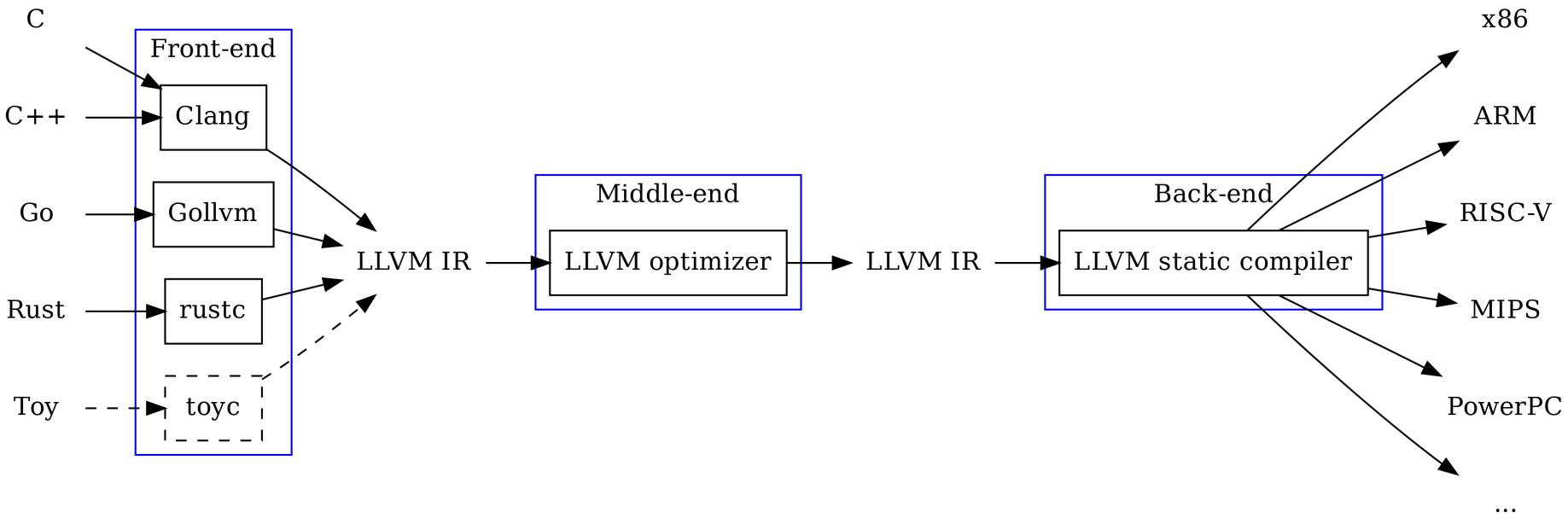
Compiler Correctness

Jingyu Qiu

University of Rochester

April 17, 2024

Compiler pipeline



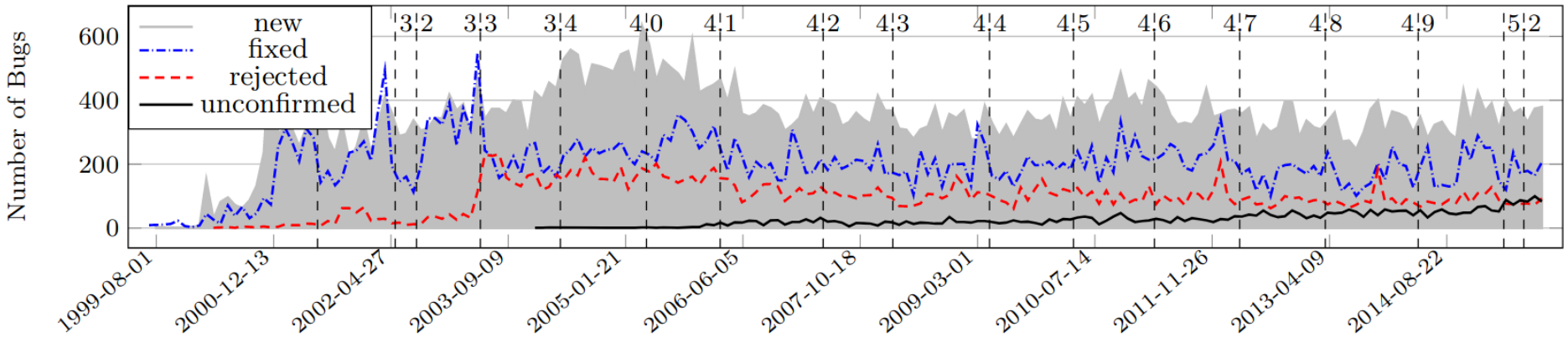
The compilation process [Leroy(2019)]

In general: any translation from a computer language to another one.

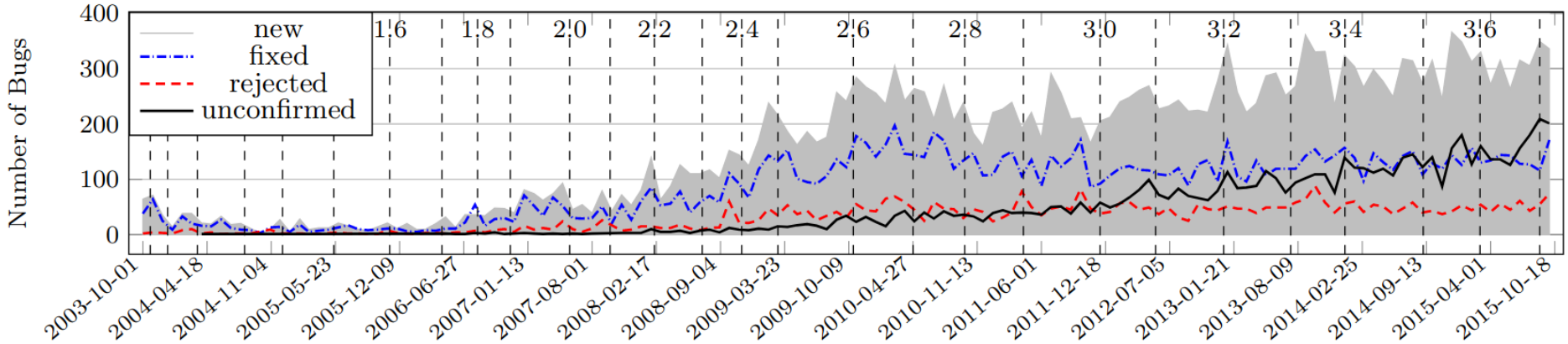
More specifically:

- automatic translation
- from a high-level language suitable for programming by humans
- to a low-level language executable by machines
- with a concern for efficiency ("optimizing" compilers)
- often used as a black-box and assume correctness

Compiler bugs [Sun et al.(2016)]



(a) GCC.



(b) LLVM

Miscompilation

Source

```
a = x * y
if (j == 1){
    x = 1
    b = x * y
}
else{
    b = x * y
}
return b
```

Target

```
a = x * y
if (j == 1){
    x = 1
    b = a
}
else{
    b = a
}
return b
```

Miscompilation [Leroy(2019)]

Bugs in the compiler can make it produce wrong executable code for a correct source program.

For low-assurance software:

- miscompilation is negligible compared with bugs in the program itself
- when happen, it is very hard to track down the cause

For high-assurance software:

- e.g. aircraft, vehicle, cardiac device
- source programs are often formally verified against its specification
- miscompilation can invalidate the guarantees obtained by the inspection to source program

Semantic preservation [Leroy(2019)]

We've claimed that compilers should "preserve semantics" or "produce code that executes in accordance with the semantics of the source program".

- What does this mean exactly?
- Should source and compiled code do exactly the same thing?
- What should be preserved?

The only thing that matters is the observable behavior.

Observable behavior

Source

```
int sum_10()
    int res = 0
    for (int i = 1; i<= 10; i++){
        res += i
    }
    return res
```

Target

```
int sum_10()
    return 55
```

Observable behaviors [Leroy(2019)]

For realistic languages, observable behaviors include

- termination
- divergence
- abnormal termination
- I/O operations

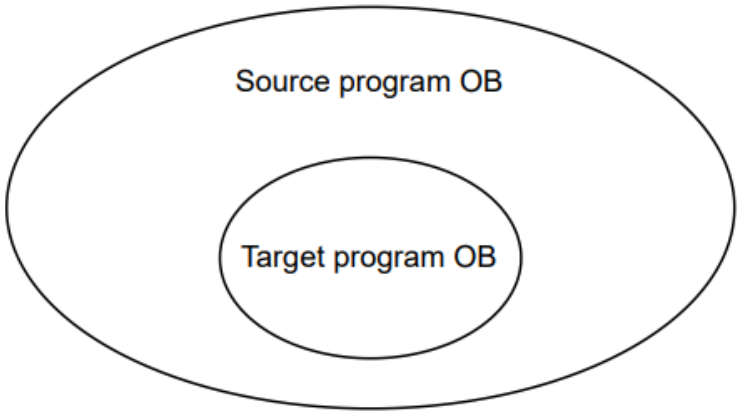
However, program contains nondeterminism:

- unspecified behavior e.g. evaluation order of $foo(g(), k())$
- undefined behavior e.g. use of uninitialized variable

Refinement relation (backward simulation) [Leroy(2019)]

Definition (refinement)

Every possible behavior of the compiled program C is a possible behavior of the source program S. However, C may have fewer behaviors than S.



Refinement suffices to show the preservation of properties established by source-level verification: If the behavior of S satisfy a specification *Spec*, then the behavior of C satisfy *Spec* as well.

Refinement simplified [Lopes et al.(2021)][Leroy(2019)]

For simplicity, we assume determinism for source program and compiled program.

- f_{src}/f_{tgt} are the source/target program, which can be regarded as functions
- I_{src}/I_{tgt} are inputs to the source/target functions
- O is the return value of functions
- $f(I, O)$ means function f will output O given input I

Then we have the simplified refinement as follows:

$$\forall I_{src}, I_{tgt}, O.(I_{src} = I_{tgt} \wedge f_{tgt}(I_{tgt}, O)) \implies f_{src}(I_{src}, O)$$

Some attempts to approach compiler correctness

Translations validation:

- avoid the need to inspect the complex logic inside compiler
- look at one transformation each time

Compiler testing/fuzzing:

- generate quality source programs for compiler
- check target program

Correct by construction:

- construct the whole compiler in proof assistant
- corresponding proof is done in proof assistant

Translation validation

The goal is to check whether this source program and this target program have the same observable behavior (same return value in our simplified case).

Source

```
foo(x, y, j):
    a = x * y
    if (j == 1){
        x = 1
        b = x * y
    }
    else{
        b = x * y
    }
    return b
```

Target

```
foo(x, y, j):
    a = x * y
    if (j == 1){
        x = 1
        b = a
    }
    else{
        b = a
    }
    return b
```

SAT/SMT solver

SAT stands for Boolean satisfiability problem. It is a problem of deciding the bool variable assignment that satisfies a given proposition formula.

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_4 \vee x_5)$$

A SAT solver will output one of the following:

- A variable assignment e.g. $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, $x_4 = \mathit{false}$, $x_5 = \mathit{true}$
- UNSAT (there is no assignment that can make the formula *true*)
- Don't know (due to intractability)

Satisfiability and Validity

SAT solver tries to find a variable assignment such that the given proposition formula is true. For any valid proposition such as

$$x \vee \neg x$$

The formula is true for any assignment. As a result, to prove its validity, we ask the SAT solver to find an assignment that will make its negation true, which is

$$\neg x \wedge x$$

If the SAT solver returns UNSAT, we have proven the formula.

SAT/SMT solver

SMT stands for Satisfiability module theory. Is is built based on SAT solver, with support for more complex variable type other than boolean.

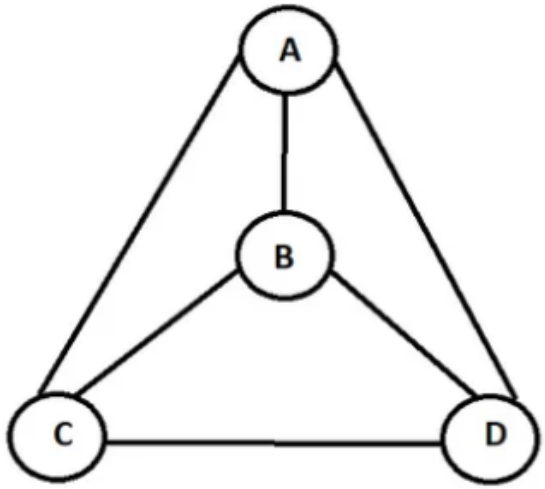
$$(x + 2y = 10) \wedge (x + y = 7)$$

Similarly SMT solver will output one of the following:

- A variable assignment e.g. $x = 4$, $y = 3$
- UNSAT
- Don't know

SMT for graph coloring problem

Fill in colors to nodes such that adjacent nodes have different colors.



$$
\begin{aligned}
&(A = 1 \vee A = 2 \vee A = 3 \vee A = 4) \wedge \\
&(B = 1 \vee B = 2 \vee B = 3 \vee B = 4) \wedge \\
&(C = 1 \vee C = 2 \vee C = 3 \vee C = 4) \wedge \\
&(D = 1 \vee D = 2 \vee D = 3 \vee D = 4) \wedge \\
&\neg(A = C) \wedge \neg(A = B) \wedge \neg(A = D) \wedge \\
&\neg(B = C) \wedge \neg(B = D) \wedge \neg(C = D)
\end{aligned}
$$

Output: $A = 1,\ B = 2,\ C = 3,\ D = 4$

SMT encoding for source program

Source

```
foo(x1, y1, j1):
    a1 = x1 * y1
    if (j1 == 1){
        x1‘ = 1
        b1 = x1‘ * y1
    }
    else{
        b1 = x1 * y1
    }
    return b1
```

SMT encoding for Source:

$$(a1 = x1 * y1) \wedge (x1‘ = 1) \wedge (b1 = ITE(j1 == 1, x2 * y1, x1 * y1))$$

SMT encoding for target program

Target

```
foo(x2, y2, j2):
    a2 = x2 * y2
    if (j2 == 1){
        x2' = 1
        b2 = a2
    }
    else{
        b2 = a2
    }
    return b2
```

SMT encoding for Target:

$$(a2 = x3 * y3) \wedge (x2' = 1) \wedge (b2 = ITE(j2 == 1, a2, a2))$$

SMT check for refinement

Recall the definition of refinement

$$\forall I_{src}, I_{tgt}, O.(I_{src} = I_{tgt} \wedge f_{tgt}(I_{tgt}, O)) \implies f_{src}(I_{src}, O)$$

Its negation is

$$\exists I_{src}, I_{tgt}, O.(I_{src} = I_{tgt} \wedge f_{tgt}(I_{tgt}, O)) \wedge \neg f_{src}(I_{src}, O)$$

The final formula passed to SMT solver is

$$\begin{aligned}
&(a1 = x1 * y1) \wedge (x1' = 1) \wedge (b1 = ITE(j1 == 1, x2 * y1, x1 * y1)) \wedge \\
&(a2 = x3 * y3) \wedge (x2' = 1) \wedge (b2 = ITE(j2 == 1, a2, a2)) \wedge \\
&(x1 = x2) \wedge (y1 = y2) \wedge (j1 = j2) \wedge \\
&\neg(b1 = b2)
\end{aligned}$$

Output can be $x1 = 2$, $y1 = 1$, $j1 = 1$.

CBMC

If you still remember.....

```
Executing 'lcm.py' on 'lcm_pre.c'... (0/3.333333333333333)

PASS: lcm.py on 'lcm_pre.c'
PASS: compile output of lcm.py
FAIL: Correctness check #1 of 1
    SUCCESS: check d
    SUCCESS: check b
    SUCCESS: check c
    SUCCESS: check a (c > d)
    SUCCESS: check a (c <= d)
    FAILURE: return value is d
===== OUTPUT ====
```

CBMC

```c
int pre_test(int a, int b, int c, int d) {

  if(c > d) {
    a = b + c;
  }

  d = b + c;

  printf("a: %d, b: %d, c: %d, d: %d\n", a, b, c, d);

  return d;
}
```

Compiler testing, fuzzing

A typical testing activity contains:

- generate test cases as inputs (manually or automatically)
- look at the outputs (testing oracle)

When it comes to compiler testing:

- generate source programs
- look at the target programs

But what properties to check?

Check for semantics preservation

For a transformation from a source program to a target program, it is correct when the semantics is preserved. Possibility for cheating?

Come up with source programs for testing (fuzzing):

- randomly generate complex source programs
- modify previous successful source programs
- reverse compiler execution for source programs that take certain path

Check for equivalence (Metamorphic testing)

I may not know what the output is, but I know what it should never be

Assume you have a *sin* function

$$sin(x) = sin(\pi - x)$$

The focus shifts from verifying to finding bugs. (less bug means more correct?)

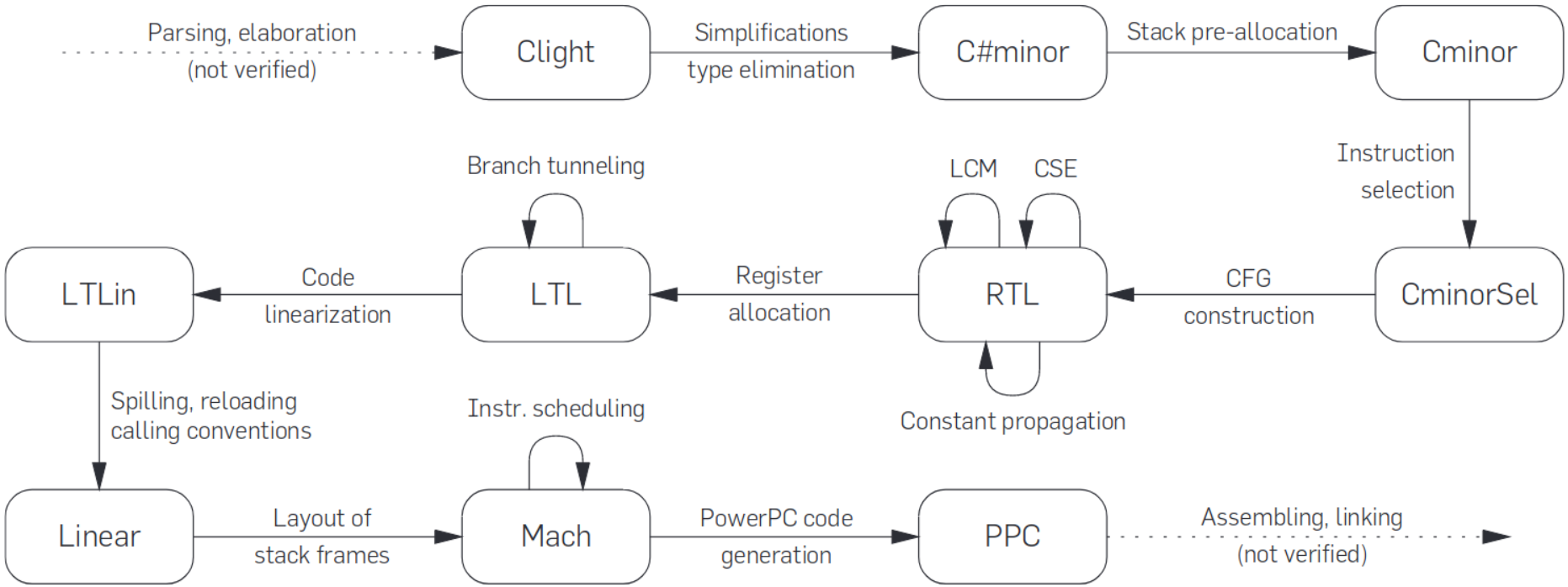
Check for equivalence

When it comes to compiler testing:

- same source programs for a compiler with near versions
- unit source programs for "peer" compilers from different vendors

Compcert

The first formally verified realistic compiler. (which took more than 5 years)



- closer look to implementation logic
- focus on code-gen transformation

Program as transition system

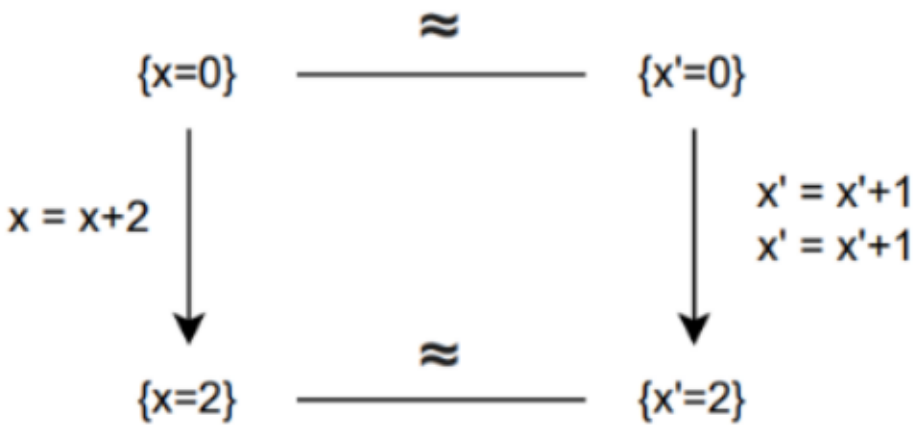
Execution of a program is a sequence of program state change.

$$\alpha_0 \to \alpha_1 \to \alpha_2 \to \cdots \to \alpha_n$$

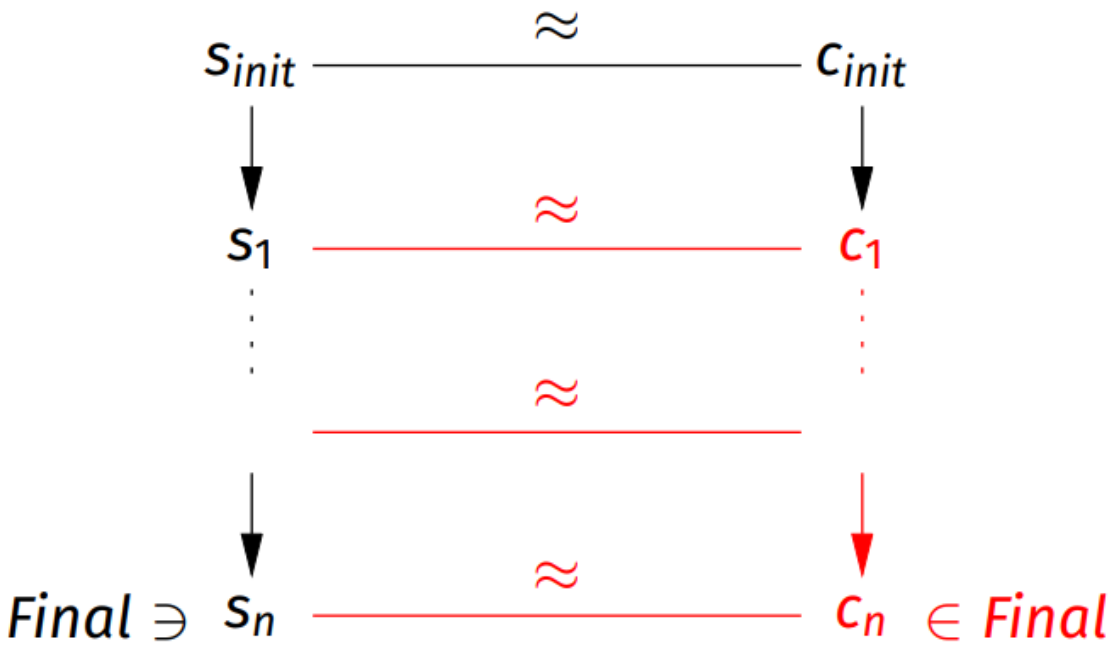
Consider an interpreter with small-step semantics. Program states only contain variable assignment.

$$\{x : 0\} \xrightarrow{x=3} \{x : 3\} \xrightarrow{x=x-1} \{x : 2\} \to \ldots$$

Simulation



Simulation diagram



Coq proof assistant

It is a proof assistant, that's all I know.

Conclusion

What is compiler correctness?

- miscompilation
- observable equivalence (semantics preservation)

Why it is important?

- miscompilation is hard to track
- source-level correctness needs to be preserved

How can we approach it?

- translation validation
- compiler testing
- proof from construction


Xavier Leroy. 2019.
Lecture Notes on European Union Types.
https://xavierleroy.org/courses/EUTypes-2019/slides.pdf.


Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021.
Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79.
https://doi.org/10.1145/3453483.3454030


Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016.
Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305.
https://doi.org/10.1145/2931037.2931074