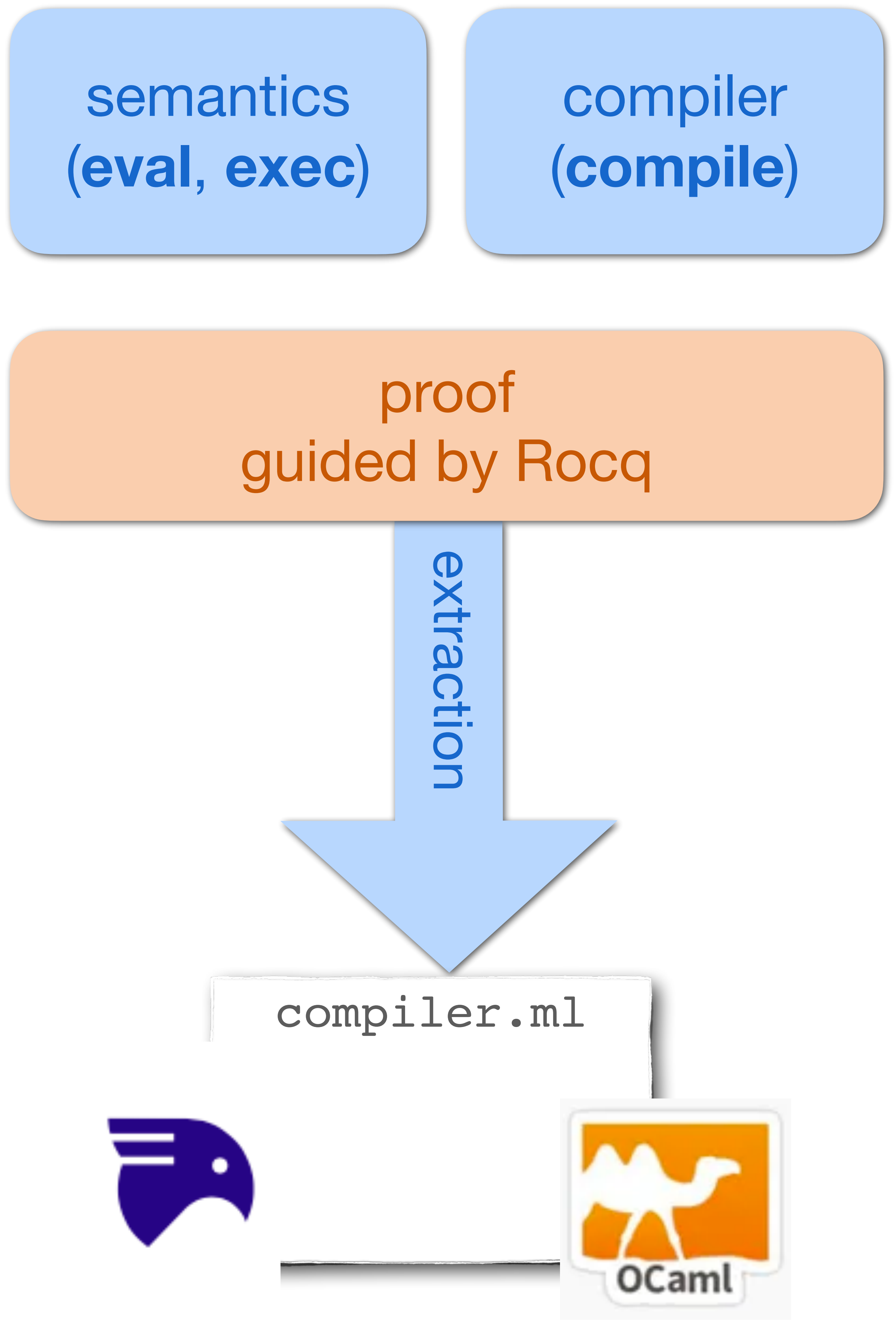
compiler
(**compile**)

Proving a property with the Rocq software

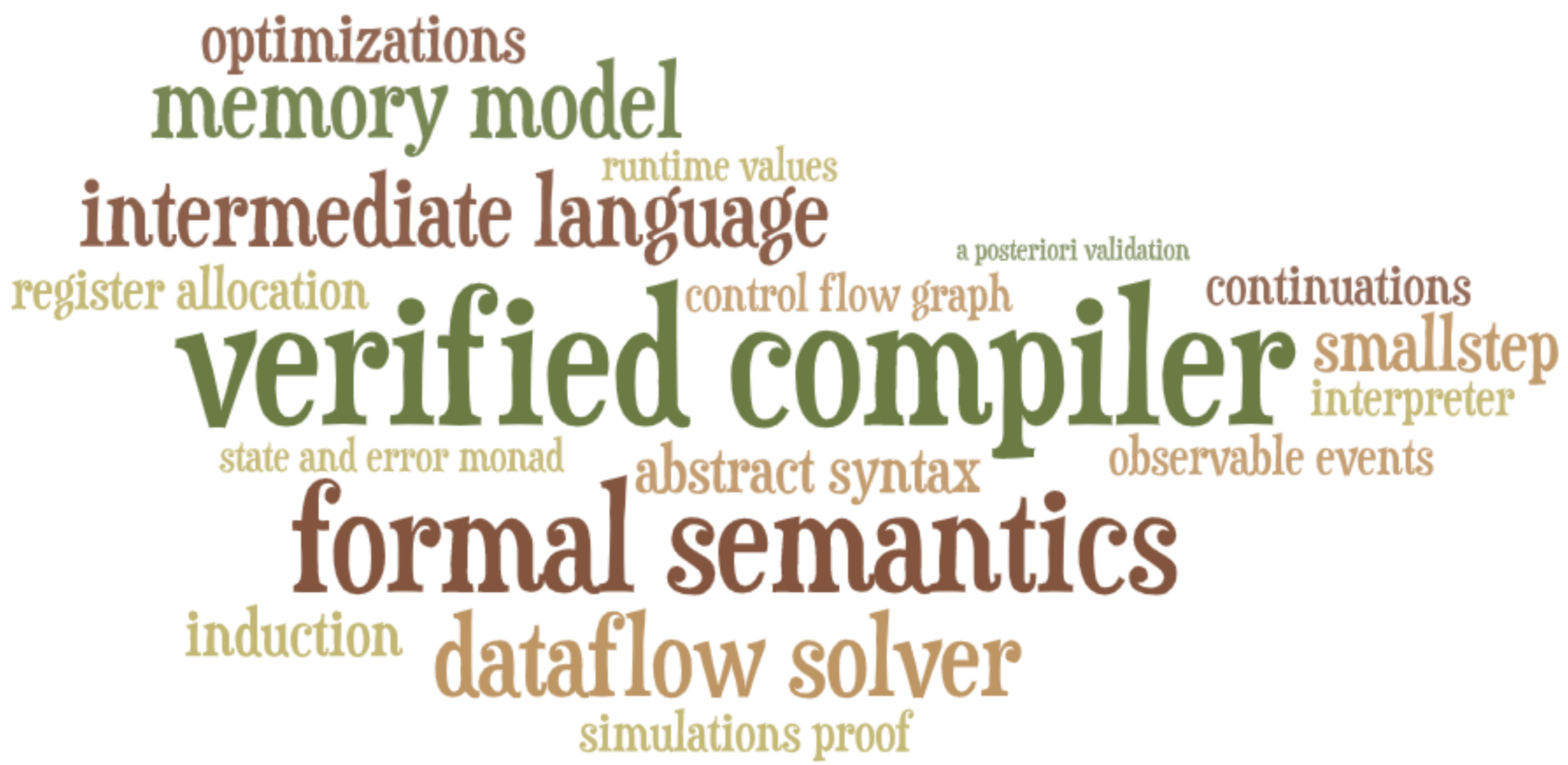
ACM SIGPLAN Programming Languages Software award 2013
ACM Software System award 2013 https://rocq-prover.org/

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



A selection of formally verified compilers

CompCert C compiler (Rocq) [Leroy, POPL’06]

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

CertiCoq Gallina compiler (Rocq) [Appel et al., CoqPL’17]

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

The CompCert formally verified compiler

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

A moderately optimizing C compiler

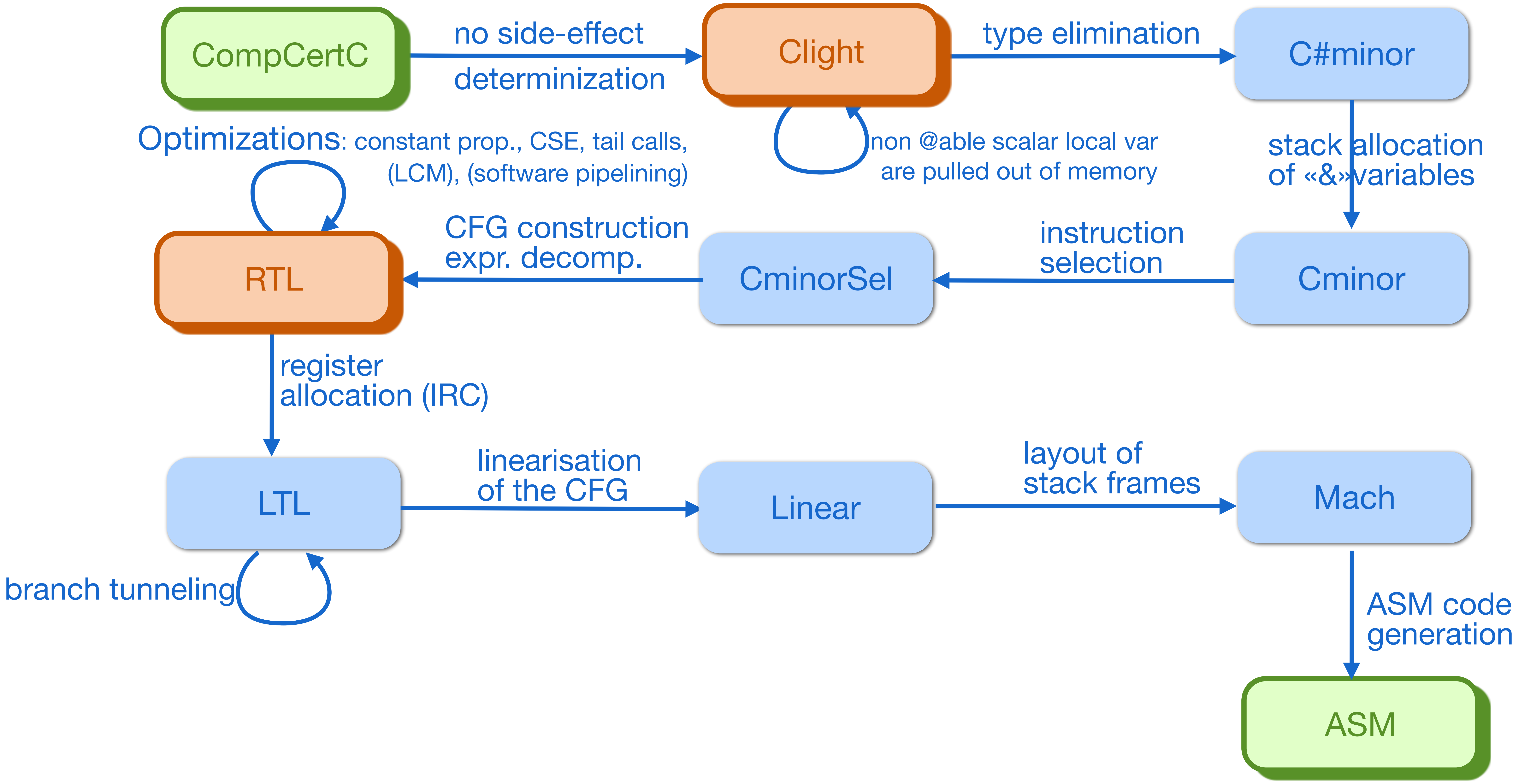
Targets several architectures (PowerPC, ARM, RISC-V and x86)

Used in commercial settings (for emergency power generators and flight control navigation algorithms) and for software certification

Improved performances of the generated code while providing proven traceability information

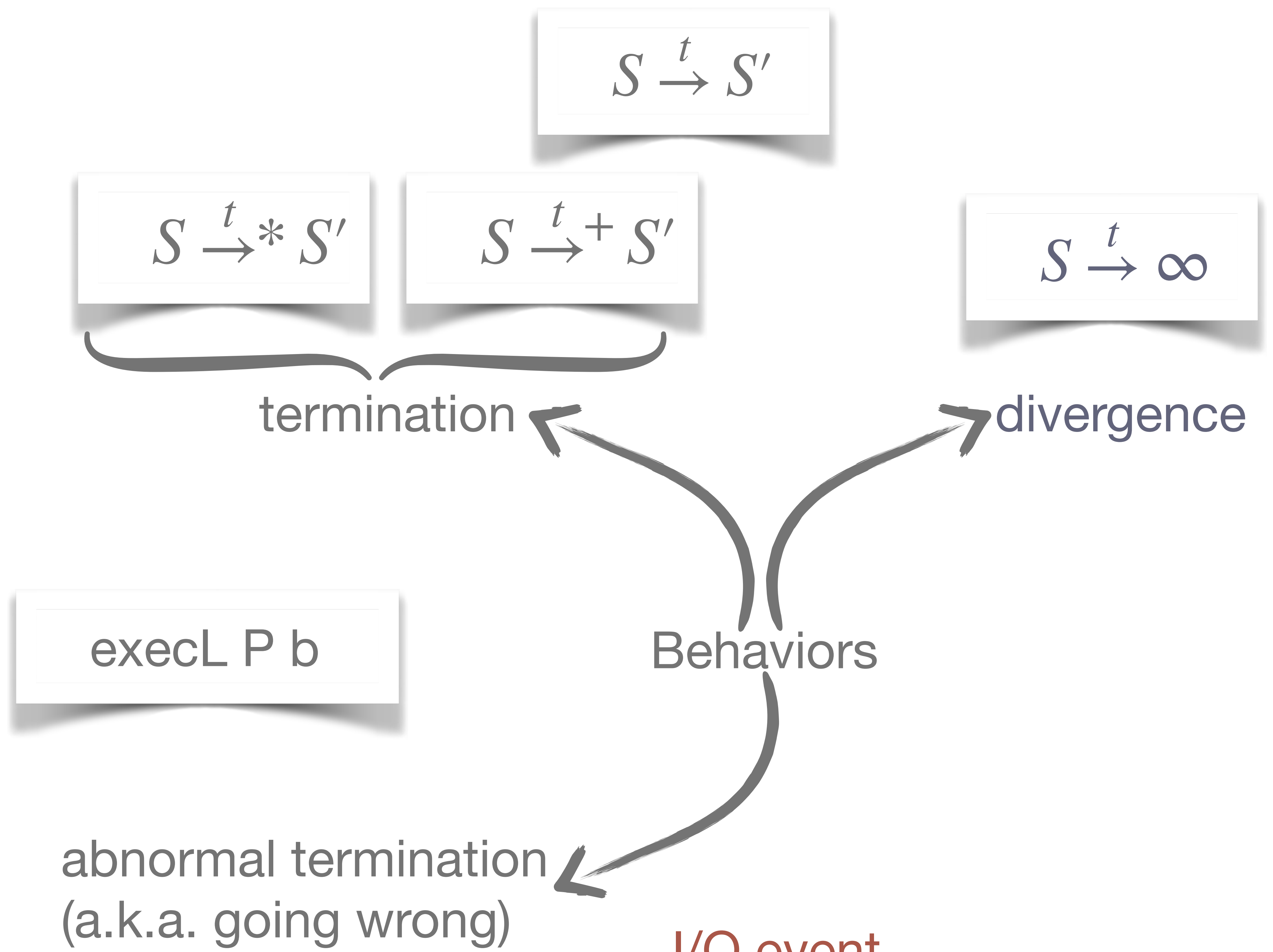
ACM Software System award 2021
ACM SIGPLAN Programming Languages Software award 2022

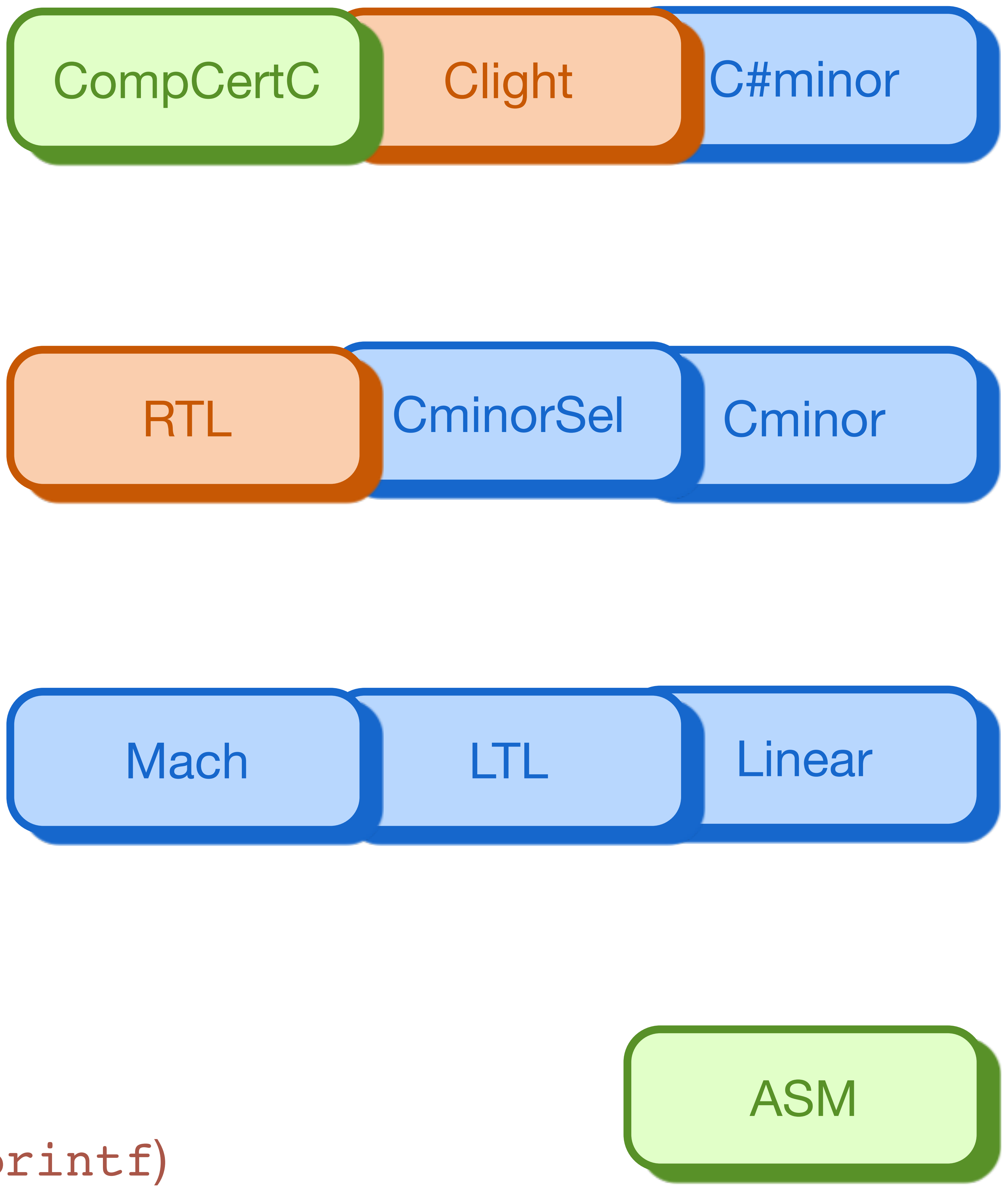
CompCert compiler: 10 languages, 18 passes



CompCert compiler: 10 languages, 18 passes

Small-step operational semantics





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**)

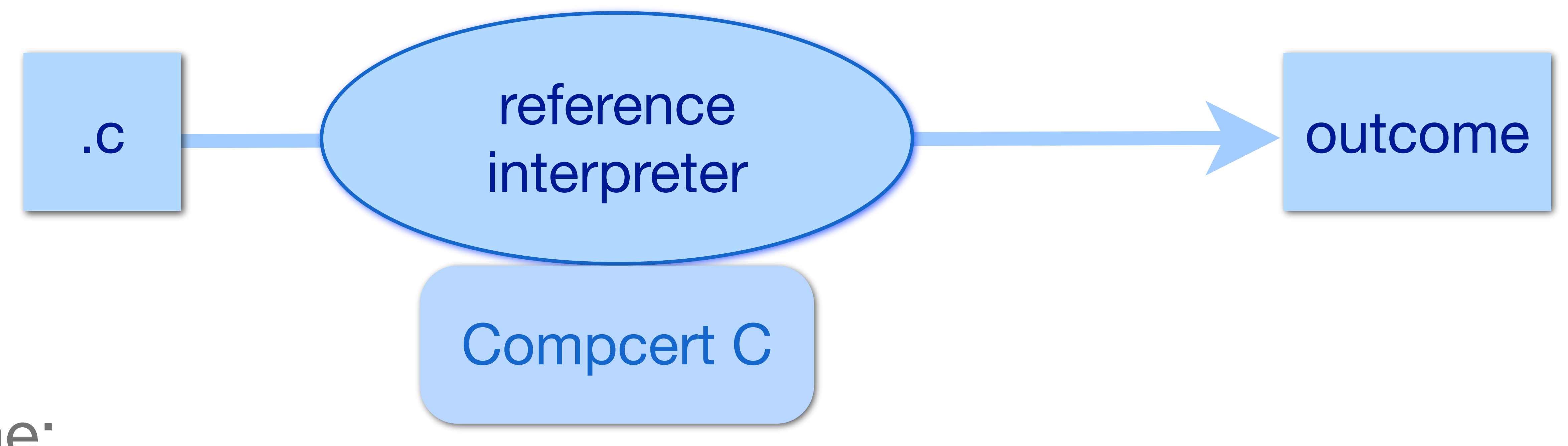
compiler

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. »

Testing the specification: CompCert C reference interpreter



Outcome:

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

Faithful to the semantics of CompCert C
The interpreter displays all the behaviors according to the semantics.

Using the reference interpreter
A first example

```c
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;  }

int main(void) {
    int res = f(12);
    printf("Result is %d \n",res);
    return 0; }
```

reference interpreter

```
Result is 76
Time 387: observable event: extcall printf(& __stringlit_1, 76)
Time 392: program terminated (exit code = 0)
```

number of execution steps

Using the reference interpreter
A second example

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

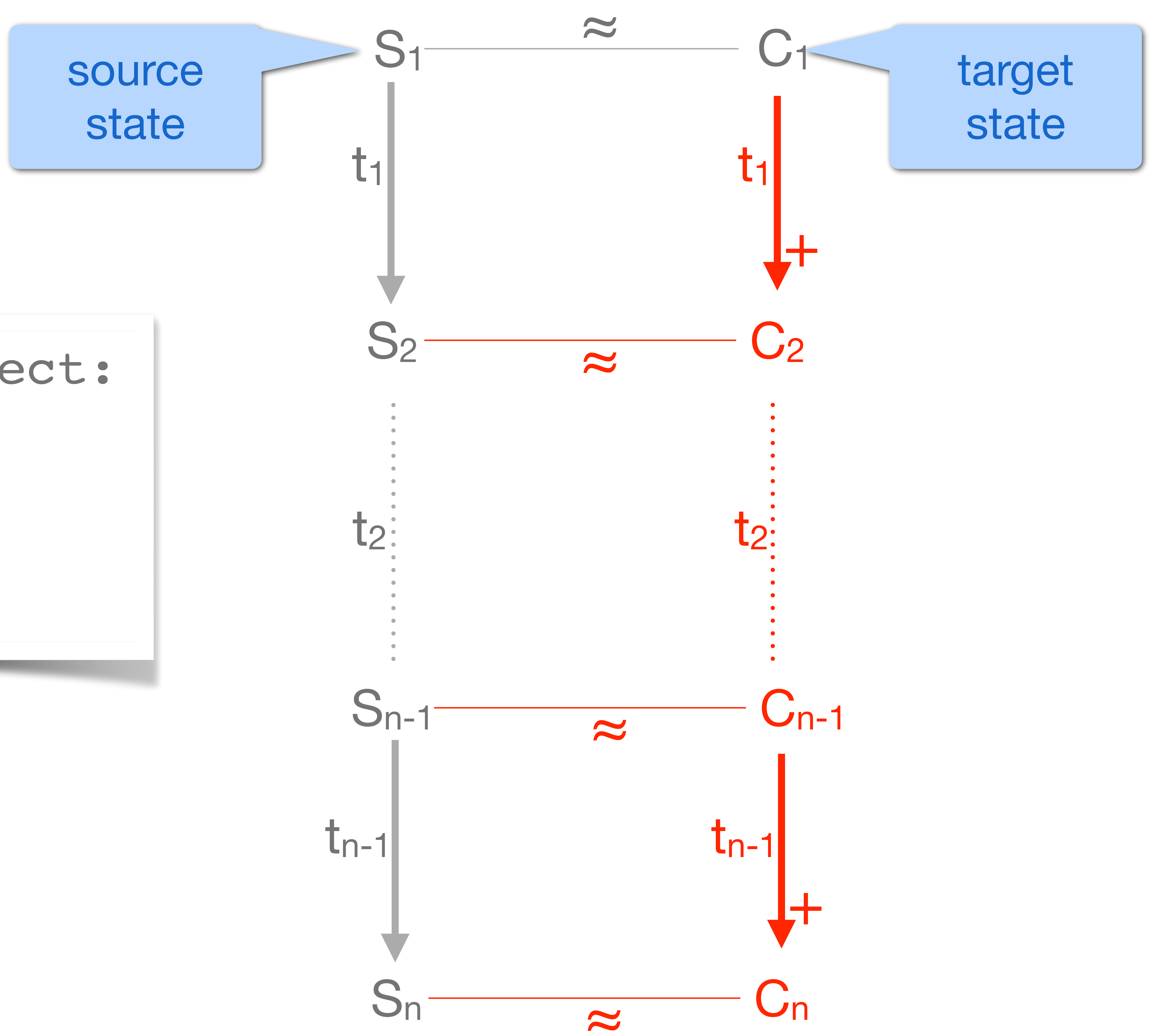
reference interpreter

The interpreter stops on this undefined behavior.
This is not the case for the compiled code.

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

Proving semantics preservation: the simulation approach

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


$$\begin{tikzcd} S_1 \arrow[r, dash, "\approx"] \arrow[d, "t_1"'] & C_1 \arrow[d, "t_1", "+"'] \\ S_2 \arrow[r, dash, "\approx"'] \arrow[d, dotted, "t_2"'] & C_2 \arrow[d, dotted, "t_2"] \\ S_{n-1} \arrow[r, dash, "\approx"'] \arrow[d, "t_{n-1}"'] & C_{n-1} \arrow[d, "t_{n-1}", "+"'] \\ S_n \arrow[r, dash, "\approx"'] & C_n \end{tikzcd}$$

Proving semantics preservation: the simulation approach

source state

target state

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

or

$$\begin{tikzcd} S \arrow[r, dash, "\approx"] \arrow[d, "\varepsilon"'] & C \arrow[dl, dash, "\approx"] \\ S' & \end{tikzcd}$$

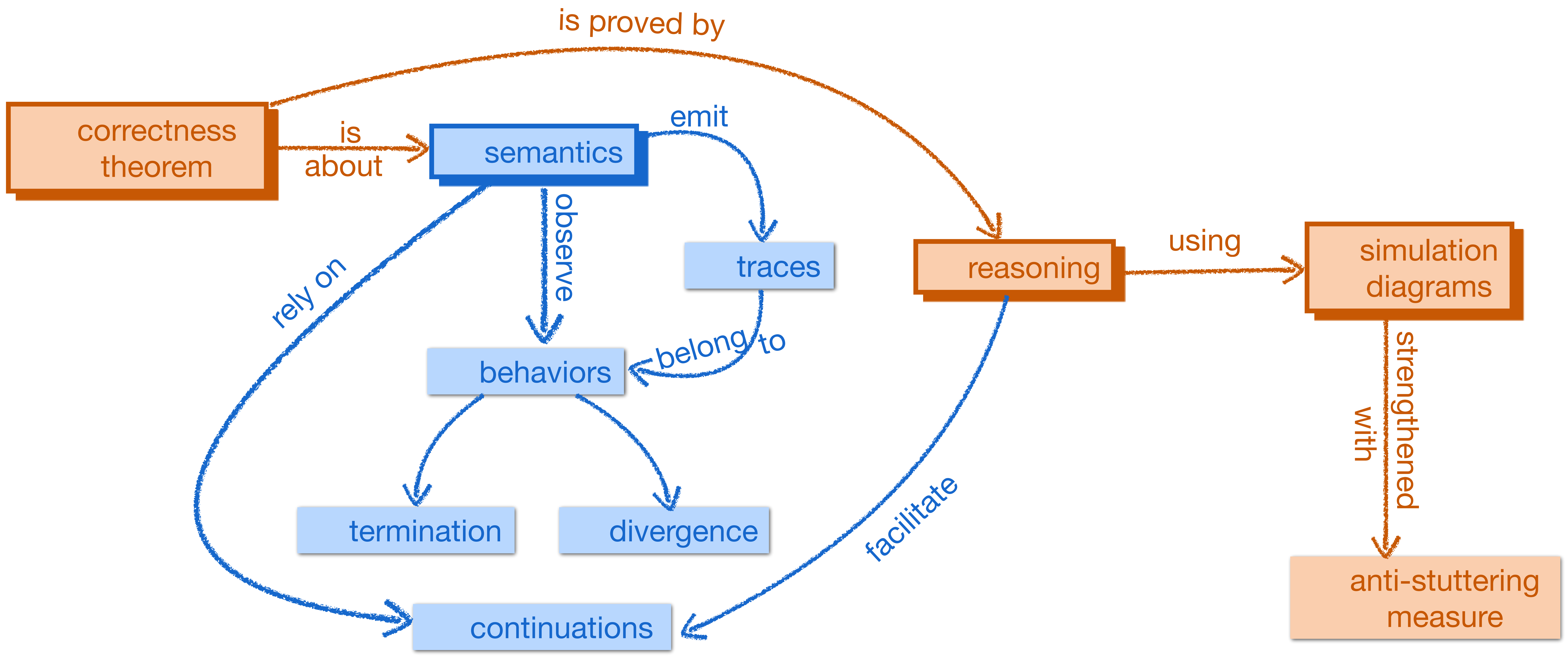
with $0 \le m(S') < m(S)$

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



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?

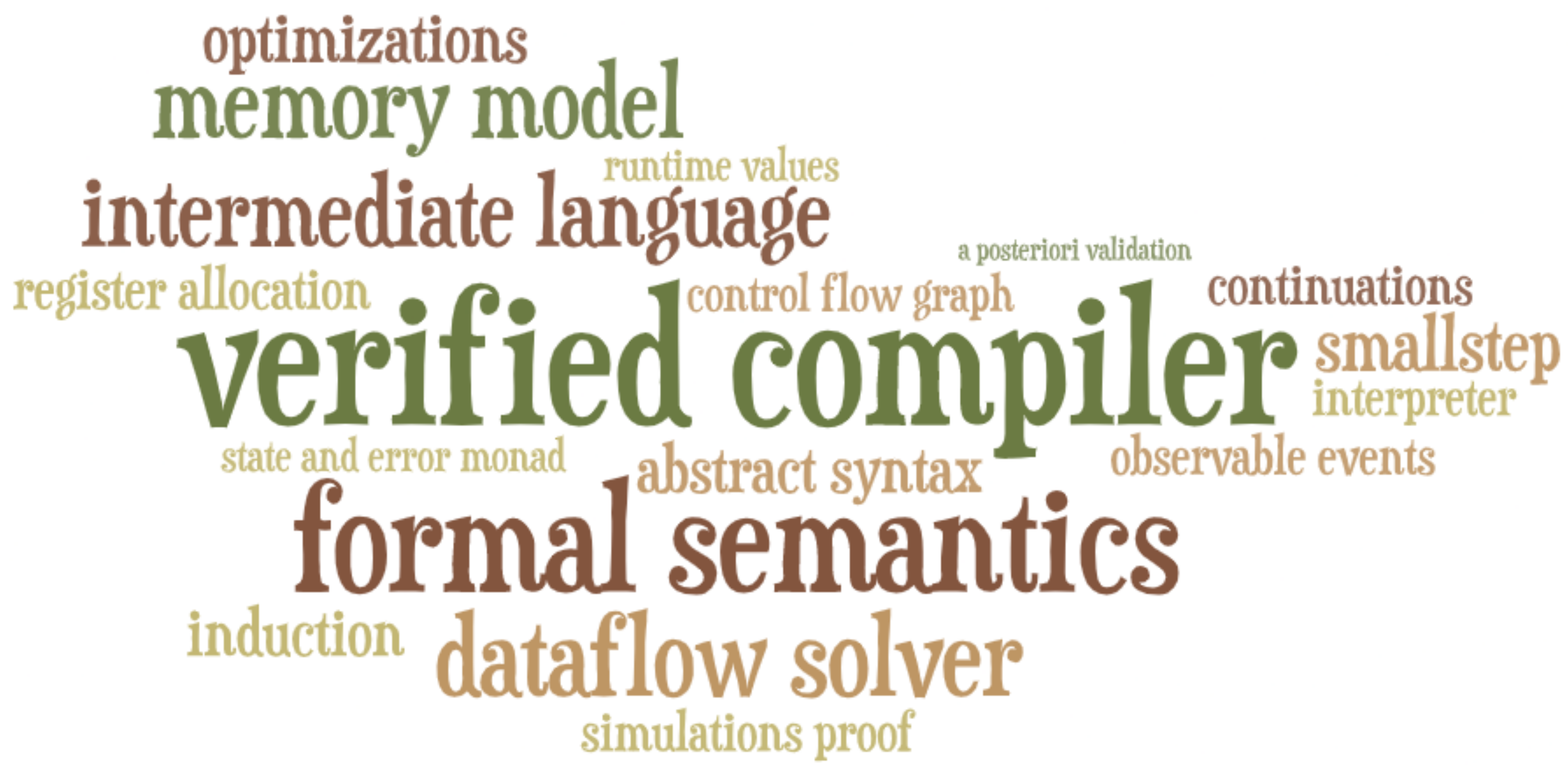
```
Theorem compiler-correct:
  ∀ S C b,
  compiler S = OK C →
  execCompCertC S b →
  execASM C b.
```

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

observe program leakages (boolean guards and memory accesses)

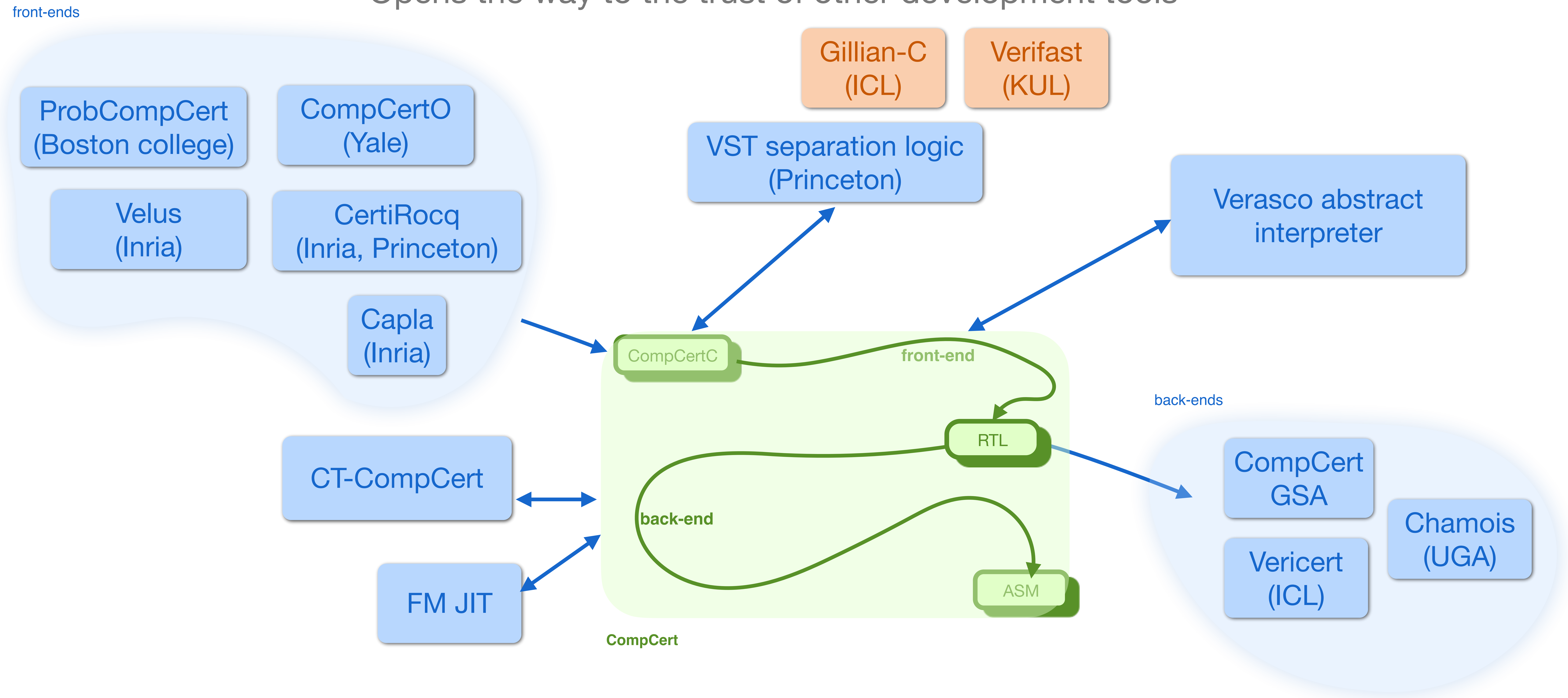
2 executions of S from 2 indistinguishable states (only share public values)

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



In closing

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

Future directions

Connection w.r.t. hardware verification

More formal guarantees for software written in recent languages

Thank you!

Questions?