A quick tour on CompCert

Sandrine Blazy

Université de Rennes

UMR IRISA

cnrs

Inria

Utilizing and scaling the WebAssembly Semantics, Dagstuhl seminar, 2025-06-10

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)

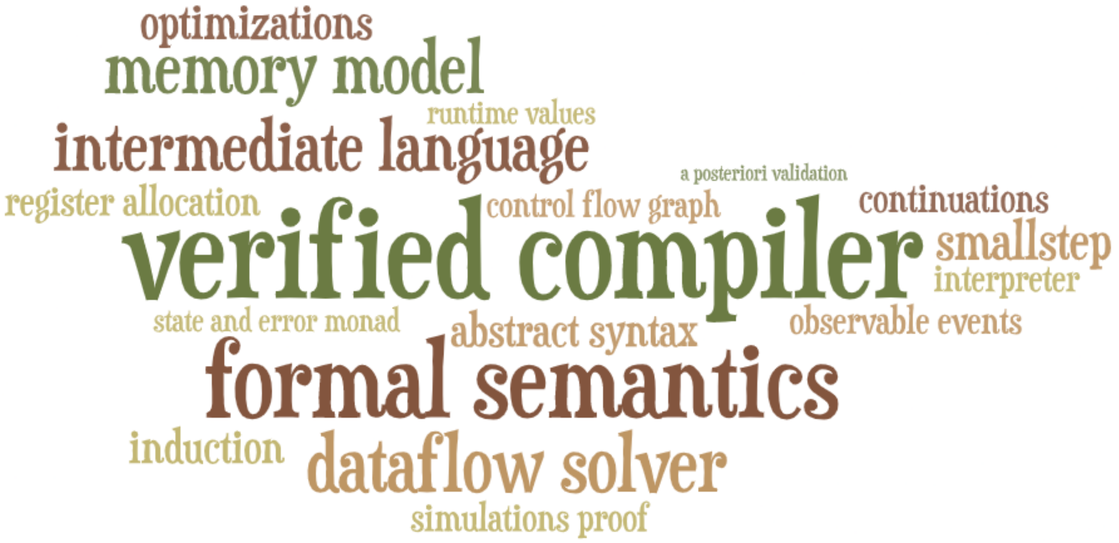
The generated code must behave as prescribed by the semantics of the source program.

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

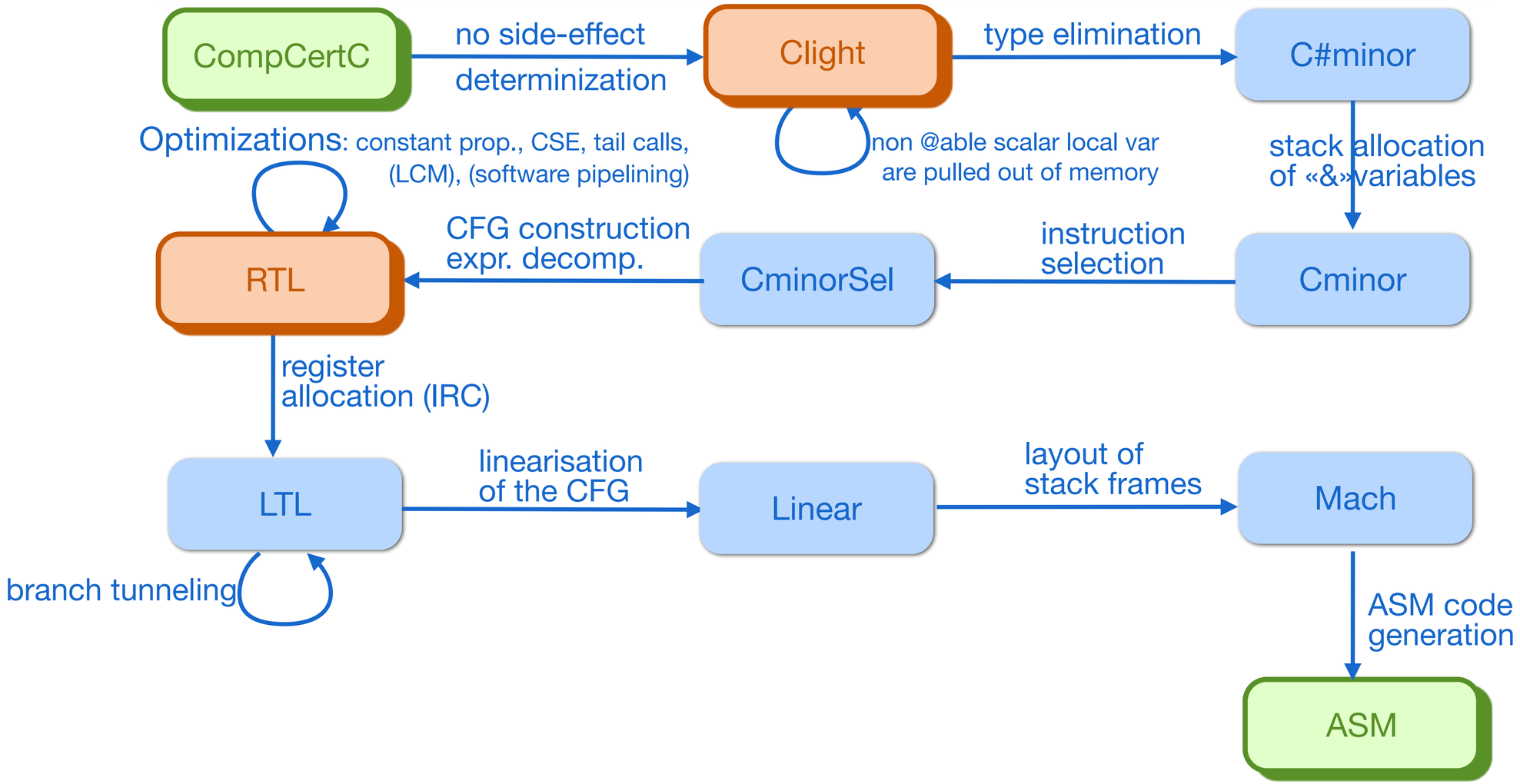
Improved performances of the generated code while providing proven traceability information

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

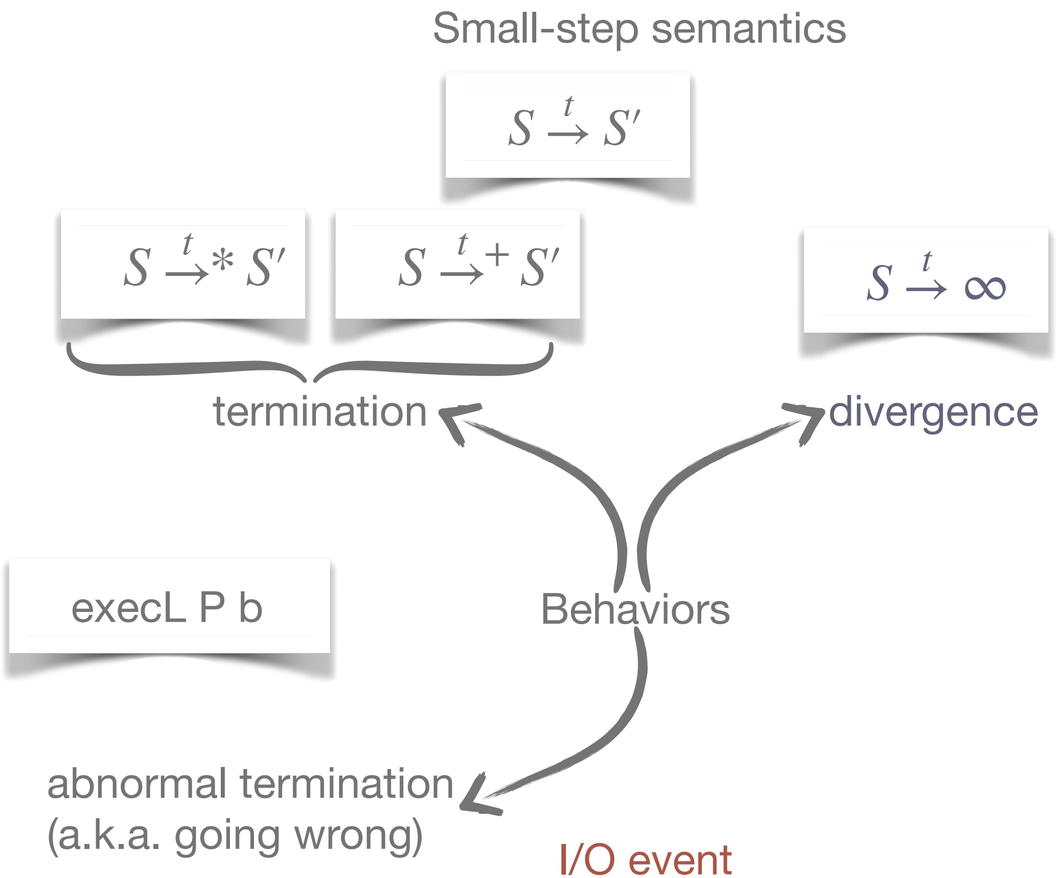
Part 1
Operational semantics

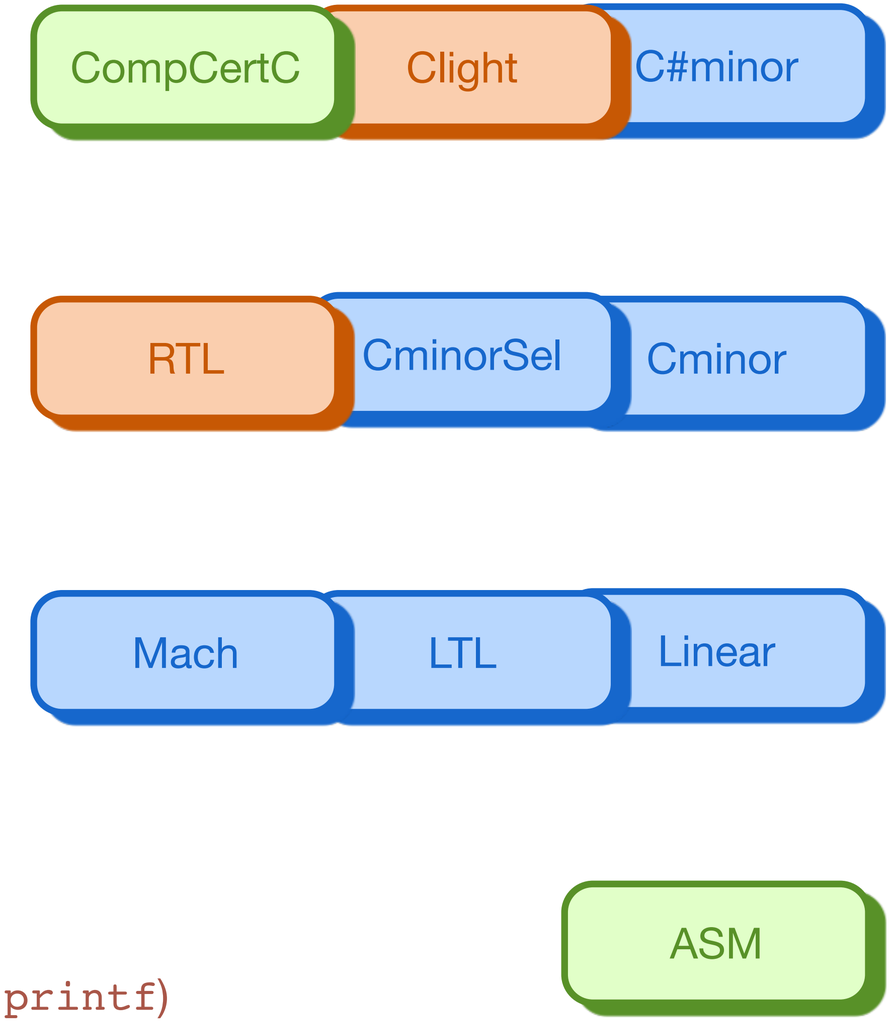


CompCert compiler: 10 languages, 18 passes



CompCert compiler: 10 languages, 18 passes





I/O event

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

The CompCert memory model

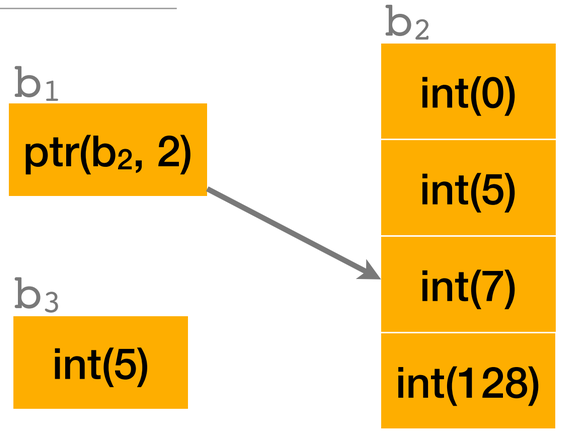
Shared by all the languages of the compiler

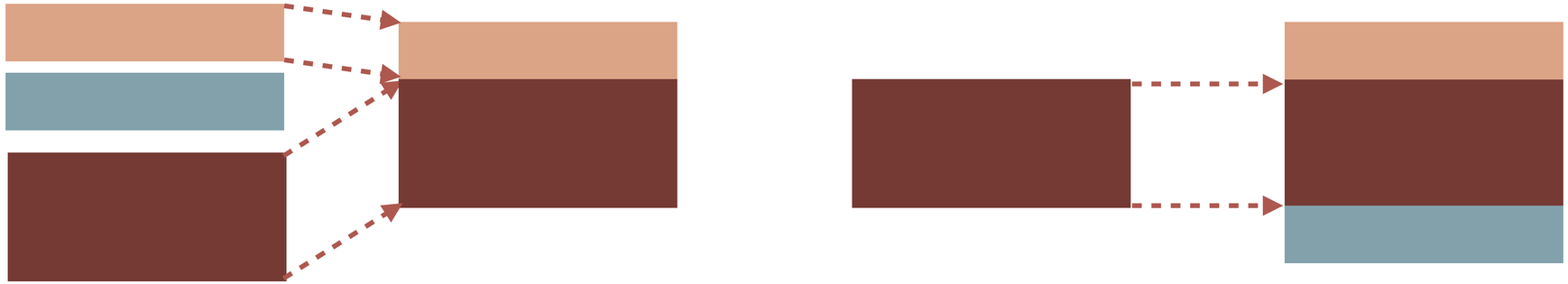
An abstract view of memory refined into a concrete memory layout

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

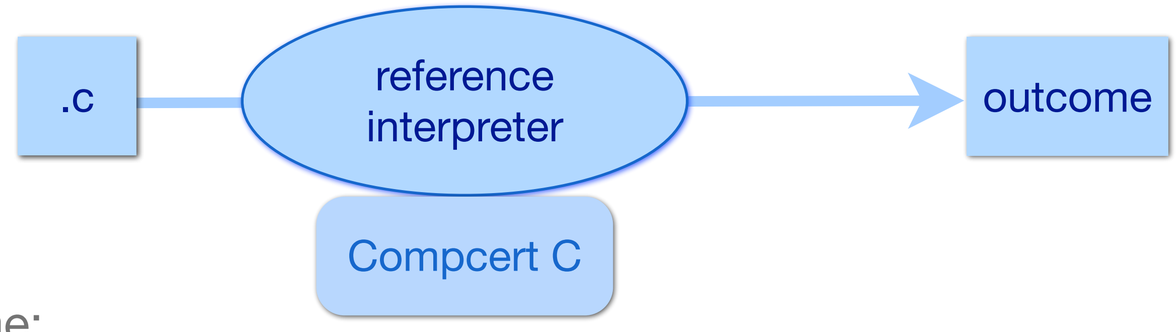
Memory safety preserved by CompCert
(not out-of-bound access + good variable properties)

Generic memory injections and memory extensions





The CompCert C reference interpreter



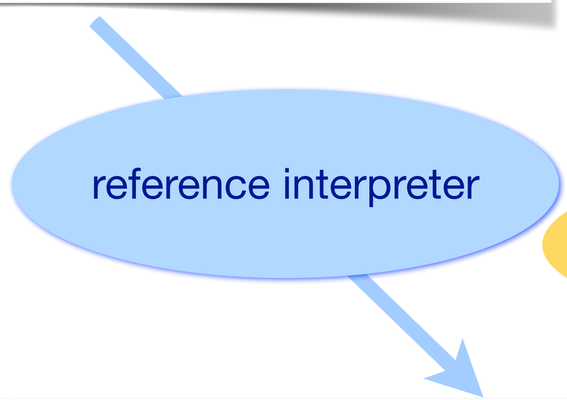
Outcome:

- 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
Example

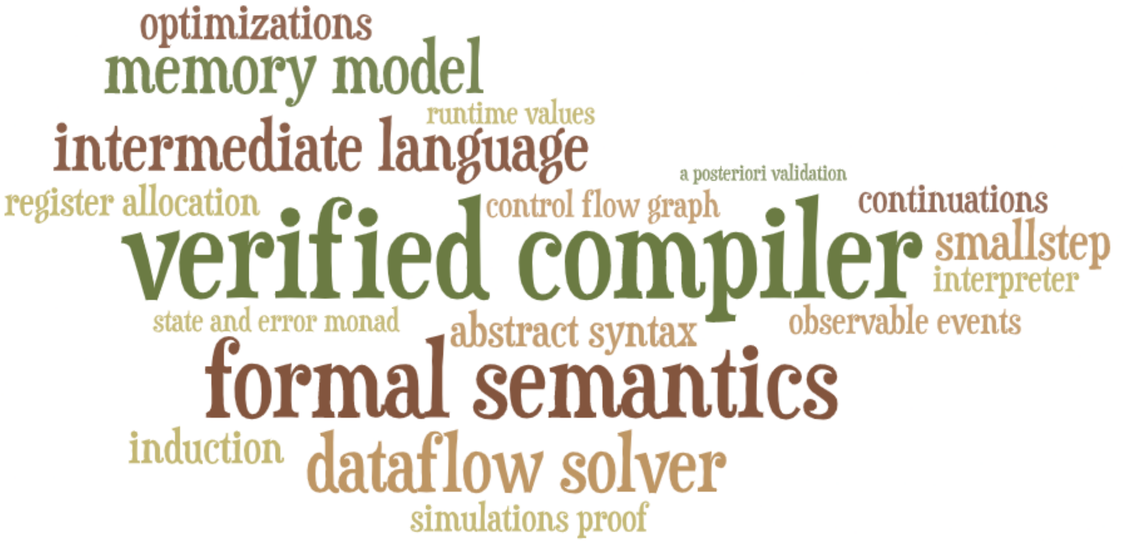
```
int main(void)
{  int x[2] = { 12, 34 };
  printf("x[2] = %d\n", x[2]);
  return 0;  }
```



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
```

Part 2
Semantic reasoning



Proving semantics preservation: the simulation approach

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

compiler

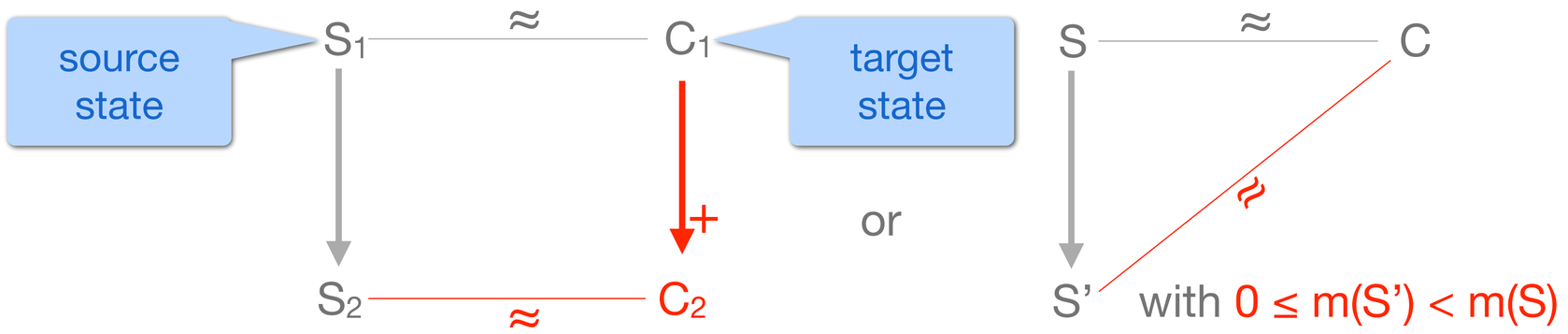
Preserved behaviors = termination and divergence

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

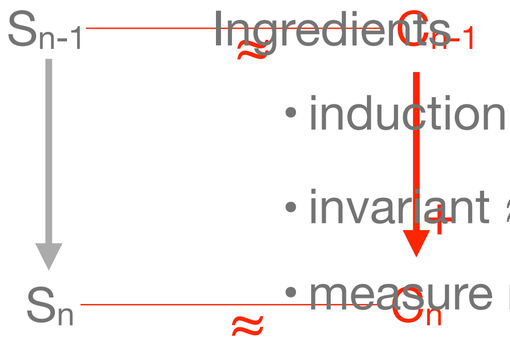
« The generated code must behave as prescribed by the semantics of the source program. »

Proof technique: simulation diagram

Proving semantics preservation: the simulation approach



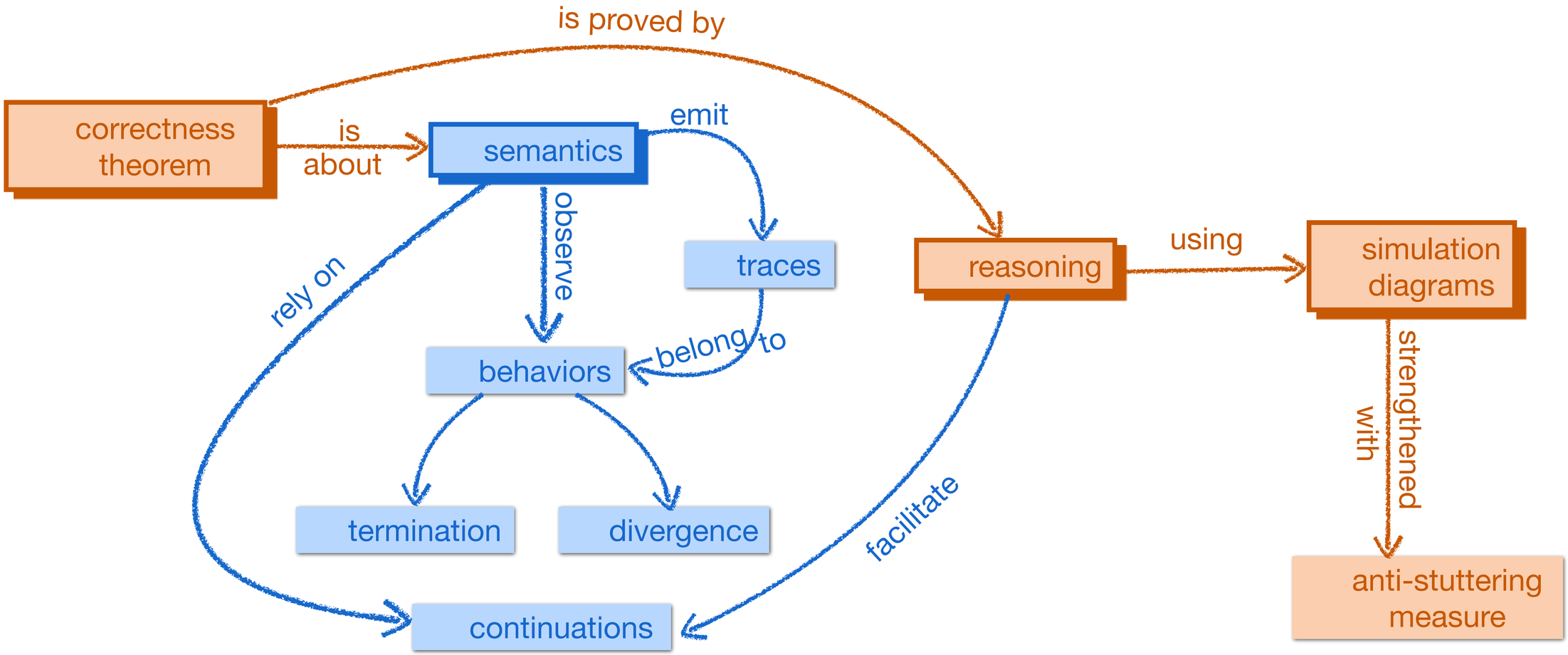
If the source program diverges, it must perform infinitely many non-stuttering steps, so the compiled code executes infinitely many steps.



Ingredients:

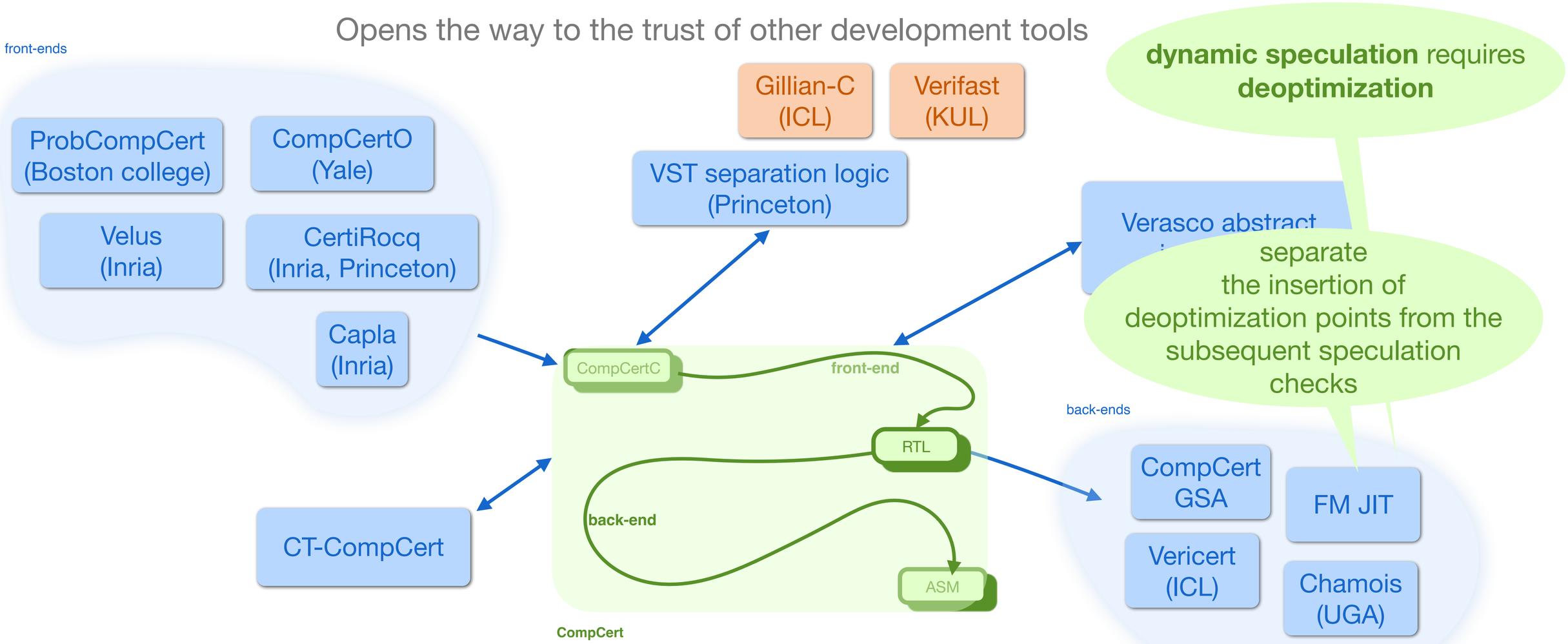
- induction on the execution relation
- invariant ≈ between source and target states
- measure m from source states to a well-founded set

Semantic reasoning for compiler correctness



CompCert, an open infrastructure for research

Opens the way to the trust of other development tools



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

Questions?