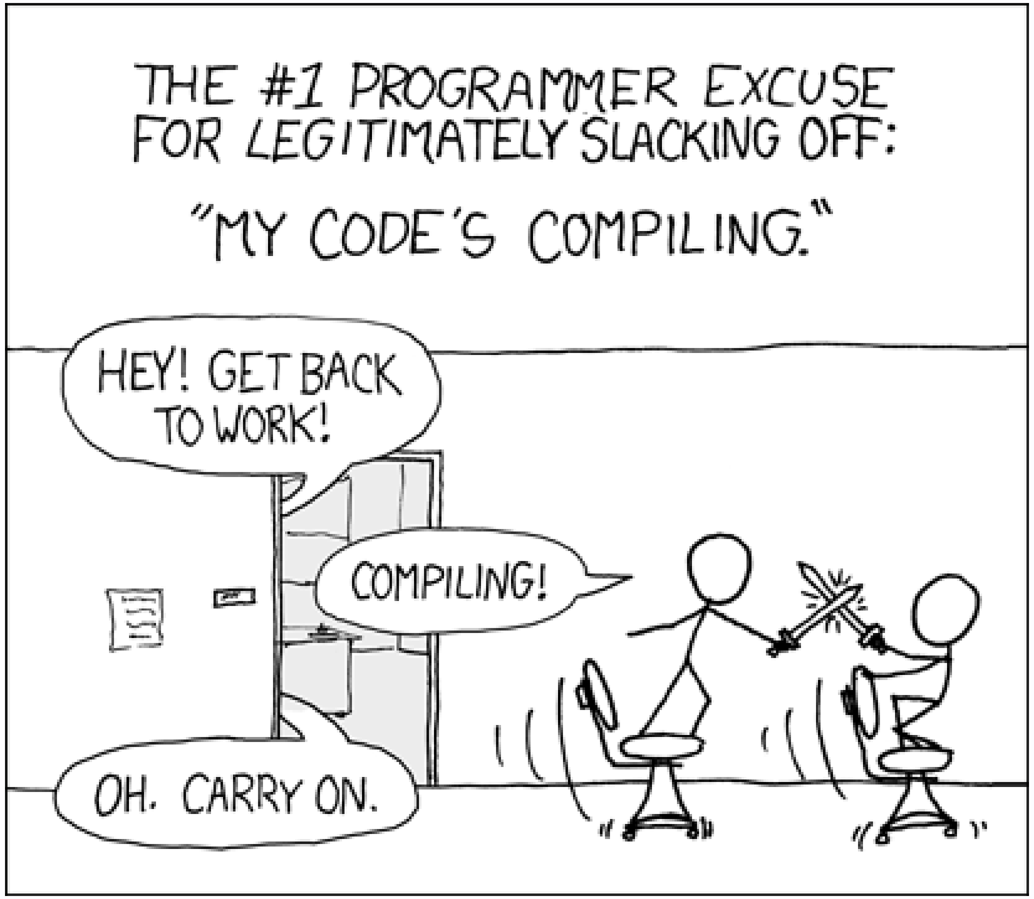


Provably Correct Compilers

Matteo Busi
(matteo.busi@di.unipi.it)

Dipartimento di Informatica, Università di Pisa

Today's agenda

- From pro**B**ably correct compilers to pro**V**ably correct ones!
- A simple correct compiler for expressions
- Beyond simple expressions
- Compilers and notions of correctness
- State of the art
- An alternative approach: translation validation
- **Wed:** beyond correctness!

Correctness: trivial?

- Aren't all compilers correct? Isn't it a trivial property?
- Well...the following is **trivially wrong**

```
for(i=0; i < 10; i++)
    printf("%d\n", i);
```

↓

```
printf("42\n");
```

Correctness: trivial? (cont.)

What about:

```c
int n = some_pt->n;
if (some_pt == NULL)
    // Some code
use (n)
```

↓

```c
int n = some_pt->n;
use (n)
```

Usually correct, but **not** when in kernel code! 🤦‍♂️

Arithmetic expressions

Recall arithmetic expressions:

$$a ::= v \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$$

when translated to a stack-based expression language:

$$i ::= Iconst(v) \mid Ivar(x) \mid Iadd \mid Isub \mid Imul \mid i_0; i_1 \mid ()$$

See the blackboard.

Correctness theorem

What's the meaning of correctness in this case?

Observe that:

1. Evaluation always terminates (why?)
2. We focus on the final result

So, show that

Theorem: $\sigma \vdash a \to^* v$ *iff* $\sigma \vdash [], [\![a]\!] \to^* [v], ()$.

Proof: By structural induction on a (see the blackboard).

Beyond expressions?

Phew! Not that simple 😛

Problem:

- This was an ad-hoc approach that does not scale well
- More complex programming languages?

Need to think *carefully* about:

- How to model compilers
- How to **define** correctness and its relation with the languages

A model for compilers

A **compiler** is a function $[\![\cdot]\!]^S_T$ that translates programs written in a source language S into programs written into a target language T.

More in general, we can see the compiler as a composition:

$$[\![\cdot]\!]^S_T \triangleq [\![\cdot]\!]^{IR_n}_T \circ \ldots \circ [\![\cdot]\!]^S_{IR_1}$$

Notation: When clear what S and T are, we will simply write $[\![\cdot]\!]$.

Notions of correctness: intuition

Intuition:

The behavior of the compiled code $\mathcal{B}(\llbracket s \rrbracket)$ **must** be the same as the behavior of the source $\mathcal{B}(s)$.

Crucial to define $\mathcal{B}$ properly:

- For expressions:
  - $\mathcal{B}(a) = \{v \mid \exists \sigma.\, \sigma \vdash a \rightarrow^* v\}$
  - $\mathcal{B}(i) = \{v \mid \exists \sigma.\, \sigma \vdash [], i \rightarrow^* [v], ()\}$
  - Shown above: $\mathcal{B}(a) = \mathcal{B}(\llbracket a \rrbracket)$
- More in general?

Behaviours

$\mathcal{B}$ depends on the set of observables of p (either in S or T):

- Set of observable actions $\mathcal{O}$, e.g. I/O ops, memory ops, return values…
- Semantics of the languages enriched with elements of $\mathcal{O}$:

$$p \to p' \qquad \text{becomes} \qquad p \xrightarrow{o} p'$$

meaning that the program performs an observable action o when moving from p to p'

Behaviours (cont.)

$\mathcal{B}(p)$ is then defined as the set of all possible strings of observable actions (traces) starting from any initial state.

In symbols:

$$\mathcal{B}(p) = \{o_0 \cdots o_k o_{k+1} \cdots \mid p \xrightarrow{o_0} \cdots \xrightarrow{o_k} p_k \xrightarrow{o_{k+1}} \dots\}$$

Correctness, not a single notion

Issue: the equality works just in special cases.

Consider again the language of expressions and the compiler on the blackboard.

What if we change the observables as follows

$$\mathcal{O} = \{\epsilon\} \cup \{\texttt{op} \mid \texttt{op} \in \{+, -, *\}\}$$

and observe each time an actual operation is performed (e.g., for debugging)?

Correctness…

Can we still consider $[\![\cdot]\!]$ correct? Indeed.

But now

$$\mathcal{B}(a) \neq \mathcal{B}([\![a]\!])$$

Why? Observables are chosen somewhat arbitrary!

Another notion of correctness

What's going on?

Our intuitive notion of correctness doesn't coincide with the formalization!

Now the compiled version has "less" behaviors, i.e.

$$\mathcal{B}(a) \supseteq \mathcal{B}(\llbracket a \rrbracket)$$

this is called *refinement*.

Finally the *real* notion of correctness?

Backward (lockstep) simulation

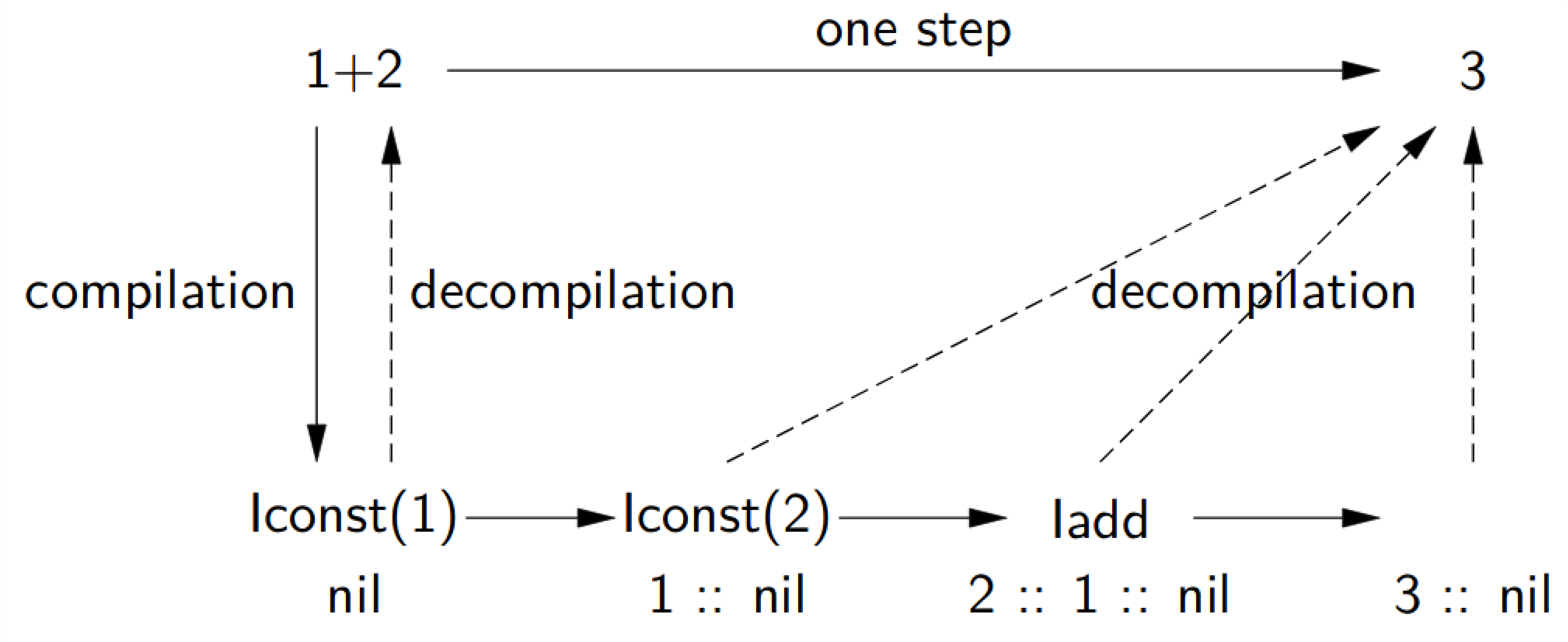
A sufficient condition for *refinement* is the existence of a *backward simulation*, i.e. a relation $\sim$ between target and source states, s.t.

1. Initial and final states are related by $\sim$;
2. If $t, \sigma_T \xrightarrow{o} t', \sigma_T'$ and $\sigma_T \sim \sigma_S$, then $(s, \sigma_S \xrightarrow{o} s', \sigma_S' \Rightarrow \sigma_T' \sim \sigma_S')$.

Pretty **hard**!

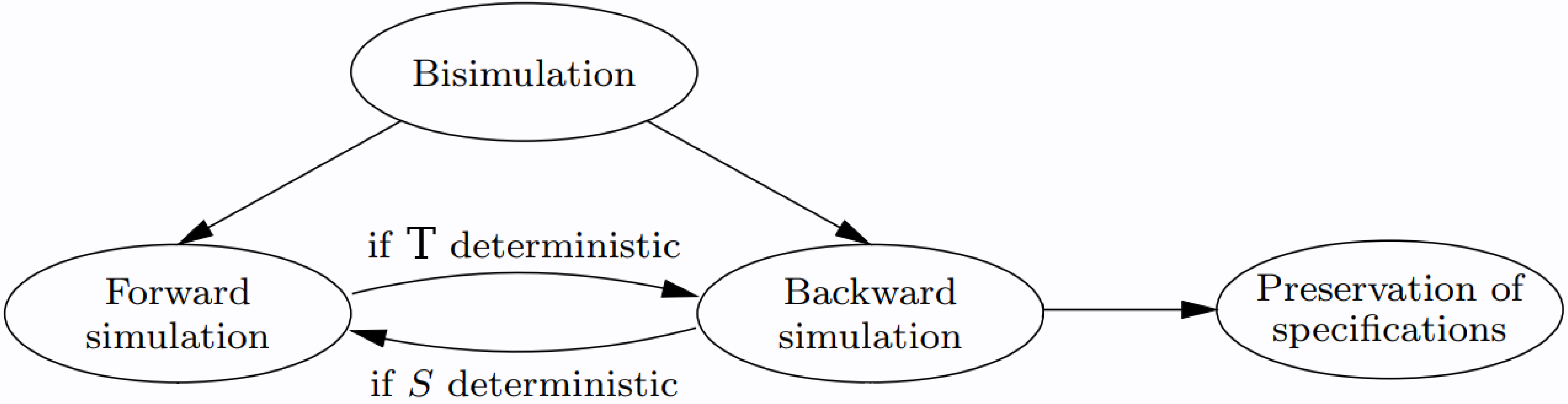
- Usually difficult to build for general languages (e.g. when considering non terminating programs)
- Especially when a single step of the source is compiled to multiple steps in the target
- Not enough in most cases (e.g. our expression compiler! :)

Example: (stuttering) backward simulation



That is: to show the existence of $\sim$ we must define a *decompilation* function!

Alternatives?



Also: stuttering (forward/backward) simulations, plus simulations, safe, …

State of the art: CompCert and CakeML

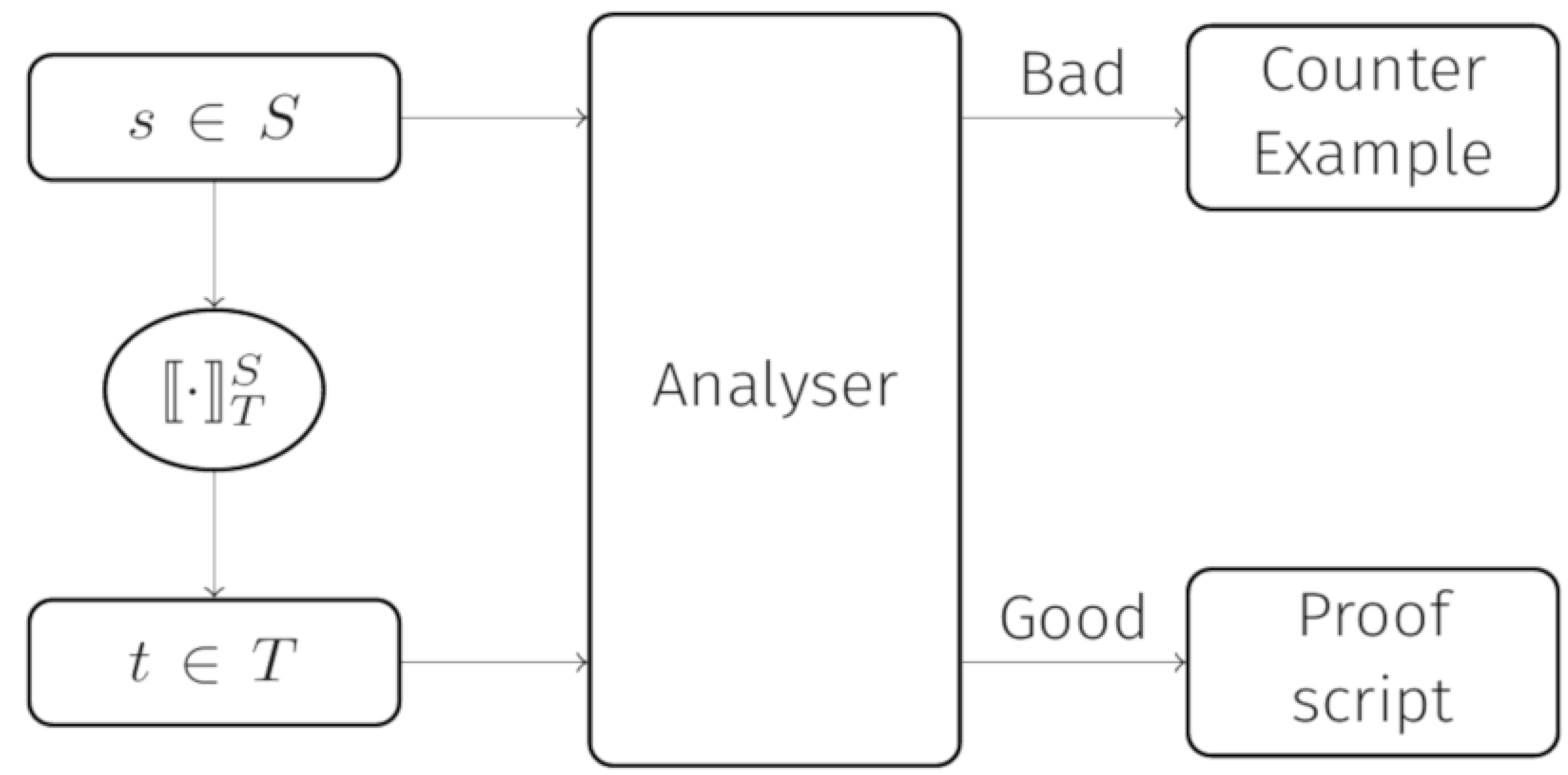
This is just *theory*, show me some real compiler!

- **CompCert:** is one of the most famous verified compilers
    - Compiles and optimizes C language to many real-world architectures
    - Fully written in Coq
    - Mechanized proof of correctness via forward simulation (enough, why? :)
    - $\mathcal{O}$: I/O and ops. on `volatile` variables
- **CakeML:** more recent
    - Compiles a subset of Standard ML
    - Bootstrapped compiler, proof mechanized in HOL4
    - $\mathcal{O}$: values of the language(s) (source, intermediate and target)

An alternative: translation validation

In this lecture, we considered an **a priori** notion of correctness.
What about considering just a *single run* of the compiler each time?

Translation validation (TV) requires this:
- Take an actual program s and compile it to $[\![s]\!]$
- Verify that *that particular* run of the compiler produced the "right" compiler



Note: this is a fully automatic process (modulo decidability!)

Beyond whole programs

- Many real-world programs are partial, i.e. they are not written as a whole by programmers
- Partial programs are made "full" by linking with a *context*
    - Contexts model external definitions from standard libraries, code written by third parties, external components, ...

Issue: All the above cannot deal with partial programs.

Beyond whole programs (cont.)

Just a glimpse of the existing solutions

1. **Separate correctness:**
    - Compile the partial source program s to $[\![s]\!]$
    - Compile the source context with the **same** compiler
    - Link them together
    - Correctness of the result is guaranteed!

Beyond whole programs (cont.)

2. **Compositional correctness:**
    - Compile the partial program s to $[\![s]\!]$
    - Choose a target context that **correctly implements** the source one
    - Link them together
    - Correctness of the result is guaranteed!

This second variant:

- is much stronger
- much more useful (think of JVM/.NET interoperability!)
- also more difficult to achieve

Summing up

- Guaranteeing the correctness of a compiler via an a priori proof
- Saw a simple example of a correct compiler for arith. expressions
    - Many issues in proving it such
    - Much more issues for (slightly) more complex languages
- However, at least two real-world compilers following this approach
- Translation validation mitigates some issues, but still not widely used

So:

- Proofs are rather involved
- Usually need a manual (or assisted, but not automatic) proof
- Still niche adoption
- Huge improvements recently!

The End

Wednesday: Is there something beyond correctness?

Bibliography

All the above material is inspired and distilled from the following papers:

[1]. "Optimization-unstable code." https://lwn.net/Articles/575563/
[2]. Xavier Leroy. "The formal verification of compilers." DeepSpec Summer School 2017. https://deepspec.org/event/dsss17/leroy-dsss17.pdf
[3]. William J. Bowman. "What even is compiler correctness?" https://www.williamjbowman.com/blog/2017/03/24/what-even-is-compiler-correctness/
[4]. Xavier Leroy. "A Formally Verified Compiler Back-end." https://link.springer.com/article/10.1007/s10817-009-9155-4
[5]. CompCert compiler. http://compcert.inria.fr/
[6]. CakeML compiler. https://cakeml.org/

Bibliography (cont.)

[7]. Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation validation." International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 1998.
[8]. George C. Necula. "Translation validation for an optimizing compiler." ACM SIGPLAN notices. Vol. 35. No. 5. ACM, 2000.