Compiler verification
for fun and profit

Xavier Leroy

Inria Paris-Rocquencourt

FMCAD, 2014-10-22
Prologue: Can you trust your compiler?
The compilation process

General definition: any automatic translation from a computer language to another.

Restricted definition: efficient ("optimizing") translation from a source language (understandable by programmers) to a machine language (executable in hardware).

A mature area of computer science:

- Nearly 60 years old! (Fortran I: 1957)
- Huge corpus of code generation and optimization algorithms.
- Many industrial-strength compilers that perform subtle transformations.
An example of compiler optimization

Consider:

```c
double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

Compiled with the Tru64/Alpha compiler and manually decompiled back to C...
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
    L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
    L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
    L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
    L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
    L5: return dp;
    L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
    L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
    L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
    L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
    L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
    L5: return dp;
    L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
    L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
    L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
    L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
    L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
    L5: return dp;
    L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
Even unoptimized code generation is delicate

double floatofint(unsigned int i) { return (double) i; }

The PowerPC 32-bit architecture provides no instruction to convert from int to float. The compiler must therefore emulate it, as follows:

double floatofint(unsigned int i)
{
    union { double d; unsigned int x[2]; } u, v;
    u.x[0] = 0x43300000;  u.x[1] = i;
    v.x[0] = 0x43300000;  v.x[1] = 0;
    return u.d - v.d;
}

(Hint: the 64-bit integer $0x43300000 \times 2^{32} + x$ is the IEEE754 encoding of the double float $2^{52} + (\text{double})x$.)
NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated.

http://www.nullstone.com/htmls/category/divide.htm

We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables. This result is disturbing because it implies that embedded software and operating systems — both typically coded in C, both being bases for many mission-critical and safety-critical applications, and both relying on the correct translation of volatiles — may be being miscompiled.

E. Eide & J. Regehr, EMSOFT 2008
Miscompilation happens

We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 325 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs.

X. Yang, Y. Chen, E. Eide, J. Regehr, PLDI 2011
Latest sighting

[Our] new method succeeded in finding bugs in the latter five (newer) versions of GCCs, in which the previous method detected no errors.

```c
int main (void)
{
    unsigned x = 2U;
    unsigned t = ((unsigned) -(x/2)) / 2;
    assert ( t != 2147483647 );
}
```

It turned out that [the program above] caused the same error on the GCCs of versions from at least 3.1.0 through 4.7.2, regardless of targets and optimization options.

E. Nagai, A. Hashimoto, N. Ishiura, SASIMI 2013
Are miscompilation bugs a problem?

For non-critical software:
- Programmers rarely run into them.
- When they do, it’s very hard to debug.
- Globally negligible compared with bugs in the program itself.

For critical software:
- A source of concern.
- Require additional verification activities.
  (E.g. manual reviews of generated assembly code; more tests.)
- Complicate the qualification process.
- Reduce the usefulness of formal verification.
The guarantees obtained (so painfully!) by source-level formal verification may not carry over to the executable code . . .
A solution? Verified compilers

Why not formally verify the compiler itself?

After all, compilers have simple specifications:

If compilation succeeds, the generated code should behave as prescribed by the semantics of the source program.

As a corollary, we obtain:

Any safety property of the observable behavior of the source program carries over to the generated executable code.
Compiler verification for profit

In the context of high-assurance software that undergoes strict certification (DO-178 in avionics, Common Criteria in security):

- Provides strong guarantees on compilers and code generators, guarantees that are very hard to obtain by more conventional methods (tests and reviews).
- Enable the use of aggressive optimizations (which would otherwise be problematic for certification).
- Generate confidence in the results of source-level formal verifications (making it easier to derive certification credit from these verifications).
Compiler verification for fun

Compilers are challenging pieces of software from a formal verification standpoint:

- Complex data structures: abstract syntax trees, control-flow graphs.
- Complex algorithms, often recursive.
- Specifications involve formal, operational semantics for “big” languages.

Beyond the reach of automated verification techniques? (model checking, static analysis, automated deductive program provers).

A very good match for interactive theorem proving!
An old idea...
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.
CompCert: a compiler you can formally trust
The CompCert project
(X.Leroy, S.Blazy, et al)

Develop and prove correct a realistic compiler, usable for critical embedded software.

- Source language: a very large subset of C99.
- Target language: PowerPC/ARM/x86 assembly.
- Generates reasonably compact and fast code
  ⇒ careful code generation; some optimizations.

Note: compiler written from scratch, along with its proof; not trying to prove an existing compiler.
The formally verified part of the compiler

CompCert C → side-effects out of expressions → Clight → type elimination loop simplifications → C#minor

Optimizations: constant prop., CSE, inlining, tail calls

RTL → CFG construction expr. decomp. → CminorSel → instruction selection → Cminor

register allocation (IRC) calling conventions

LTL → linearization of the CFG → Linear → layout of stack frames → Mach

Linear

asm code generation

Asm x86, Asm ARM, Asm PPC
Formally verified using Coq

The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant.

Theorem transf_c_program_preservation:
  forall p tp beh,
  transf_c_program p = OK tp ->
  program_behaves (Asm.semantics tp) beh ->
  exists beh', program_behaves (Csem.semantics p) beh'
  /
  behavior_improves beh' beh.
What does semantic preservation say?

Behaviors $\text{beh} =$
- termination / divergence / crashing on an undefined behavior
- trace of I/O operations (system calls & volatile accesses)

The theorem says that the behavior of the generated code is at least as good as one of the behaviors of the source program:

Source code: $i_1.o_1.o_2.i_2.o_3$ $i_1.o_1.\dagger$ undefined behavior

Compiled code: $i_1.o_1.o_2.i_2.o_3$ $i_1.o_1.o_2\ldots$

(same behavior) ("improved" undefined behavior)

If the source code was verified to be free of undefined behaviors, we know that the compiled code behaves exactly like the source program.
Proof effort

<table>
<thead>
<tr>
<th>Code</th>
<th>Sem.</th>
<th>Claims</th>
<th>Proof scripts</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>15%</td>
<td>8%</td>
<td>17%</td>
<td>54%</td>
<td>7%</td>
</tr>
</tbody>
</table>

100,000 lines of Coq.

Including 15000 lines of “source code” (≈ 60,000 lines of Java).

6 person.years

Low proof automation (could be improved).
Programmed (mostly) in Coq

All the verified parts of the compiler are programmed directly in Coq’s specification language, using pure functional style.

- Monads to handle errors and mutable state.
- Purely functional data structures.

Coq’s extraction mechanism produces executable Caml code from these specifications.

Claim: purely functional programming is the shortest path to writing and proving a program.
The whole Compcert compiler

C source → parsing, construction of an AST → AST C
  type-checking, de-sugaring

Type reconstruction

Register allocation

Code linearization heuristics

Executable → assembling linking → Assembly
  printing of asm syntax

AST Asm

Verified compiler

Part of the TCB
Not proved
(hand-written in Caml)

Not part of the TCB
Proved in Coq
(extracted to Caml)
Performance of generated code
(On a Power 7 processor)
A tangible increase in quality

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

X. Yang, Y. Chen, E. Eide, J. Regehr, PLDI 2011
A peek under the hood: how to verify a compilation pass
Compiler verification patterns (for each pass)

Verified transformation

transformation

Validated transformation

transformation

External solver with verified validation

untrusted solver

checker

validator

= formally verified

= not verified

X. Leroy (Inria)
Verified transformation vs. verified validation

Verified validation: usually less to prove; sound; may fail at compile-time. Verified transformation: usually more to prove; sound; complete.

Example: register allocation via graph coloring.

\[
\begin{align*}
a & = i \ll 2 \\
b & = \text{load}(t+a) \\
c & = \text{float}(b) \\
s & = s + c
\end{align*}
\]

\[
\begin{align*}
R3 & = R2 \ll 2 \\
R3 & = \text{load}(R1+R3) \\
F1 & = \text{float}(R3) \\
F2 & = \text{reload}(SP+16) \\
F2 & = F2 + F1 \\
\text{spill}(F2, SP+16)
\end{align*}
\]
The verified-validated continuum

Not proved

Proved in Coq

(May fail)

Graph coloring
(IRC algorithm)

Checker for colorings

Liveness analysis
Construction of interference graph
Code rewriting

2600 LOC

(Proved in Coq)

(Not proved)

2600 LOC

(Proved in Coq)

(Not proved)
The verified-validated continuum

Not proved

Graph coloring (IRC algorithm)

(May fail)

Liveness analysis
Construction of interference graph
Code rewriting

2600 LOC

Checker for colorings

(Leroy, JAR 2009)

Proved in Coq

Fully verified implementation of IRC

Liveness analysis
Construction of interference graph
Code rewriting

12000 LOC

(Blazy, Robillard, Appel 2010)
### The verified-validated continuum

<table>
<thead>
<tr>
<th>Proved in Coq</th>
<th>Not proved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation validation via sets ( { \text{var} = \text{loc} } )</td>
<td>Any register allocator incl. spilling and live-range splitting</td>
</tr>
<tr>
<td>800 LOC</td>
<td>(May fail)</td>
</tr>
<tr>
<td>(Rideau and Leroy 2010)</td>
<td>(May fail)</td>
</tr>
</tbody>
</table>

#### (Total)
- Fully verified implementation of IRC
- Liveness analysis
- Construction of interference graph
- Code rewriting
- Liveness analysis
- Construction of interference graph
- Code rewriting
- 12000 LOC

#### Graph coloring (IRC algorithm)
- Checker for colorings
- 2600 LOC

#### (Total!)
- Defensive coloring engine
- Liveness analysis
- Construction of interference graph
- Code rewriting
- Elimination order
- Coalescing decisions (90% of IRC)
- 2800 LOC

### Translation validation via sets
- \( \{ \text{var} = \text{loc} \} \)

### Code rewriting
- (Rideau and Leroy 2010)
- (Blazy, Robillard, Appel 2010)
### The verified-validated continuum

<table>
<thead>
<tr>
<th>Not proved</th>
<th>Proved in Coq</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Any register allocator incl. spilling and live-range splitting</td>
<td>Translation validation via sets {var = loc}</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(May fail)</th>
<th>(May fail)</th>
<th>(Total!)</th>
<th>(Total)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph coloring (IRC algorithm)</td>
<td>Checker for colorings</td>
<td>Elimination order Coalescing decisions (90% of IRC)</td>
<td>Fully verified implementation of IRC</td>
</tr>
<tr>
<td>2600 LOC</td>
<td>2800 LOC</td>
<td>12000 LOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liveness analysis Construction of interference graph Code rewriting</td>
<td>Liveness analysis Construction of interference graph Code rewriting</td>
<td>Liveness analysis Construction of interference graph Code rewriting</td>
<td>(Blazy, Robillard, Appel 2010)</td>
</tr>
</tbody>
</table>

(Rideau and Leroy 2010) (Leroy, JAR 2009) (Total!) (Blazy, Robillard, Appel 2010)
Validating register allocation a posteriori
(Silvain Rideau & Xavier Leroy, Compiler Construction 2010)

For each program point $p$, infer and check the consistency of a set of equations $E(p)$ between variables and locations (*):

$$E(p) = \{x_1 = \ell_1; \ldots; x_n = \ell_n\}$$

Intuition: in every execution of the original code and the transformed code, the current value of $\ell_i$ at $p$ is the same as that of $x_i$ at $p$.

(*) locations $=$ processor registers $\cup$ stack slots.
Forward analysis

Assume given a set of equations \( \text{BEFORE} \) that holds “before” point \( p \). Check that \( \{x_2 = \ell_2\} \in \text{BEFORE} \) and \( \{x_3 = \ell_3\} \in \text{BEFORE} \).

Compute set of equations \( \text{AFTER} \) that holds “after” points \( p \):

- Remove all equations \( x = \ell \) such that \( x = x_1 \) or \( \ell \) overlaps with \( \ell_1 \).
- Add equation \( x_1 = \ell_1 \)
Alternative: backward analysis

Assume given a set of equations \textbf{AFTER} that must hold “after” point \( p \) for the rest of the executions to behave identically.

Check that \textbf{AFTER} contains no equations \( x = \ell \) such that \((x, \ell) \neq (x_1, \ell_1)\) and \((x = x_1 \text{ or } \ell \text{ overlaps with } \ell_1)\).
(These equations cannot be satisfied in general.)

Compute set of equations \textbf{BEFORE} that must hold “before” point \( p \) for the rest of the executions to behave identically:

- Remove the equation \( x_1 = \ell_1 \) if present.
- Add equations \( x_2 = \ell_2 \) and \( x_3 = \ell_3 \).
Comparing backward and forward approaches

If we project the sets of equations \( \{x_i = \ell_i\} \) on one side, say \( \{x_i\} \):

Equations inferred:   Forward approach \( \supseteq \) Backward approach

Projections:       Reaching definitions \( \supseteq \) Live variables

In general, the backward approach is more efficient because it produces smaller sets of equations.
Backward equations for coalesced copies

Assume given a set of equations \textsc{AFTER} that must hold “after” point $p$ for the rest of the executions to behave identically.

The set \textsc{BEFORE} of equations that must hold “before” is

\[
\begin{align*}
\{ (x_2 = \ell) \mid (x_1 = \ell) \in \textsc{AFTER} \} \\
\cup \\
\{ (x = \ell) \mid (x = \ell) \in \textsc{AFTER} \text{ and } x \neq x_1 \}
\end{align*}
\]
Backward equations for inserted moves

Check that AFTER contains no equation $x = l$ with $l \neq l_1$ and $l$ overlaps $l_1$.

The set BEFORE of equations that must hold “before” is

\[
\{(x = l_2) \mid (x = l_1) \in \text{AFTER}\} \\
\cup \{(x = l) \mid (x = l) \in \text{AFTER} \text{ and } l \neq l_1\}
\]
The validation algorithm

\[
\text{check\_function}(f, f') = \\
\text{compute the solutions } E(p) \text{ of the dataflow equations} \\
E(p) = \bigcup \{\text{transfer}(f, f', s', E(s')) \mid s' \text{ successor of } p \text{ in } f\} \\
\text{let } E_0 = \text{transfer}(f, f', f'.\text{entrypoint}, E(f'.\text{entrypoint})) \\
\text{check } E_0 \neq \top \text{ and} \\
E_0 \cap f.\text{params} \subseteq \{f.\text{params} = \text{parameters}(f'.\text{typesig})\}
\]
Soundness proof

Theorem: if \( \text{check\_function}(f, f') = \text{true} \), the transformed function \( f' \) behaves at run-time exactly like \( f \).

The proof builds on a forward simulation diagram:

\[
\begin{align*}
(p_1, e_1, m_1) \quad &\xrightarrow{e_1, e_1' \models \text{BEFORE}(p_1)} \quad (p_1, e_1', m_1) \\
\downarrow & \hspace{2cm} + \\
(p_2, e_2, m_2) \quad &\xrightarrow{e_2, e_2' \models \text{BEFORE}(p_2)} \quad (p_2, e_2', m_2)
\end{align*}
\]

Satisfaction of a set \( E \) of equations by a state \( e : \text{variable} \rightarrow \text{value} \) and a state \( e' : \text{location} \rightarrow \text{value} \):

\[
e, e' \models E \overset{\text{def}}{=} \forall (x = \ell) \in E, \ x \in \text{Dom}(e) \implies e(x) = e'(\ell)
\]
Semantic preservation for whole executions

\[
\begin{align*}
\text{(initial state)} & \quad S_1 \quad \text{invariant} \quad T_1 \quad \text{(initial state)} \\
S_2 & \quad \epsilon \quad S_2 \quad \text{invariant} \quad \epsilon \\
S_3 & \quad \nu_1 \quad S_3 \quad \text{invariant} \quad T_2 \quad \nu_1 \\
S_4 & \quad \nu_2 \quad S_4 \quad \text{invariant} \quad T_3 \quad \nu_2 \\
S_5 & \quad \epsilon \quad S_5 \quad \text{invariant} \quad T_4 \\
\text{(final state)} & \quad S_5 \quad \text{invariant} \quad T_5 \quad \text{(final state)}
\end{align*}
\]

Proves that the original program and the transformed program have the same behavior (the trace \( t = \nu_1.\nu_2 \)).
Towards other source languages
Verified compilation of various languages

High-level, garbage-collected languages (Java, C#, functional)

Scripting languages (Javascript)

C++

Domain-specific languages (hardware description, synchronous/reactive, query languages, etc)

C: the *lingua franca* of systems programming
- low-level semantics with many dark corners
+ relatively simple compilation (but: optimization is difficult)
+ no run-time system
Verified compilation of various languages

High-level, garbage-collected languages (Java, C#, functional)

Scripting languages (Javascript)

C++:
- all the dark corners of C plus a complex object model
+ C-like compilation
- a bit of a run-time system (exceptions)

Domain-specific languages (hardware description, synchronous/reactive, query languages, etc)
Verified compilation of various languages

High-level, garbage-collected languages (Java, C#, functional)

Scripting languages (Javascript)

C

C++

Domain-specific languages (hardware description, synchronous/reactive, query languages, etc)

High-level garbage-collected languages:
+ clean semantics
+ nontrivial but interesting compilation
- large run-time system (allocation, GC, exceptions, . . . )
Verified compilation of various languages

- High-level, garbage-collected languages (Java, C#, functional)
- C
- C++

- Domain-specific languages (hardware description, synchronous/reactive, query languages, etc)

Scripting languages:
- obscure semantics
- not designed for compilation
- very large run-time system (GC + DOM + ...)

X. Leroy (Inria)
Verified compilation of various languages

- High-level, garbage-collected languages (Java, C#, functional)
- Scripting languages (Javascript)
- Domain-specific languages (hardware description, synchronous/reactive, query languages, etc)

Domain-specific languages with limited expressiveness:
- clean semantics
- opportunities for superoptimization & synthesis
- no run-time system
- used in critical embedded systems (e.g. Scade, Simulink)
FeSi (Featherweight Synthesis): verified hardware synthesis
(Thomas Braibant & Adam Chlipala, CAV’13)

A simple, declarative hardware description language in the style of Lava and Bluespec.

Oriented towards the description and proof of parameterized circuits (e.g. $n$-bit multiplier for all $n$).

Embedded within Coq $\rightarrow$ dependent types, recursion, . . .

Simple but nontrivial synthesis of RTL circuits, verified in Coq.

\[
\begin{align*}
\text{Coq functions} & \quad \text{evaluation} \quad \text{FeSi AST} \quad \text{synthesis} \quad \text{RTL} \\
\uparrow \quad \text{Coq proof} (\forall n) & \quad \text{semantic equivalence}
\end{align*}
\]
A taste of FeSi: $n$-bit carry-lookaheads ahead (simplified)

Fixpoint add \{\Phi\} n (x : expr V (Tint [2^n])) (y : expr V (Tint [2^n])) :=
    match n with
    | 0 =>
      ret [tuple ((x = #i 1) \lor (y = #i 1)), (* propagated carry *)
          ((x = #i 1) \&\& (y = #i 1)), (* generated carry *)
          x + y, (* sum if no carry-in *)
          x + y + #i 1 ] (* sum if carry-in *)
    | S n =>
      do xL <- low x; do xH <- high x; do yL <- low y; do yH <- high y;
      do rL <- add n xL yL; do rH <- add n xH yH;
      do (pL, gL, sL, tL) <- rL; do (pH, gH, sH, tH) <- rH;
      do sH' <- (Emux (gL) (tH) (sH));
      do tH' <- (Emux (pL) (tH) (sH));
      do pH' <- (gH \lor (pH \&\& pL));
      do gH' <- (gH \lor (pH \&\& gL));
      ret [tuple pH', gH', combineLH sL sH', combineLH tL tH']
    end

Note: dependent types + recursion + circuit generation.
A taste of FeSi: \(n\)-bit carry-lookahead adder (simplified)

Fixpoint add \(\Phi\) \(n\) (\(x: expr \text{ V } (\text{Tint } [2^n])\)) (\(y: expr \text{ V } (\text{Tint } [2^n])\)) : action \(\Phi\) \(\text{ V } (\text{Ttuple } [\text{Tbool; Tbool; Tint } [2^n]; \text{Tint } [2^n]])\) :=

\[
\begin{align*}
\text{match } n \text{ with} \\
| 0 &\Rightarrow \\
&\text{ret } \left[ \text{tuple } ((x = \#i 1) \text{ || } (y = \#i 1)), \text{ (* propagated carry *)} \right. \\
&\text{((x = \#i 1) \&\& (y = \#i 1)), \text{ (* generated carry *)}} \\
&x + y, \text{ (* sum if no carry-in *)} \\
&x + y + \#i 1 \text{ ]} \text{ (* sum if carry-in *)} \\
| \text{S } n &\Rightarrow \\
&\text{do } xL \leftarrow \text{ low } x; \text{ do } xH \leftarrow \text{ high } x; \text{ do } yL \leftarrow \text{ low } y; \text{ do } yH \leftarrow \text{ high } y; \\
&\text{do } rL \leftarrow \text{ add } n \text{ xL yL}; \text{ do } rH \leftarrow \text{ add } n \text{ xH yH}; \\
&\text{do } (pL, gL, sL, tL) \leftarrow rL; \text{ do } (pH, gH, sH, tH) \leftarrow rH; \\
&\text{do } sH' \leftarrow (\text{Emux } (gL) (tH) (sH)); \\
&\text{do } tH' \leftarrow (\text{Emux } (pL) (tH) (sH)); \\
&\text{do } pH' \leftarrow (\text{gH } || \text{ (pH } \&\& \text{ pL)}); \\
&\text{do } gH' \leftarrow (\text{gH } || \text{ (pH } \&\& \text{ gL)}); \\
&\text{ret } \left[ \text{tuple } pH', gH', \text{ combineLH } sL sH', \text{ combineLH } tL tH' \right]
\end{align*}
\]

end

Note: dependent types + recursion + circuit generation.
A taste of FeSi: $n$-bit carry-lookahead adder (simplified)

Fixpoint add \( \{\Phi\} \) \( n \) \((x : \text{expr} V (\text{Tint}[2^n]))\) \((y : \text{expr} V (\text{Tint}[2^n]))\) : action \( \Phi \) \( V (\text{Ttuple} [\text{Tbool}; \text{Tbool}; \text{Tint}[2^n]; \text{Tint}[2^n]]) \) :=

\[
\begin{align*}
&\text{match } n \text{ with} \\
&| 0 => \\
&\text{ret } \text{[tuple } ((x = #i 1) || (y = #i 1)), (* \text{propagated carry} *) \((x = #i 1) && (y = #i 1)), (* \text{generated carry} *) \]
&\text{x + y}, (* \text{sum if no carry-in} *) \\
&\text{x + y + #i 1 }] (* \text{sum if carry-in} *) \\
&| S n => \\
&\text{do } xL \leftarrow \text{low } x; \text{do } xH \leftarrow \text{high } x; \text{do } yL \leftarrow \text{low } y; \text{do } yH \leftarrow \text{high } y; \\
&\text{do } rL \leftarrow \text{add } n \text{ xL yL}; \text{do } rH \leftarrow \text{add } n \text{ xH yH}; \\
&\text{do } (pL, gL, sL, tL) \leftarrow rL; \text{do } (pH, gH, sH, tH) \leftarrow rH; \\
&\text{do } sH' \leftarrow (\text{Emux } (gL) (tH) (sH)); \\
&\text{do } tH' \leftarrow (\text{Emux } (pL) (tH) (sH)); \\
&\text{do } pH' \leftarrow (gH || (pH && pL)); \\
&\text{do } gH' \leftarrow (gH || (pH && gL)); \\
&\text{ret } \text{[tuple } pH', gH', \text{combineLH } sL sH', \text{combineLH } tL tH'] \\
\end{align*}
\]

Note: dependent types + recursion + circuit generation.
A taste of FeSi: $n$-bit carry-lookahead adder (simplified)

Fixpoint add $\{\Phi\}$ $n$ ($x : expr$ $V$ (Tint $[2^n]$)) ($y : expr$ $V$ (Tint $[2^n]$)) $: action$ $\Phi$ $V$ (Ttuple [Tbool; Tbool; Tint $[2^n]$; Tint $[2^n]$]) :=
match $n$ with
| $0$ =>
  ret $[tuple ((x = #i 1) || (y = #i 1)), (* propagated carry *)$
  $((x = #i 1) && (y = #i 1)), (* generated carry *)$
  $x + y, (* sum if no carry-in *)$
  $x + y + #i 1 ] (* sum if carry-in *)$
| $S$ $n$ =>
do $xL$ $<$~ low $x$; do $xH$ $<$~ high $x$; do $yL$ $<$~ low $y$; do $yH$ $<$~ high $y$;
do $rL$ $<- add n xL yL$; do $rH$ $<- add n xH yH$;
do ($pL$, $gL$, $sL$, $tL$) $<$~ $rL$; do ($pH$, $gH$, $sH$, $tH$) $<$~ $rH$;
do $sH'$ $<$~ (Emux ($gL$) ($tH$) ($sH$));
do $tH'$ $<$~ (Emux ($pL$) ($tH$) ($sH$));
do $pH'$ $<$~ ($gH$ || ($pH$ && $pL$));
do $gH'$ $<$~ ($gH$ || ($pH$ && $gL$));
ret $[tuple pH', gH', combineLH sL sH', combineLH tL tH']$
end

Note: dependent types + recursion + circuit generation.
FeSi internal representation

A type of expressions (= combinatorial circuits) ...

Inductive expr : ty → Type :=
  (* Input wires *)
  | Evar : ∀ t, V t → expr t
  (* Operations on Booleans *)
  | Eandb : expr B → expr B → expr B | ...
  (* Operations on n-bit integers *)
  | Eadd : ∀ n, expr (Int n) → expr (Int n) → expr (Int n) | ...
  (* Operations on tuples *)
  | Efst : ∀ l t, expr (Tuple (t:: l)) → expr t | ...
FeSi internal representation

... and a type of actions (≡ sequential circuits).

Inductive action: \( ty \rightarrow Type := \)
  \( | \ Return: \ \forall \ t, expr \ t \rightarrow action \ t \)
  (* Connecting two actions via a wire *)
  \( | \ Bind: \ \forall \ t \ u, action \ t \rightarrow (V \ t \rightarrow action \ u) \rightarrow action \ u \)
  (* Guards (control flow) *)
  \( | \ Assert: expr \ B \rightarrow action \ Unit \)
  \( | \ OrElse: \ \forall \ t, action \ t \rightarrow action \ t \rightarrow action \ t \)
  (* Operations on registers *)
  \( | \ RegRead : \ \forall \ t, member \ \Phi (Reg \ t) \rightarrow action \ t \)
  \( | \ RegWrite: \ \forall \ t, member \ \Phi (Reg \ t) \rightarrow expr \ t \rightarrow action \ Unit \)

High-level semantics: close to that of a functional language.
Register writes are batched and performed at end of cycle.
The semantics of an action is a state transformer

\[ \text{state at beginning of cycle} \rightarrow \text{state at beginning of next cycle} \]
Compiling FeSi to RTL

A simple 4-stage compiler with a few optimizations:

1. Normalization: give names to intermediate results.
2. Transform control-flow into data-flow; synthesize write-enable signals for register updates.
3. Syntactic common subexpression elimination.
4. BDD-based reduction of Boolean expressions.
Putting it all together

The FeSi compilation pipeline and its correctness statement:

Variable \( (\Phi: \text{list mem}) \) \( (t: ty) \).

Definition \( \text{fesic} \ (A: \text{Fesi. Action} \ \Phi \ t) : \text{RTL.Block} \ \Phi \ t := \)
  \[
  \begin{aligned}
  &\text{let } x := \text{IR. Compile} \ \Phi \ t \ a \ \text{in} \\
  &\text{let } x := \text{RTL.Compile} \ \Phi \ t \ x \ \text{in} \\
  &\text{let } x := \text{CSE.Compile} \ \Phi \ t \ x \ \text{in} \\
  &\text{BDD.Compile} \ \Phi \ t \ x.
  \end{aligned}
  \]

Theorem \( \text{fesic_correct} : \)
  \[
  \forall \ A \ (\Gamma : \Phi \ ), \ \text{Front.Next} \ \Gamma \ A = \text{RTL.Next} \ \Gamma \ (\text{fesic} \ A).
  \]
In closing...
Current status

At this stage of the CompCert experiment, the initial goal — proving correct a nontrivial compiler — appears feasible.

(Within the limitations of today’s proof assistants such as Coq.)

Towards industrialization (partnership with AbsInt Gmbh).
Some directions for future work

- More optimizations
- "Bootstrap" (proved extraction)
- Shared-memory concurrency
- More assurance
- Connections w/ hardware verification
- Verifying program provers & static analyzers
- Other source languages

Other source languages besides C (already discussed).
Some directions for future work

- More assurance
- Verifying program provers & static analyzers
- Other source languages
- Connections w/ hardware verification
- Shared-memory concurrency
- "Bootstrap" (proved extraction)
- More optimizations

Prove or validate more of the TCB: lexing, typing, elaboration, assembling, linking, ...
Add advanced optimizations, esp. loop optimizations. Verified validation as the approach of least resistance.
Some directions for future work

- More optimizations
- "Bootstrap" (proved extraction)
- More assurance
- Other source languages
- Verifying program provers & static analyzers
- Connections w/ hardware verification
- Shared-memory concurrency

Increase confidence in the tools used to build CompCert: Coq’s extraction facility + the Caml compiler.
Some directions for future work

- More optimizations
- "Bootstrap" (proved extraction)
- More assurance
- Verifying program provers & static analyzers
- Other source languages
- Connections w/ hardware verification
- Shared-memory concurrency

Race-free programs + concurrent separation logic (A. Appel et al)
or: racy programs + hardware memory models (P. Sewell et al).
Some directions for future work

- More optimizations
- "Bootstrap" (proved extraction)
- More assurance
- Other source languages
- Verifying program provers & static analyzers
- Connections w/ hardware verification
- Shared-memory concurrency

Formal specs for architectures & instruction sets, as the missing link between compiler verification and hardware verification.
Some directions for future work

- More optimizations
- "Bootstrap" (proved extraction)
- Shared-memory concurrency
- More assurance
- Connections w/ hardware verification
- Other source languages
- Verifying program provers & static analyzers

The Verasco project: formal verification of a static analyzer based on abstract interpretation. (Inria, Verimag, Airbus).
In closing... 

Critical software deserves the most trustworthy tools that computer science can provide.

The formal verification of development and verification tools for critical software

- appears within reach,
- raises fascinating verification issues,
- improves our understanding of the algorithms involved,
- and could have practical impact.
For more information

http://compcert.inria.fr/

Research papers.

Complete source & proofs available for evaluation and research purposes.

Compiler runs on / produces code for
\{Linux, MacOSX, Windows+cygwin\} / \{PowerPC, ARM, x86\}.