Semantyka i weryfikacja programów

Bartosz Klin

(slajdy Andrzeja Tarleckiego)

Instytut Informatyki
Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

http://www.mimuw.edu.pl/~klin pok. 5680
klin@mimuw.edu.pl

Strona tego wykładu:
http://www.mimuw.edu.pl/~klin/sem18-19.html

Program Semantics & Verification

Bartosz Klin

(slides courtesy of Andrzej Tarlecki)

Institute of Informatics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

http://www.mimuw.edu.pl/~klin office: 5680
klin@mimuw.edu.pl

This course: http://www.mimuw.edu.pl/~klin/sem18-19.html

Certified compilers

Do you trust your compiler?

- Most software errors arise from source code
- But what if the compiler itself is flawed?
- Testing is immune to this problem, since it is applied to target code

But good luck identifying the bug!

- Formal verification is harmed: even if the source program is proved correct, the compiled one may be wrong.
- Common practice for safety-critical systems:
  - turn off most optimizations
  - perform human audit of target code

It this paranoia?

- In 1995, 12 out of 20 commercially available C compilers were found to have flaws in optimizing integer division.
- In 2008, all 13 tested C compilers had flaws in dealing with volatile variables. Some GCC versions optimized this out:

```
extern volatile int WATCHDOG;
void reset_watchdog() { WATCHDOG = WATCHDOG;}
```

- CSmith: a tool for testing C compilers with randomly generated programs. In 2011, it found 325 errors in GCC, LLVM and other mainstream compilers.
- GCC shipped with Ubuntu 8.04.1 had this wrong on all optimization levels:

```
int foo(void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y; }
```

Solution I: Target code validation

After compilation, prove that the target code is equivalent to the source code.

Problems:

- Formal semantics of both source and target languages must be provided.
- Program equivalence is almost always undecidable.
- Typically needs human assistance.
- Even if it works, it is very time-consuming.

Solution II: Proof-carrying code

Augment target code with a formal proof of its desirable properties.

Advantages:

- Source code semantics is not needed
- Very robust framework, exending beyond compiler correctness
- Small burden on the user: checking proofs is not very costly
- Great for mobile code

Problems:

- Does not really check compiler correctness
- Huge burden on the developer

Solution III: Certified compiler

Formally prove that the compiler is correct.

Advantages:

- No burden on the developer or on the user
- Guarantees that source-code analyses apply to target code
- One-off effort

Problems:

- Formal semantics of both source and target languages must be provided.
- Huge burden on the compiler developer

CompCert

- A certified C compiler
- Developed since 2005 at INRIA Paris (principal: Xavier Leroy)
- Free for non-commercial use
- Licenses sold for commercial use

Main ingredients:

- Small-step operational semantics of the source language
- Small-step operational semantics of the target language
- A compiler written in (a functional sublanguage of) Coq
- A proof of correctness in Coq
- A translation from the functional sublanguage of Coq to Caml

Languages

The source language:

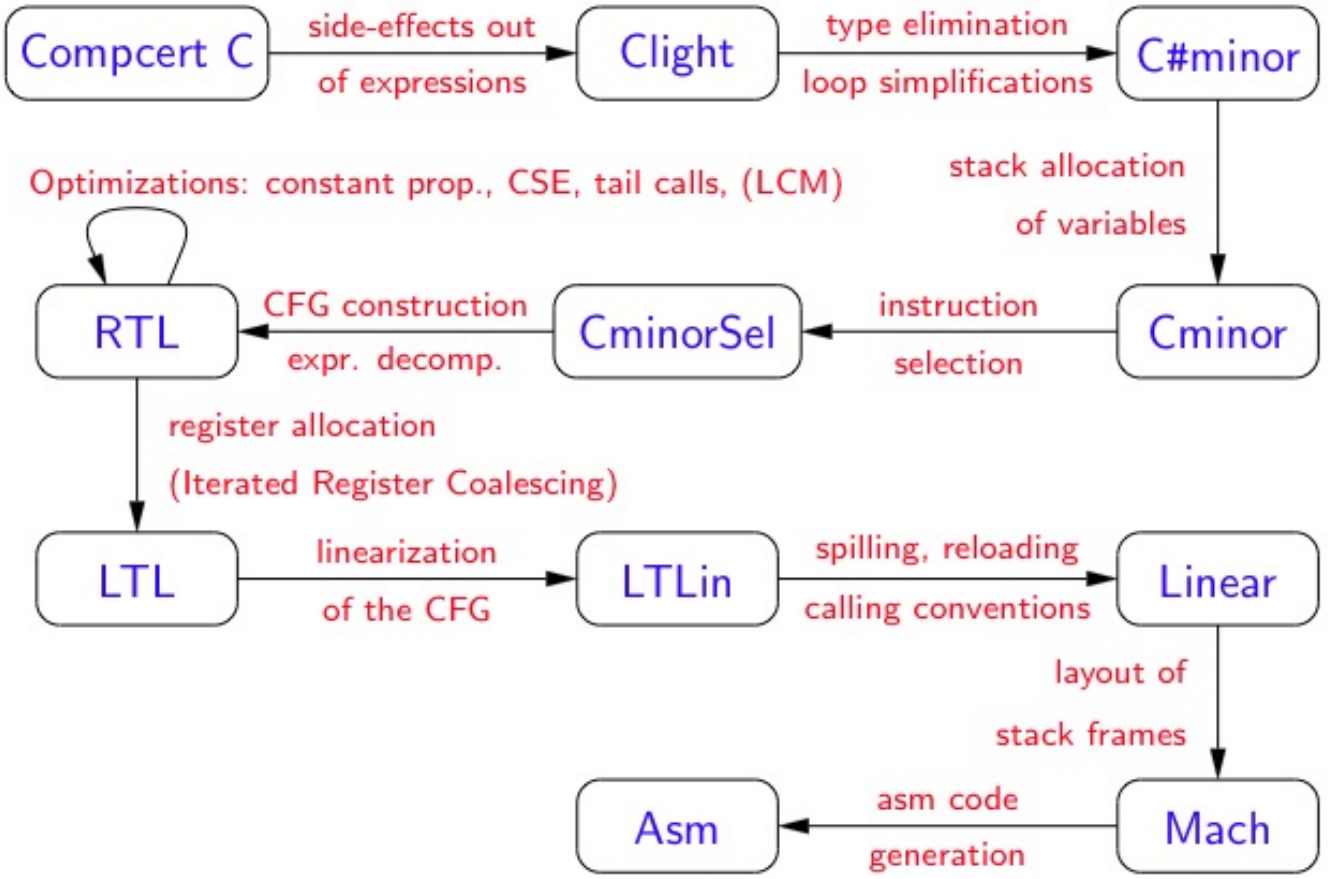
- A large subset of C
- No `longjmp` or `setjmp`
- Only structural `switch`, no "Duff's device"
- No variable-length array types

Supported target architectures:

- PowerPC
- RISC-V
- Intel x86, 32- and 64-bit

The compiler structure

- 20 passes, 11 intermediate languages, each with its own small-step operational semantics



Example intermediate language

RTL: Register Transfer Language

$$
\begin{aligned}
i ::= \quad & \texttt{nop}(l) \mid \texttt{op}(op, \vec{r}, r, l) \mid \texttt{load}(k, m, \vec{r}, r, l) \mid \texttt{store}(k, m, \vec{r}, r, l) \\
\mid \quad & \texttt{call}(sig, (r \mid id), \vec{r}, r, l) \mid \texttt{tailcall}(sig, (r \mid id), \vec{r}, r) \\
\mid \quad & \texttt{cond}(b, \vec{r}, l_t, l_f) \mid \texttt{return}(r)
\end{aligned}
$$

A CFG (Control Flow Graph) is a finite map $g : l \mapsto i$

Example semantic rule:

$$
\frac{g(l) = \texttt{op}(op, \vec{r}, r, l') \qquad \texttt{eval\_op}(G, \sigma, op, R(\vec{r})) = v}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \to \mathcal{S}(\Sigma, g, \sigma, l', R[r \mapsto v], M)}
$$

Example transformation

RTL to LTL: register allocation

- Purpose: divide pseudo-registers r into actual registers and stack allocations
- First step: back-propagation to check which r is alive in which point l
- Two pseudo-registers interfere if they are both alive at some point
- If r and r' do not interfere, they can be stored in the same register
- Coloring pseudo-registers with registers: an NP-complete problem, but good heuristics exist

Property to prove:

> *Each transition of program is "simulated"*
>
> *by transitions of the transformed program*

CompCert performance

- no errors uncovered so far (after years of attempts)
- compilation process: approx. 2 times slower than GCC with no optimization
- compiled code: approx. 10% slower than GCC with level 1 optimization, 20% slower than GCC with level 2 optimization
- main reason: lack of fancy loop optimizations etc.

What can go wrong?

Unverified parts of the compilation process:

- on the front end: preprocessing
- on the back end: assembling and linking

The verification process itself:

- What if one or both semantics are wrong?
- What if the translation from the functional sublanguage of Coq to Caml is wrong?
- What if the Caml compiler is wrong?
- What if the Coq proof system is wrong?
- What if mathematics is inconsistent?