The CompCert verified compiler

Tahina Ramananandro$^1$

$^1$Yale University

November 28th, 2012
Outline

1. General overview: compiler correctness
2. The CompCert compilation passes in detail
3. Proof techniques (NOT part of the exam)
4. Conclusion
Outline

1. General overview: compiler correctness
   - Motivation: software verification
   - Can you trust your compiler?
   - Compiler correctness
   - The architecture of CompCert
   - Live Demo!

2. The CompCert compilation passes in detail

3. Proof techniques (NOT part of the exam)

4. Conclusion
Software errors

- Software is ubiquitous
- Software errors (bugs, failures) mostly with limited effects...
Software errors

- Software is ubiquitous
- Software errors (bugs, failures) mostly with limited effects...
- ...except in specific areas of **critical software**, where the slightest bug can lead to dramatic consequences:
  - medical devices
  - transportation (space, avionics, railways)
  - military applications
Therac 25 radiotherapy machine (1985): at least 6 patients dead due to software activating wrong radiation mode
Ariane 5 maiden flight (1996): US$370 million lost material and project delayed by 4 years due to overflow in floating-point computations.
Ariane 5 maiden flight (1996): US$370 million lost material and project delayed by 4 years due to overflow in floating-point computations.
Trusted software

- Software more and more present in critical systems
- Need highest quality
- Need to be trusted
Software testing

Usual approach in industry: Testing and manual code reviews

- Required in avionics by DO-178B official regulations
- Software errors caused no casualties so far in avionics
- All cases covered?
- Costs?
Scalability of software testing?
Formal verification of software

A complementary approach: software verification by formal methods: model-checking, abstract interpretation, deductive verification, automated program generation. . .
Formal verification of software

A complementary approach: software verification by formal methods:
model-checking, abstract interpretation, deductive verification, automated program

generation... 

- Stronger guarantees
- Exhaustive: all behaviours taken into account
- No need to run the software
- Solid mathematical backgrounds
Formal verification of software

Program

```c
int main () {
    int x = 21;
    return x+x;
}
```

Specification

- The program terminates
- The program is not interrupted by an error
- If the program terminates, then it returns 42
Formal verification of software

Program

```c
int main () {
    int x = 21;
    return x+x;
}
```

⇝

Specification

- The program terminates
- The program is not interrupted by an error
- If the program terminates, then it returns 42
Programs are verified at their source level...
... but what about the compiled code?

Source code  \[\rightarrow\]  Compilation  \[\rightarrow\]  Machine-executable code

Proof that the program meets its specification
The compiler must be trusted as well!

Source code → Compilation → Machine-executable code

Proof that the program meets its specification

Proof that the program meets its specification?
1. General overview: compiler correctness
   - Motivation: software verification
   - Can you trust your compiler?
   - Compiler correctness
   - The architecture of CompCert
   - Live Demo!

2. The CompCert compilation passes in detail

3. Proof techniques (NOT part of the exam)

4. Conclusion
Compiler bugs can impact production-level programs

Released just today, Java 7 includes hotspot-compiler optimizations that miscompile certain loops, potentially affecting projects such as Apache Lucene Core, Apache Solr, and possibly others (...)
At best, the bugs only cause JVMs to crash; in other cases, they result in miscalculations that can lead to application bugginess.

JavaWorld, July 29, 2011
Compiler bugs can be found by intensive testing

Yang et al., PLDI 2011

- Csmith, tool generating random C programs
- $>325$ previously unknown C compiler bugs in all commercial and open-source real-world compilers (including Intel, GCC, Clang, etc.)
- Bugs confirmed (wrong code silently generated)
How to trust the compiler?

- Manual review of a compiler is not realistic
- Testing can only help discover bugs, not realistic to ensure that a compiler is correct
- So, enable simultaneous manual review of source and compiled code
- Disable any optimization
So why not formally verify the compiler itself?

- After all, compilers have a simple specification:
  
  If compilation succeeds, then the generated program behaves as prescribed by the semantics of the source program.

- This is the goal of CompCert.
Not a new idea

John McCarthy
James Painter

CORRECTNESS OF A COMPILER
FOR ARITHMETIC EXPRESSIONS

1. Introduction. This paper contains a proof of the correctness of a simple
compiling algorithm for compiling arithmetic expressions into machine
language.

The definition of correctness, the formalism used to express the description
of source language, object language and compiler, and the methods
of proof are all intended to serve as prototypes for the more complicated
task of proving the correctness of usable compilers. The ultimate goal,
as outlined in references [1], [2], [3] and [4] is to make it possible to use
a computer to check proofs that compilers are correct.

Mathematical Aspects of Computer Science, 1967
Even mechanized compiler correctness proofs are not new...

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
and proof scripts look somewhat familiar!
Outline

1. General overview: compiler correctness
   - Motivation: software verification
   - Can you trust your compiler?
   - Compiler correctness
   - The architecture of CompCert
   - Live Demo!

2. The CompCert compilation passes in detail

3. Proof techniques (NOT part of the exam)

4. Conclusion
What does it mean for a compiler to be correct?

Apply formal methods to the compiler itself to prove the following:

**Theorem (Semantic preservation)**

*For all source code* $S$,

*if the compiler generates machine code* $C$ *from source* $S$ *without reporting any compilation error,*

*then* $C$ *behaves like* $S$.

Note: a compiler is allowed to crash without producing any code (e.g. capacity exceeded).
What does it mean for a compiler to be correct?

Apply formal methods to the compiler itself to prove the following:

**Theorem (Semantic preservation)**

*For all source code $S$, if the compiler generates machine code $C$ from source $S$ without reporting any compilation error, then $C$ behaves like $S.*

Note: a compiler is allowed to crash without producing any code (e.g. capacity exceeded).
What does it mean for a compiler to be correct?

Apply formal methods to the compiler itself to prove the following:

**Theorem (Semantic preservation)**

For all source code $S$, if the compiler generates machine code $C$ from source $S$ without reporting any compilation error, then $C$ behaves like $S$.

Note: a compiler is allowed to crash without producing any code (e.g. capacity exceeded).
Observable program behavior

- A program may:
  - Terminate and return a result value
  - Diverge (= indefinitely run, e.g. infinite loop)
  - Go wrong (= crash, e.g. division by zero, or invalid memory access)
Observable program behavior

- A program may:
  - Terminate and return a result value
  - Diverge (= indefinitely run, e.g. infinite loop)
  - Go wrong (= crash, e.g. division by zero, or invalid memory access)

- During its execution, it can also produce *events* (e.g. user input, screen output, network communications, etc.). The sequence (or *trace*) of events can be:
  - Finite, in the case of terminating or “going wrong” behaviors
  - Finite or infinite, in the case of diverging behaviors. In the infinite case, the program *reacts*.

- This follows the ISO C99 notion of observable behavior.
Observed program behavior (contd.)

\[ \text{Beh} ::= \text{Terminates}(res; e_1, e_2, \ldots, e_n) \]
\[ \quad | \quad \text{GoesWrong}(e_1, e_2, \ldots, e_n) \]
\[ \quad | \quad \text{Diverges}(e_1, e_2, \ldots, e_n) \]
\[ \quad | \quad \text{Reacts}(e_1, e_2, \ldots) \]
Observable program behavior (contd.)

\[
\text{Beh} \quad ::= \quad \text{Terminates}(\text{res}; e_1, e_2, \ldots, e_n) \\
| \quad \text{GoesWrong}(e_1, e_2, \ldots, e_n) \\
| \quad \text{Diverges}(e_1, e_2, \ldots, e_n) \\
| \quad \text{Reacts}(e_1, e_2, \ldots)
\]

- The \textit{semantics} of the language gives to each program \( P \) written in this language, the nonempty set of its observable behaviors \( \mathcal{B}(P) \subseteq \text{Beh} \)
- \( \mathcal{B}(P) \) can have:
  - One element: \( P \) is deterministic
  - Several elements: \( P \) is nondeterministic
- \text{Beh} is common to all studied programming languages
Examples

```c
unsigned int x = 0;
while (1) {
    if (x < 3) {
        printf("%d\n", x);
        x++;
    }
}
```

//

Diverges(print(0), print(1), print(2))
Examples

unsigned int x = 0;
while (1) {
    if (x < 3) {
        printf("%d\n", x);
        x++;
    }
}

// Diverges(print(0), print(1), print(2))

unsigned int x = 0;
while (1) {
    printf("%d\n", x);
    x++;
}

// Reacts(print(0), print(1), print(2), ..., print(0), print(1), print(2), ...)

Ramananandro (Yale)
Program verification

- A *functional specification* $\text{Spec}$ is a predicate over program behaviors. A program behavior $b \in \text{Beh}$ satisfies $\text{Spec}$ if, and only if, $\text{Spec}(b)$ holds.
- A program $P$ satisfies the specification $\text{Spec}$ ($P \models \text{Spec}$) if, and only if:
  \[ \forall b \in \mathcal{B}(P), \text{Spec}(b) \]
- Examples: program safety, program termination, etc.
A **functional specification** Spec is a predicate over program behaviors. A program behavior $b \in \text{Beh}$ satisfies Spec if, and only if, $\text{Spec}(b)$ holds.

A program $P$ satisfies the specification $\text{Spec}$ ($P \models \text{Spec}$) if, and only if:

$$\forall b \in \mathcal{B}(P), \text{Spec}(b)$$

**Definition (Semantics preservation by program refinement)**

A compiler $C$ is correct if, and only if, for all source code $P$ such that the compiled code $C(P)$ exists:

$$\forall \text{Spec}, P \models \text{Spec} \Rightarrow C(P) \models \text{Spec}$$

which is equivalent to (**program refinement**)

$$\mathcal{B}(C(P)) \subseteq \mathcal{B}(P)$$

In particular, a correct compiler preserves safety.
Observational equivalence

- If the source language is deterministic, then a compiler $C$ is correct if, and only if, $B(C(P)) = B(P)$.

  Consider for instance the C language:

  ```
  int x = 0;
  int f(void) { x = x + 1; return x; }
  int g(void) { x = x - 1; return x; }
  
  The expression $f() + g()$ can evaluate either:
  - to 1 if $f()$ is evaluated first (returning 1), then $g()$ (returning 0);
  - to -1 if $g()$ is evaluated first (returning -1), then $f()$ (returning 0).
  
  Every C compiler will choose one evaluation order at compile-time. The compiled code therefore will have strictly fewer behaviors than the source program.
Observational equivalence is too strong

- If the source language is deterministic, then a compiler $C$ is correct if, and only if, $B(C(P)) = B(P)$.
- If the source language is nondeterministic, then this is too strong. Consider for instance the C language:

```c
int x = 0;
int f(void) { x = x + 1; return x; }
int g(void) { x = x - 1; return x; }
```

The expression $f() + g()$ can evaluate either:
- to 1 if $f()$ is evaluated first (returning 1), then $g()$ (returning 0);
- to -1 if $g()$ is evaluated first (returning -1), then $f()$ (returning 0).

Every C compiler will choose one evaluation order at compile-time. The compiled code therefore will have strictly fewer behaviors than the source program.
Should “going wrong” behaviors be preserved?

Compilers routinely “optimize away” going-wrong behaviors. For instance:

\[ x = 1/y; \ x = 42; \ \text{optimized to} \ \ x = 42; \]
(goes wrong if \( y == 0 \)) (always terminates normally)

- We know that the program being compiled does not go wrong
  - because it was type-checked with a sound type system
  - or because it was formally verified
- Or just, “garbage in, garbage out”.
Safe program refinement

- We restrict ourselves to those specifications Spec that are predicates over program behaviors that are not of the form $\text{GoesWrong}(\sigma)$. We write $\text{Spec} \in \text{SAFE}$.
- If $P \models \text{Spec}$ for some $\text{Spec} \in \text{SAFE}$, then $P$ does not go wrong.
Safe program refinement

- We restrict ourselves to those specifications Spec that are predicates over program behaviors that are not of the form \( \text{GoesWrong}(\sigma) \). We write \( \text{Spec} \in \text{SAFE} \).
- If \( P \models \text{Spec} \) for some \( \text{Spec} \in \text{SAFE} \), then \( P \) does not go wrong.

**Definition (Semantics preservation by safe program refinement)**

A compiler \( C \) is correct if, and only if:

\[
\forall \text{Spec} \in \text{SAFE}, \ P \models \text{Spec} \Rightarrow C(P) \models \text{Spec}
\]

which is equivalent to (safe program refinement):

\[
(\forall \sigma, \text{GoesWrong}(\sigma) \notin B(P)) \Rightarrow B(C(P)) \subseteq B(P)
\]

In particular, a correct compiler preserves safety.
Outline

1. General overview: compiler correctness
   - Motivation: software verification
   - Can you trust your compiler?
   - Compiler correctness
   - The architecture of CompCert
   - Live Demo!

2. The CompCert compilation passes in detail

3. Proof techniques (NOT part of the exam)

4. Conclusion
Xavier Leroy (1968 – )
Xavier Leroy (1968 – )

Most slides taken from his talks (LCTES 2008, POPL 2011, Oregon summer school 2012).
The CompCert experiment

(X. Leroy; S. Blazy, Z. Dargaye, J.B. Tristan; et al.)

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

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

Compiler written from scratch, along with its proof; not trying to prove an existing compiler.
CompCert resources

The subset of C supported

CompCert C supports all of ISO C 90 except:
- unstructured `switch` statements (such as Duff’s device)
- unprototyped (K&R-style) functions
- variadic functions
- `setjmp` and `longjmp`

In particular, CompCert C includes all of MISRA C 2004 (used in embedded software for vehicle manufacturers etc.) plus recursive functions, dynamic heap memory allocation, etc.
Performance of generated code on a PowerPC G5 processor

![Bar chart showing execution time for different benchmarks using gcc, CompCert, gcc -01, and gcc -03.]
Structure of the CompCert C compiler

C source
- external preprocessor
  - Preprocessed C
    - lexing and parsing
      - Parse tree
        - type-checking and elaboration
          - CompCert C AST

Pull side effects out of expressions
- Clight
  - type elimination; simplification of control

Stack allocation
- Cminor
  - instruction selection

Construction of a CFG
- CminorSel

Function inlining
- KTL
  - tail call optimization
  - constant propagation
  - common subexpression elimination
  - register allocation

Spilling, reboxing, calling conventions
- LTL

Linearization of the CFG
- LTLin

Layout of the stack frame
- Linear

Generation of Asm code
- Mach

Not verified yet
- Asm AST
  - printing
    - Asm text
      - external assembler
        - Object file
          - external linker
            - Executable

Formally verified
- validation by the stdc++/link tool
The whole CompCert compiler

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

Type reconstruction

Graph coloring

Code linearization heuristics

Executable → assembling → Assembly → printing of asm syntax
  linking

Verified compiler

AST C

Part of the TCB
Not part of the TCB

Not proved (hand-written in Caml)

Proved in Coq (extracted to Caml)
The formally verified part of the compiler

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

Optimizations: constant prop., CSE, tail calls, (LCM)

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

register allocation (Iterated Register Coalescing)

LTL → linearization of the CFG → LTLin → spilling, reloading calling conventions

Linear → layout of stack frames

Asm → asm code generation → Mach
Formally verified in Coq

After 50 000 lines of Coq and 4 person-years of effort:

Theorem transf_c_program_is_refinement:
  forall p tp,
  transf_c_program p = OK tp ->
  (forall beh, exec_C_program p beh -> not_wrong beh) ->
  (forall beh, exec_asm_program tp beh -> exec_C_program p beh).
What is a proof assistant

- An implementation of a well-defined mathematical logic.
  - (for Coq: the Calculus of Inductive Constructions with predicative Set, derived from intuitionistic logic)
- Provides a specification language to write definitions and state theorems.
  - (for Coq: Gallina)
- Provides a way to build a proof in interaction with the user (e.g. using tactics)
  - (Not fully automated proving.)
  - (for Coq: Ltac)
- Checks the proofs for soundness and completeness.
Programmed in Coq

Compilation algorithms written in Coq’s specification language, in pure functional style.

```coq
Fixpoint transl_expr (map: mapping) (a: expr) (rd: reg) (nd: node) {struct a}: mon node :=
  match a with
  | Evar v =>
    do r <- find_var map v; add_move r rd nd
  | Eop op al =>
    do rl <- alloc_regs map al;
    do no <- add_instr (Iop op rl rd nd);
    transl_exprlist map al rl no
  | Eload chunk addr al =>
    do rl <- alloc_regs map al;
    do no <- add_instr (Iload chunk addr rl rd nd);
    transl_exprlist map al rl no
  |...
```

Executable via automatic extraction to OCaml.
Outline

1 General overview: compiler correctness
   - Motivation: software verification
   - Can you trust your compiler?
   - Compiler correctness
   - The architecture of CompCert
   - Live Demo!

2 The CompCert compilation passes in detail

3 Proof techniques (NOT part of the exam)

4 Conclusion
Outline

1. General overview: compiler correctness
2. The CompCert compilation passes in detail
3. Proof techniques (NOT part of the exam)
4. Conclusion
Reminder: CompCert verified compilation passes

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

Optimizations: constant prop., CSE, tail calls, (LCM)

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

register allocation

(Iterated Register Coalescing)

**LTL** → linearization of the CFG → **LTLin** → spilling, reloading calling conventions → **Linear**

spilling, reloading calling conventions

**Asm** → asm code generation → **Mach**

stack allocation of variables

(Instruction scheduling)

layout of stack frames
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail
   - The front-end: C and C-like intermediate languages
   - RTL
   - Register allocation
   - Subsequent passes down to assembly code
   - Verified translation validation: the example of lazy code motion

3. Proof techniques (NOT part of the exam)

4. Conclusion
A mechanized semantics for C

- Official C semantics is defined by textual standard (> 500 pages)
  ISO/IEC 9899:2011
- Prone to errors and misinterpretations
- Nondeterministic semantics of expressions with side effects
- Undefined vs. unspecified vs. implementation-specific behaviors
The memory model

- Common to all languages
- Correctly reflects undefined behaviors for pointer arithmetics:

```c
void f() {
    int i[2], j;
    i[2] = 18;       // This is NOT j
}
```
Example C program

Compute the mean of an array of (double-precision) floating-point numbers.

```c
int mean (double* array, int size) {
    double* arrayEnd = &array[size];
    double sum = 0;
    while (array != arrayEnd) {
        sum += *(array++);
    }
    return sum/((double)size);
}

double test () {
    double foo[3] = {18, 1729, 42};
    return mean (foo, 3);
}
```
Clight

A deterministic subset of C with no side effects in expressions.

double mean (double* array, int size) {
    double* arrayEnd = &array[size];
    double sum = 0;
    while (array != arrayEnd) {
        sum = sum + *array;
        array = array + 1;
    }
    return sum/(float)size;
}

double test () {
    double foo[3];
    return mean (foo, 3);
}
Local addressed variables are laid out into stack. Other ones are transformed into non-addressable pseudo-registers.

test = fun () : () -> float {
    stack 24;
    stack[0] = 18d; stack[8] = 1729d; stack[16] = 42d;
    return mean (&stack[0], 3) : int -> int -> float
}
Local addressed variables are laid out into stack. Other ones are transformed into non-addressable pseudo-registers.

```c
mean = fun (array, size) : (int, int) -> int {
    stack 0;
    vars arrayEnd : int, sum : float;
    size = opIntTimes(size, opInt_8());
    arrayEnd = opIntPlus(array, size);
    sum = opFloat_0();
    block {
        loop {
            if opIntEq(array, arrayEnd) {
                exit 1;
            } else {
                sum = opIntPlus(sum, load(Mint32, array));
                array = opIntPlus(array, opInt_8());
            }
        }
    }
    return opDiv(sum, size);
}
```
“Internal” Cminor

Replace “abstract” Cminor operators with platform-specific operators.

```c
mean = fun (array, size) : (int, int) -> int {
    stack 0;
    vars arrayEnd: int, sum: float;
    size = opIntAddImm_8(size);
    arrayEnd = opIntPlus(array, size);
    sum = opInt_0();
    block {
        loop {
            if opIntEq(array, arrayEnd) {
                exit 1;
            } else {
                sum = opIntPlus(sum, load(Mint32, Direct, array));
                array = opIntAddImm_8(array);
            }
        }
    }
    return opDiv(sum, size);
}
```
Instruction selection

\[
\begin{align*}
\text{add}(e, \text{intconst}(n)) & \rightarrow \text{addi}_{n}(e) \\
\text{add}(\text{addi}_{n}(e_1), \text{addi}_{m}(e_2)) & \rightarrow \text{addi}_{n+m}(\text{add}(e_1, e_2)) \\
\text{add}(e_1, \text{addi}_{n}(e_2)) & \rightarrow \text{addi}_{n}(\text{add}(e_1, e_2)) \\
\text{mul}_{m}(\text{addi}_{n}(e)) & \rightarrow \text{addi}_{m \times n}(\text{mul}_{m}(e)) \\
\text{shl}(e, \text{intconst}(n)) & \rightarrow \text{rol}_{m,-1}.n(e) \\
\text{shru}(e, \text{intconst}(n)) & \rightarrow \text{rol}_{32-n,-1}.n(e) \\
\text{and}(e, \text{intconst}(n)) & \rightarrow \text{rol}_{0,n}(e) \\
\text{rol}(e, m_1, m_2)(e) & \rightarrow \text{rol}_{n_1+n_2,m}(e) \\
\text{or}(\text{rol}(e, m_1), \text{rol}(e, m_2))(e) & \rightarrow \text{rol}(e, m_1 \vee m_2)(e)
\end{align*}
\]

(\text{rol}_{n,m} is a left rotation by \(n\) bits followed by a logical “and” with \(m\).)
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail
   - The front-end: C and C-like intermediate languages
   - RTL
   - Register allocation
   - Subsequent passes down to assembly code
   - Verified translation validation: the example of lazy code motion

3. Proof techniques (NOT part of the exam)

4. Conclusion
Register Transfer Language, a.k.a. 3-address code.

The code of a function is represented by a control-flow graph:

- Nodes = instructions corresponding roughly to that of the processor, operating over variables (temporaries).
- Edge from $I$ to $J$: $J$ is a successor of $I$ (i.e. $J$ can execute just after $I$)
\textbf{RTL}

\begin{align*}
  & s = 0.0 \\
  & i = 0 \\
  \text{if} \ (i \geq \text{size}) \\
  & d = \text{float}(\text{size}) \\
  & e = s / f d \\
  & \text{return}(e) \\
  & a = i \ll 2 \\
  & b = \text{load(tbl, a)} \\
  & c = \text{float}(b) \\
  & s = s + f c \\
  & i = i + 1
\end{align*}
Cminor-to-RTL proof

Forward simulation:

\[
\begin{align*}
  sp, L, a, E, M & \xrightarrow{I \land P} sp, n_s, R, M \\
  \downarrow & \\
  sp, L, v, E', M' & \xrightarrow{I \land Q} sp, n_d, R', M'
\end{align*}
\]

where `transl_expr map mut a r_d n_d s = OK(s', n_s)`

\(I\): the values of Cminor variables mapped by `map` with RTL pseudo-registers match

\(P\): variable `a` corresponds to register `r_d`

\(Q\): `R'(r_d) = v` and the values of pseudo-registers not involved in the expression are preserved
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail
   - The front-end: C and C-like intermediate languages
   - RTL
   - Register allocation
   - Subsequent passes down to assembly code
   - Verified translation validation: the example of lazy code motion

3. Proof techniques (NOT part of the exam)

4. Conclusion
Register allocation

- **Purpose:** refine the notion of variables used as arguments and results of RTL operations.
  - RTL (before register allocation):
    - an unbounded quantity of variables
  - LTL (after register allocation):
    - a fixed number of hardware registers;
    - an unbounded number of stack slots.

- Insertion of spilling and reloading code is performed by a later pass.
- **Objective:** maximize the use of registers.
- **Technique used:** coloring of an interference graph.
- Instructions are grouped into basic blocks
  - Each block is a flat sequence of instructions
  - One can jump only to the beginning of a basic block
- Pseudo-registers are replaced with *locations*:
  - either physical registers
  - or locations to be later spilled into stack
Register allocation

1. Liveness analysis
2. Construction the interference graph
3. Coloring of the interference graph
4. Rewriting the code
Set up dataflow equations:

\[
L_{\text{in}}(p) = \text{transf}(L_{\text{out}}(p), \text{instr-at}(p))\\
L_{\text{out}}(p) = \bigcup\{L_{\text{in}}(s) | s \text{ successor of } p\}
\]

where:

\[
\text{transf}(X, r := \text{op}(r_1, \ldots, r_n)) = (X\setminus\{r\}) \cup \{r_1, \ldots, r_n\}
\]
Solve the equations using fixpoint iteration (Kildall’s algorithm)

```plaintext
while st_wrk is not empty, do
    extract a node n from st_wrk
    compute out = transf n l_in[n]
    for each successor s of n:
        compute in = lub l_in[s] out
        if in <> l_in[s]:
            l_in[s] := in
            st_wrk := st_wrk union {s}
    end if
end for
end while
return st_in
```
1: Liveness analysis

Show, by induction on the number of iterations, that the mapping $L_{out}$ computed by Kildall’s algorithm satisfies the inequations:

$$L_{out}(p) \supseteq \text{transf}(L_{out}(s), \text{instr-at}(p)) \text{ if } s \text{ successor of } p$$
2: Construction of the interference graph

For each instruction \( p : r := \ldots \), add edges between \( r \) and \( L_{out}(p) \setminus \{ r \} \)

Show that the final graph \( G \) contains all expected edges:

\[
p : x := \ldots \land y \neq x \land y \in L_{out}(p) \Rightarrow (x, y) \in G
\]
3: Coloring the interference graph

Construct a function $\phi : \text{Variable} \rightarrow \text{Register} \cup \text{Stackslot}$ such that $\phi(x) \neq \phi(y)$ if $x$ and $y$ interfere.
3: Coloring the interference graph

- The coloring algorithm $G \mapsto \phi$ is NOT written and proved in Coq
- Written in OCaml
- The output is checked by a validator written and proved in Coq: $V(G, \phi) = \text{true}$ if $\phi$ is a valid coloring of $G$, i.e. if:
  $$(x, y) \in G \Rightarrow \phi(x) \neq \phi(y)$$
- Allows using different coloring heuristics without changing the proof
- Correctness proof done for each compilation
4: Rewriting the code

Replace all variables $x$ with their color $\phi(x)$.

\[
\begin{align*}
&f_1 = 0.0 \\
r_3 = 0 \\
&\text{if } (r_3 \geq r_2) \\
f_2 = \text{float}(r_2) \\
f_1 = f_1 / f_2 \\
&\text{return } (f_1) \\
r_4 = r_3 \ll 2 \\
r_4 = \text{load}(r_1, r_4) \\
f_2 = \text{float}(r_4) \\
f_1 = f_1 + f_2 \\
r_3 = r_3 + 1
\end{align*}
\]
Proving the pass

What does “\( x \) is dead at \( p \)” mean, semantically?
That the program behaves the same regardless of the value of \( x \) at point \( p \).
Proving the pass

What does “x is dead at p” mean, semantically?
That the program behaves the same regardless of the value of x at point p.

**Invariant**

Let $E : \text{Variable} \rightarrow \text{Value}$ be the values of variables at point p in the original program. Let $R : \text{Location} \rightarrow \text{Value}$ be the values of locations at point p in the transformed program.

$E$ and $R$ agree at p, written $p \vdash E \sim R$, iff:

$$E(x) = R(\phi(x)) \text{ for all } x \text{ live before point } p$$
Proving the pass

Show a forward simulation of the form:

\[ p, E, M \quad \Rightarrow \quad p \vdash E \approx R \quad \Rightarrow \quad p, R, M \]

\[ \downarrow t \quad \downarrow t \]

\[ p', E', M' \quad \Rightarrow \quad p' \vdash E' \approx R' \quad \Rightarrow \quad p', R', M' \]

Hypotheses: left, a transition in the original code; top, the invariant (register agreement) before the transition.
Conclusions: one transition (lock-step) in the transformed code; bottom, the invariant after the transition.
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail
   - The front-end: C and C-like intermediate languages
   - RTL
   - Register allocation
   - Subsequent passes down to assembly code
   - Verified translation validation: the example of lazy code motion

3. Proof techniques (NOT part of the exam)

4. Conclusion
Linearization

1. Tunneling: eliminate chains of gotos
2. Ordering reachable nodes
3. Listing those nodes in a sorted list wrt. this order (heuristic approach similar to trace picking), each followed with a goto
4. Eliminating every goto to an immediate successor in the list
The only pass changing memory layout, thus hard to prove (although usually deemed trivial)

One intermediate language syntax (Mach) with two different semantics
Stack frame layout, incl. register spilling

Figure 1. Overview of register allocation and introduction of activation records. For each intermediate language, the placement of function-local data is outlined, either in the memory-allocated activation record (top part) or in non memory-resident execution environments (bottom part).
Generating PPC assembly code

- Small-step PPC assembly semantics
- Macro-instructions expanded to PPC assembly instructions
- Correctness depends on handling floating-point numbers (IEEE-compliance?), axiomatized in the whole development
- Straightforward. Tedious but trivial.
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail
   - The front-end: C and C-like intermediate languages
   - RTL
   - Register allocation
   - Subsequent passes down to assembly code
   - Verified translation validation: the example of lazy code motion

3. Proof techniques (NOT part of the exam)

4. Conclusion
Lazy code motion (Knoop, Rüthing & Steffen, 1992) and its predecessor, partial redundancy elimination (Morel & Renvoise, 1979) perform:

- Elimination of common subexpressions, even across basic blocks
- Loop invariant code motion
- Factoring of partially redundant computations (i.e. computations that occur multiple times on some paths, but 0 or 1 times on others.)

This platform-independent optimization is performed at the RTL level.
An example of lazy code motion

c := a + b

d := a + b
e := a + b
An example of lazy code motion

c := t

t := a + b

d := t

e := t

t := a + b
Proving the correctness of lazy code motion?

A direct mechanized proof of lazy code motion appears very difficult:
- LCM exploits the results of no less than 4 dataflow analyses.
- LCM is a highly non-local transformation: instructions are moved across basic blocks and even across loops.
- The transformation generates fresh temporaries, which adds significant bureaucratic overhead to mechanized proofs.
An efficient proof: verified translation validation

Tristan & Leroy, PLDI 2009.
Unverified, untrusted implementation of the transformation (in OCaml):
- Can use bitvectors, imperative data structures, etc.
- Easy to experiment with variants.

A posteriori validation with a validator written and proved correct in Coq:
- Input: the code before and after LCM.
- Output: a boolean, true = “semantics is preserved”, false = “I don’t know”.

The validator

- Define a mapping from instructions of the original program to instructions of the transformed program. (This mapping can be proved by the untrusted transformation.)
- Check that this mapping embeds the original control-flow graph in the transformed control-flow graph.

![Diagram showing the mapping between the original and transformed control-flow graphs.](image-url)
Proving the validator

Check each matching pairs of instructions.

<table>
<thead>
<tr>
<th>Original instruction</th>
<th>Transformed instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>( t := \text{op}(y, z) )</td>
<td>Check that the computation ( \text{op}(y, z) ) is <strong>anticipable</strong> at this point in the original program (see later).</td>
</tr>
<tr>
<td>( x := \text{op}(y, z) )</td>
<td>( x := t )</td>
<td>Check that the equality ( t = \text{op}(y, z) ) holds at this point in the transformed program, based on the results of a standard <strong>reaching definition</strong> analysis.</td>
</tr>
<tr>
<td>Otherwise</td>
<td>Otherwise</td>
<td>Check that the two instructions are identical</td>
</tr>
</tbody>
</table>
The anticipability problem

Consider a computation that can go wrong at run-time, such as an integer division.

If we place a computation of $a/b$ at one of the $\times$ points, the transformed program can crash on a division by zero while the original program didn’t.

**Anticipability criterion:** a computation $a/b$ is *anticipable* at point $p$ if all execution paths starting at $p$ eventually compute $a/b$. 
Proving the correctness of LCM

Assuming the validator returns true, show the simulation diagram:

\[ p, E, M \xrightarrow{t} p', E', M' \xrightarrow{\text{Invariant}} p, E_1, M \]

\[ p', E', M' \xrightarrow{t} p', E'_1, M' \]

(doesn’t go wrong)

The invariant includes:

- Agreement on the values of non-temporary variables:

\[ \forall x \in \text{Dom}(E) : E_1(x) = E(x) \]

- The equations inferred by reaching definition analysis are satisfied.
The definition and correctness proof of the validator are not small (7000 lines of Coq). So, is the verified validator approach effective?

- Yes, because the proof remains conceptually simple. In particular, only 2 dataflow analyses are used (reaching definitions and anticipability), both of which have simple semantic characterizations.

- Yes, because the validator (possibly with extensions) can be reused for other optimizations.
Outline

1. General overview: compiler correctness
2. The CompCert compilation passes in detail
3. Proof techniques (NOT part of the exam)
4. Conclusion
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail

3. Proof techniques (NOT part of the exam)
   - Proof architecture
   - Formalizing the languages
   - Safe program refinement through simulation diagrams
   - Verified translation validation

4. Conclusion
Decomposition in multiple compiler passes

- If every compiler pass preserves semantics, then so does their composition!
- Each compiler pass can be proved independently of the others.
Proving the correctness of each pass

1. Formalize the source and the target language of the pass.
2. Prove the semantics preservation for the pass.

In fact, for each pass, a stronger result is proved under the form of a *simulation diagram*. Simulation diagrams for consecutive passes compose together.
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail

3. Proof techniques (NOT part of the exam)
   - Proof architecture
   - Formalizing the languages
   - Safe program refinement through simulation diagrams
   - Verified translation validation

4. Conclusion
Why indulge in formal semantics?

- An intellectually challenging issue.
- When English prose is not enough.
  (e.g. ISO C standardization document)
- A prerequisite to formal program verification
  (e.g. program proof, model checking, static analysis, etc.)
- A prerequisite to building reliable “meta-programs”
  (programs that operate over programs: compilers, code generators, program verifiers, type-checkers, etc.)
Formal syntax and semantics

A language is formally defined as:

- a *syntax*, which defines the set of all valid programs in this language.
  In CompCert, a program is an abstract syntax tree. The syntax is defined by inductive types.

- a *semantics*, which defines for each valid program $P$ its set of behaviors $B(P)$. 
Big-step semantics

A predicate $c/s \Rightarrow s'$ meaning “started in state $s$, command $c$ terminates and the final state is $s'$.

\[
\begin{align*}
\text{SKIP}/s \Rightarrow s & \quad x := a/s \Rightarrow s[x \leftarrow \text{aeval } s \ a] \\
c_1/s \Rightarrow s_1 & \quad c_2/s_1 \Rightarrow s_2 & \quad c_1/s \Rightarrow s' \text{ if beval } s \ b = \text{true} \\
& \quad c_2/s \Rightarrow s' \text{ if beval } s \ b = \text{false} & \quad \text{IFB } b \ \text{THEN } c_1 \ \text{ELSE } c_2 \ \text{FI}/s \Rightarrow s' \\
\hline
\text{beval } s \ b = \text{false} & \quad \text{WHILE } b \ \text{DO } c \ \text{END}/s \Rightarrow s \\
\hline
\text{beval } s \ b = \text{true} & \quad c/s \Rightarrow s_1 & \quad \text{WHILE } b \ \text{DO } c \ \text{END}/s_1 \Rightarrow s_2 \\
& \quad \text{WHILE } b \ \text{DO } c \ \text{END}/s \Rightarrow s_2
\end{align*}
\]
Pros and cons of big-step semantics

Pros:
- Follows naturally the structure of programs. (cf. Gilles Kahn: “natural semantics”)
- Close connection with language interpreters.
- Powerful induction principle (on the structure of derivations)
- Easy to extend with various structured constructs (functions and procedures, other forms of loops)

Cons:
- Fails to characterize diverging executions. (More precisely: no distinction between divergence and going wrong.)
- Unstructured control (goto) nearly impossible to handle.
Big-step semantics and divergence

Big-step semantics fails to distinguish between divergence and going wrong:

\[ c/s \text{ diverges} \lor c/s \text{ goes wrong} \iff \neg(\exists s', c/s \Rightarrow s') \]

Highly desirable: a \textit{positive} characterization of divergence, distinguishing it from “going wrong”.
Small-step semantics

Also called “structural operational semantics”
Like \( \beta \)-reduction in the \( \lambda \)-calculus: view computations as sequences of reductions:

\[
M \rightarrow M_1 \rightarrow M_2 \rightarrow \ldots
\]

- Each reduction \( M \rightarrow M' \) represents an elementary computation.
- \( M' \) also contains a representation of the residual computations that remain to be done later. (= continuations).
- In CompCert, a transition also produces a finite list of events \( \sigma \): 
  \[
  M \xrightarrow{\sigma} M'.
  \]
Program behaviors through sequences of reductions

The behaviors of a program $P$ are obtained by computing an *initial* state $s_0$ from $P$ then starting a sequence of reductions from $s_0$:

- **Termination:** finite sequence of reductions to a final state $s'$
  \[
  s_0 \rightarrow \cdots \rightarrow s'
  \]

- **Divergence:** infinite sequence of reductions.
  \[
  s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \rightarrow \ldots
  \]

- **Going wrong:** finite sequence of reductions to an irreducible, non-final state $s'$.
Program behaviors through sequences of reductions

The behaviors of a program $P$ are obtained by computing an initial state $s_0$ from $P$ then starting a sequence of reductions from $s_0$:

- **Termination**: finite sequence of reductions to a final state $s'$
  
  $$s_0 \rightarrow \cdots \rightarrow s'$$

- **Divergence**: infinite sequence of reductions.
  
  $$s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n \rightarrow \cdots$$

- **Going wrong**: finite sequence of reductions to an irreducible, non-final state $s'$.

With events:

- record the concatenation of event traces for terminating or wrong
- for diverging behaviors, distinguish whether finitely or infinitely many events are produced.
Outline

1. General overview: compiler correctness

2. The CompCert compilation passes in detail

3. Proof techniques (NOT part of the exam)
   - Proof architecture
   - Formalizing the languages
   - Safe program refinement through simulation diagrams
   - Verified translation validation

4. Conclusion
How to prove safe program refinement?

\[(\forall \sigma, \text{GoesWrong}(\sigma) \notin \mathcal{B}(P)) \Rightarrow \mathcal{B}(C(P)) \subseteq \mathcal{B}(P)\]

Safe program refinement is rather difficult to prove:

- We need to consider all steps that the compiled code can take, and trace them back to steps that the source program can take. This is called \textit{backward simulation}.

- This is problematic if one source-level step is broken into several target-level steps.
General shape of a backward simulation

Example: compilation of Java to JVM

Source code: \[1 + 2\]  →  3

| VM code:   | Iconst(1) → Iconst(2) → Iadd |
| VM stack:  | nil → 1 :: nil → 2 :: 1 :: nil → 3 :: nil |

Intermediate JVM code sequences like \texttt{Iconst(2); Iadd} or just \texttt{Iadd} do not correspond to the compilation of any Java source expression. One solution: invent a \textit{decompilation} function that is left-inverse of compilation. Very hard in general!
Forward simulation

Intuitive simulation diagram:

Source code: 1 + 2  \rightarrow  3

compilation

VM code: \texttt{i}const(1) \rightarrow \texttt{i}const(2) \rightarrow \texttt{i}add

VM stack: nil \rightarrow 1 :: nil \rightarrow 2 :: 1 :: nil \rightarrow 3 :: nil
Forward simulation

Intuitive simulation diagram:

Source code: 1+2 \rightarrow 3

Compilation

VM code: Iconst(1) \rightarrow Iconst(2) \rightarrow Iadd

VM stack: nil \rightarrow 1 :: nil \rightarrow 2 :: 1 :: nil \rightarrow 3 :: nil

Helps showing forward program refinement:

\[ \mathcal{B}(P) \subseteq \mathcal{B}(C(P)) \]

and safe forward program refinement:

\[ (\forall \sigma, \text{GoesWrong}(\sigma) \notin \mathcal{B}(P)) \Rightarrow \mathcal{B}(P) \subseteq \mathcal{B}(C(P)) \]
Forward simulation

Intuitive simulation diagram:

Source code: \(1+2\) \[\rightarrow\] \(3\)

compilation

VM code: \(\text{lconst}(1)\) \[\rightarrow\] \(\text{lconst}(2)\) \[\rightarrow\] \(\text{ladd}\)

VM stack: \(\text{nil}\) \[\rightarrow\] \(1::\text{nil}\) \[\rightarrow\] \(2::1::\text{nil}\) \[\rightarrow\] \(3::\text{nil}\)

Helps showing forward program refinement:

\[\mathcal{B}(P) \subseteq \mathcal{B}(C(P))\]

and safe forward program refinement:

\[(\forall \sigma, \text{GoesWrong}(\sigma) \notin \mathcal{B}(P)) \Rightarrow \mathcal{B}(P) \subseteq \mathcal{B}(C(P))\]

What if the compiled program has more behaviors than the source?
Determinism to the rescue!

Lemma

If the target language is deterministic, then:
- Forward program refinement implies program refinement
- Safe forward program refinement implies safe program refinement

Easy result: follows from ∅ ⊊ B ⊆ {b} ⇒ B = {b}

Generalizes to forward/backward simulation diagrams (a trickier result).
Relating preservation properties

- Bisimulation
  - Forward simulation
    - if $P_2$ deterministic
  - Backward simulation
    - if $P_1$ deterministic
  - Safe forward simulation
    - if $P_2$ deterministic
  - Safe backward simulation
    - if $P_1$ deterministic
  - Preservation of specifications
Reminder: CompCert verified compilation passes

CompCert C \(\rightarrow\) Clight \(\rightarrow\) C#minor

Optimizations: constant prop., CSE, tail calls, (LCM)

RTL \(\rightarrow\) CminorSel \(\rightarrow\) Cminor

CFG construction expr. decomp.

register allocation

(Iterated Register Coalescing)

LTL \(\rightarrow\) LTLin \(\rightarrow\) Linear

linearization of the CFG

spilling, reloading calling conventions

stack allocation of variables

Instruction scheduling

LTLin \(\rightarrow\) Linear

spilling, reloading calling conventions

layout of stack frames

asm code generation

Asm \(\rightarrow\) Mach

type elimination loop simplifications
Plan for verifying CompCert passes

1. Prove “safe forward simulation” between source and compiled codes.
2. Prove that the target language is deterministic.
3. Conclude that all functional specifications are preserved by compilation.

Note: (1) + (2) imply that the source language has deterministic semantics (to be able to compose forward simulations). This is the case except for the first pass from CompCert C, so start by determinizing C’s semantics (here, by fixing evaluation order of arguments a priori).
Handling multiple compilation passes

Source (non-det)

Source (determinized)

Intermediate language 1

Intermediate language 2

Machine code
Handling multiple compilation passes

Source (non-det) → (same code) → Source (determinized)

- pass 1 → Intermediate language 1
- pass 2 → Intermediate language 2
- pass 3 → Machine code

- blue: forward simulation proof
- red: backward simulation proof
Handling multiple compilation passes

Source (non-det) → (same code) → Source (determinized)

pass 1 → Intermediate language 1

pass 2 → Intermediate language 2

pass 3 → Machine code

- forward simulation proof
- backward simulation proof
Handling multiple compilation passes

Source (non-det) → (same code) → Source (determinized) → pass 1 → Intermediate language 1 → pass 2 → Intermediate language 2 → pass 3 → Machine code

- : forward simulation proof
- : backward simulation proof
Handling multiple compilation passes

Source (non-det) → (same code) → Source (determinized) → pass 1 → Intermediate language 1 → pass 2 → Intermediate language 2 → pass 3 → Machine code

- Blue arrows: forward simulation proof
- Red arrows: backward simulation proof
Outline

1. General overview: compiler correctness
2. The CompCert compilation passes in detail
3. Proof techniques (NOT part of the exam)
   - Proof architecture
   - Formalizing the languages
   - Safe program refinement through simulation diagrams
   - Verified translation validation
4. Conclusion
Reminder: The whole CompCert compiler

The verified passes of CompCert can use unverified code, as long as the output of such code is checked through a verified validator.
Validation to the rescue

Verified transformation

Transformation

Verified translation validation

Transformation

External solver with verified validation

Transformation

Untrusted solver

Checker

Validator

= formally verified

= not verified

Ramananandro (Yale)
Verified translation validation

- Input: a source $P$ and a target program $P'$
- Output: a boolean $V(P, P')$
- Formal proof that, if $V(P, P') = \text{true}$, then $P'$ refines $P$. Thus, the validator is written in Coq within CompCert.
Verified translation validation

- Input: a source $P$ and a target program $P'$
- Output: a boolean $V(P, P')$
- Formal proof that, if $V(P, P') = \text{true}$, then $P'$ refines $P$. Thus, the validator is written in Coq within CompCert.

Allows trusting the transformation, while the validator does not need to know how the target program was computed:

- The transformation can be written in an unverified language.
- It can be optimized without having to modify the validator or its proof.
- Validation occurs at each compilation. If $V(P, P') = \text{false}$, then compilation crashes, or the optimization pass (if single-language) is ignored.
- Example: lazy code motion (including loop unwinding).

Generalizes to parts of compilation passes (e.g. interference graph coloring).
Another technique: certifying compilation (Necula et al.)

Proof-carrying code instantiated to compilers. For a program \( P \), \( C(P) = (P', \pi) \) where \( \pi \) is a proof certificate that \( P' \) refines \( P \).

CompCert is not focused on this technique (which includes some efficiency challenges about the computation of the proof certificate). However, the two techniques are theoretically equivalent:

- a certifying compiler can be easily turned into a certified compiler by dropping the proof certificate
- a certified compiler can be easily turned into a certifying compiler by specializing the compiler correctness proof to the source program
Outline

1. General overview: compiler correctness
2. The CompCert compilation passes in detail
3. Proof techniques (NOT part of the exam)
4. Conclusion
Conclusion

The formal verification of realistic compilers is feasible. Much work remains:

- Shrinking the trusted computing base: what about parsing, assembling, linking?
- More optimizations
- Front-ends for other languages (C++?)
- Concurrency! (Shao et al., Appel et al.)
- Connections with source-level verification
Thank you for your attention

- tahina.ramananandro@yale.edu
- http://cs.yale.edu/~tahina