DOI:10.1145/1538788.1538814

Formal Verification of a Realistic Compiler

By Xavier Leroy

Abstract

This paper reports on the development and formal verification (proof of semantic preservation) of CompCert, a compiler from Clight (a large subset of the C programming language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. Such a verified compiler is useful in the context of critical software and its formal verification: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

1. INTRODUCTION

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compilers to crash at compile-time or—much worse—to silently generate an incorrect executable for a correct source program.

For low-assurance software, validated only by testing, the impact of compiler bugs is low: what is tested is the executable code produced by the compiler; rigorous testing should expose compiler-introduced errors along with errors already present in the source program. Note, however, that compiler-introduced bugs are notoriously difficult to expose and track down. The picture changes dramatically for safety-critical, high-assurance software. Here, validation by testing reaches its limits and needs to be complemented or even replaced by the use of formal methods such as model checking, static analysis, and program proof. Almost universally, these formal verification tools are applied to the source code of a program. Bugs in the compiler used to turn this formally verified source code into an executable can potentially invalidate all the guarantees so painfully obtained by the use of formal methods. In future, where formal methods are routinely applied to source programs, the compiler could appear as a weak link in the chain that goes from specifications to executables. The safety-critical software industry is aware of these issues and uses a variety of techniques to alleviate them, such as conducting manual code reviews of the generated assembly code after having turned all compiler optimizations off. These techniques do not fully address the issues, and are costly in terms of development time and program performance.

An obviously better approach is to apply formal methods to the compiler itself in order to gain assurance that it preserves the semantics of the source programs. For the last 5 years, we have been working on the development of a *realistic*, *verified* compiler called CompCert. By *verified*, we mean a compiler that is accompanied by a machine-checked proof of a semantic preservation property: the generated machine code behaves as prescribed by the semantics of the source program. By *realistic*, we mean a compiler that could realistically be used in the context of production of critical software. Namely, it compiles a language commonly used for critical embedded software: neither Java nor ML nor assembly code, but a large subset of the C language. It produces code for a processor commonly used in embedded systems: we chose the PowerPC because it is popular in avionics. Finally, the compiler must generate code that is efficient enough and compact enough to fit the requirements of critical embedded systems. This implies a multipass compiler that features good register allocation and some basic optimizations.

Proving the correctness of a compiler is by no ways a new idea: the first such proof was published in 1967[16] (for the compilation of arithmetic expressions down to stack machine code) and mechanically verified in 1972.[17] Since then, many other proofs have been conducted, ranging from single-pass compilers for toy languages to sophisticated code optimizations.[8] In the CompCert experiment, we carry this line of work all the way to end-to-end verification of a complete compilation chain from a structured imperative language down to assembly code through eight intermediate languages. While conducting the verification of CompCert, we found that many of the nonoptimizing translations performed, while often considered obvious in the compiler literature, are surprisingly tricky to formally prove correct.

This paper gives a high-level overview of the CompCert compiler and its mechanized verification, which uses the Coq proof assistant.[3, 7] This compiler, classically, consists of two parts: a front-end translating the Clight subset of C to a low-level, structured intermediate language called Cminor, and a lightly optimizing back-end generating PowerPC assembly code from Cminor. A detailed description of Clight can be found in Blazy and Leroy[5]; of the compiler front-end in Blazy et al.[4]; and of the compiler back-end in Leroy.[11, 13] The complete source code of the Coq development, extensively commented, is available on the Web.[12]

The remainder of this paper is organized as follows. Section 2 compares and formalizes several approaches to establishing trust in the results of compilation. Section 3

A previous version of this paper was published in *Proceedings of the 33rd Symposium on the Principles of Programming Languages*. ACM, NY, 2006.

describes the structure of the CompCert compiler, its performance, and how the Coq proof assistant was used not only to prove its correctness but also to program most of it. By lack of space, we will not detail the formal verification of every compilation pass. However, Section 4 provides a technical overview of such a verification for one crucial pass of the compiler: register allocation. Finally, Section 5 presents preliminary conclusions and directions for future work.

2. APPROACHES TO TRUSTED COMPILATION

2.1. Notions of semantic preservation

Consider a source program S and a compiled program C produced by a compiler. Our aim is to prove that the semantics of S was preserved during compilation. To make this notion of semantic preservation precise, we assume given semantics for the source and target languages that associate observable behaviors B to S and C. We write $S \Downarrow B$ to mean that program S executes with observable behavior B. The behaviors we observe in CompCert include termination, divergence, and "going wrong" (invoking an undefined operation that could crash, such as accessing an array out of bounds). In all cases, behaviors also include a trace of the input–output operations (system calls) performed during the execution of the program. Behaviors therefore reflect accurately what the user of the program, or more generally the outside world the program interacts with, can observe.

The strongest notion of semantic preservation during compilation is that the source program S and the compiled code C have exactly the same observable behaviors:

$$\forall B,\ S \Downarrow B \Leftrightarrow C \Downarrow B \tag{1}$$

Notion (1) is too strong to be usable. If the source language is not deterministic, compilers are allowed to select one of the possible behaviors of the source program. (For instance, C compilers choose one particular evaluation order for expressions among the several orders allowed by the C specifications.) In this case, C will have fewer behaviors than S. Additionally, compiler optimizations can optimize away "going wrong" behaviors. For example, if S can go wrong on an integer division by zero but the compiler eliminated this computation because its result is unused, C will not go wrong. To account for these degrees of freedom in the compiler, we relax definition (1) as follows:

$$S\ \texttt{safe} \Rightarrow (\forall B,\ C \Downarrow B \Rightarrow S \Downarrow B) \tag{2}$$

(Here, S `safe` means that none of the possible behaviors of S is a "going wrong" behavior.) In other words, if S does not go wrong, then neither does C; moreover, all observable behaviors of C are acceptable behaviors of S.

In the CompCert experiment and the remainder of this paper, we focus on source and target languages that are deterministic (programs change their behaviors only in response to different inputs but not because of internal choices) and on execution environments that are deterministic as well (the inputs given to the programs are uniquely determined by their previous outputs). Under these conditions, there exists exactly one behavior B such that $S \Downarrow B$, and similarly for C. In this case, it is easy to prove that property (2) is equivalent to

$$\forall B \notin \texttt{Wrong},\ S \Downarrow B \Rightarrow C \Downarrow B \tag{3}$$

(Here, `Wrong` is the set of "going wrong" behaviors.) Property (3) is generally much easier to prove than property (2), since the proof can proceed by induction on the execution of S. This is the approach that we take in this work.

From a formal methods perspective, what we are really interested in is whether the compiled code satisfies the functional specifications of the application. Assume that these specifications are given as a predicate $Spec(B)$ of the observable behavior. We say that C satisfies the specifications, and write $C \models Spec$, if C cannot go wrong (C `safe`) and all behaviors of B satisfy $Spec$ ($\forall B, C \Downarrow B \Rightarrow Spec(B)$). The expected correctness property of the compiler is that it preserves the fact that the source code S satisfies the specification, a fact that has been established separately by formal verification of S:

$$S \models Spec \Rightarrow C \models Spec \tag{4}$$

It is easy to show that property (2) implies property (4) for all specifications $Spec$. Therefore, establishing property (2) once and for all spares us from establishing property (4) for every specification of interest.

A special case of property (4), of considerable historical importance, is the preservation of type and memory safety, which we can summarize as "if S does not go wrong, neither does C":

$$S\ \texttt{safe} \Rightarrow C\ \texttt{safe} \tag{5}$$

Combined with a separate check that S is well-typed in a sound type system, property (5) implies that C executes without memory violations. Type-preserving compilation[18] obtains this guarantee by different means: under the assumption that S is well typed, C is proved to be well typed in a sound type system, ensuring that it cannot go wrong. Having proved properties (2) or (3) provides the same guarantee without having to equip the target and intermediate languages with sound type systems and to prove type preservation for the compiler.

2.2. Verified, validated, certifying compilers

We now discuss several approaches to establishing that a compiler preserves semantics of the compiled programs, in the sense of Section 2.1. In the following, we write $S \approx C$, where S is a source program and C is compiled code, to denote one of the semantic preservation properties (1) to (5) of Section 2.1.

Verified Compilers. We model the compiler as a total function $Comp$ from source programs to either compiled code (written $Comp(S) = \texttt{OK}(C)$) or a compile-time error (written $Comp(S) = \texttt{Error}$). Compile-time errors correspond to cases where the compiler is unable to produce code, for instance if the source program is incorrect (syntax error, type error,

etc.), but also if it exceeds the capacities of the compiler. A compiler *Comp* is said to be verified if it is accompanied with a formal proof of the following property:

$$\forall S, C,\ \mathit{Comp}(S) = \mathtt{OK}(C) \Rightarrow S \approx C \qquad (6)$$

In other words, a verified compiler either reports an error or produces code that satisfies the desired correctness property. Notice that a compiler that always fails ($\mathit{Comp}(S) = \mathtt{Error}$ for all S) is indeed verified, although useless. Whether the compiler succeeds to compile the source programs of interest is not a correctness issue, but a quality of implementation issue, which is addressed by nonformal methods such as testing. The important feature, from a formal verification standpoint, is that the compiler never silently produces incorrect code.

Verifying a compiler in the sense of definition (6) amounts to applying program proof technology to the compiler sources, using one of the properties defined in Section 2.1 as the high-level specification of the compiler.

Translation Validation with Verified Validators. In the translation validation approach[20, 22] the compiler does not need to be verified. Instead, the compiler is complemented by *a validator*: a boolean-valued function $\mathit{Validate}(S, C)$ that verifies the property $S \approx C$ a posteriori. If $\mathit{Comp}(S) = \mathtt{OK}(C)$ and $\mathit{Validate}(S, C) = \mathtt{true}$, the compiled code C is deemed trustworthy. Validation can be performed in several ways, ranging from symbolic interpretation and static analysis of S and C to the generation of verification conditions followed by model checking or automatic theorem proving. The property $S \approx C$ being undecidable in general, validators are necessarily incomplete and should reply `false` if they cannot establish $S \approx C$.

Translation validation generates additional confidence in the correctness of the compiled code, but by itself does not provide formal guarantees as strong as those provided by a verified compiler: the validator could itself be incorrect. To rule out this possibility, we say that a validator *Validate* is verified if it is accompanied with a formal proof of the following property:

$$\forall S, C,\ \mathit{Validate}(S, C) = \mathtt{true} \Rightarrow S \approx C \qquad (7)$$

The combination of a verified validator *Validate* with an unverified compiler *Comp* does provide formal guarantees as strong as those provided by a verified compiler. Indeed, consider the following function:

$$\begin{array}{l}\mathit{Comp'}(S) = \\ \quad \mathtt{match}\ \mathit{Comp}(S)\ \mathtt{with} \\ \quad \mid \mathtt{Error} \rightarrow \mathtt{Error} \\ \quad \mid \mathtt{OK}(C) \rightarrow \mathtt{if}\ \mathit{Validate}(S, C)\ \mathtt{then}\ \mathtt{OK}(C)\ \mathtt{else}\ \mathtt{Error}\end{array}$$

This function is a verified compiler in the sense of definition (6). Verification of a translation validator is therefore an attractive alternative to the verification of a compiler, provided the validator is smaller and simpler than the compiler.

Proof-Carrying Code and Certifying Compilers. The proof-carrying code (PCC) approach[1, 19] does not attempt to establish semantic preservation between a source program and some compiled code. Instead, PCC focuses on the generation of independently checkable evidence that the compiled code C satisfies a behavioral specification *Spec* such as type and memory safety. PCC makes use of a *certifying compiler*, which is a function *CComp* that either fails or returns both a compiled code C and a proof π of the property $C \models \mathit{Spec}$. The proof π, also called a *certificate*, can be checked independently by the code user; there is no need to trust the code producer, nor to formally verify the compiler itself. The only part of the infrastructure that needs to be trusted is the client-side checker: the program that checks whether π entails the property $C \models \mathit{Spec}$.

As in the case of translation validation, it suffices to formally verify the client-side checker to obtain guarantees as strong as those obtained from compiler verification of property (4). Symmetrically, a certifying compiler can be constructed, at least theoretically, from a verified compiler, provided that the verification was conducted in a logic that follows the "propositions as types, proofs as programs" paradigm. The construction is detailed in Leroy.[11, section 2]

2.3. Composition of compilation passes

Compilers are naturally decomposed into several passes that communicate through intermediate languages. It is fortunate that verified compilers can also be decomposed in this manner. Consider two verified compilers Comp_1 and Comp_2 from languages L_1 to L_2 and L_2 to L_3, respectively. Assume that the semantic preservation property $\approx$ is transitive. (This is true for properties (1) to (5) of Section 2.1.) Consider the error-propagating composition of Comp_1 and Comp_2:

$$\begin{array}{l}\mathit{Comp}(S) = \mathtt{match}\ \mathit{Comp}_1(S)\ \mathtt{with} \\ \qquad \mid \mathtt{Error} \rightarrow \mathtt{Error} \\ \qquad \mid \mathtt{OK}(I) \ \rightarrow \mathit{Comp}_2(I)\end{array}$$

It is trivial to show that this function is a verified compiler from L_1 to L_3.

2.4. Summary

The conclusions of this discussion are simple and define the methodology we have followed to verify the CompCert compiler back-end. First, provided the target language of the compiler has deterministic semantics, an appropriate specification for the correctness proof of the compiler is the combination of definitions (3) and (6):

$$\forall S, C, B \not\in \mathtt{Wrong},\quad \mathit{Comp}(S) = \mathtt{OK}(C) \wedge S \Downarrow B \Rightarrow C \Downarrow B$$

Second, a verified compiler can be structured as a composition of compilation passes, following common practice. However, all intermediate languages must be given appropriate formal semantics.

Finally, for each pass, we have a choice between proving the code that implements this pass or performing the transformation via untrusted code, then verifying its results using a verified validator. The latter approach can reduce the amount of code that needs to be verified.

3. OVERVIEW OF THE COMPCERT COMPILER

3.1. The source language

The source language of the CompCert compiler, called Clight,[5] is a large subset of the C programming language, comparable to the subsets commonly recommended for writing critical embedded software. It supports almost all C data types, including pointers, arrays, `struct` and `union` types; all structured control (`if/then`, loops, `break`, `continue`, Java-style `switch`); and the full power of functions, including recursive functions and function pointers. The main omissions are extended-precision arithmetic (`long long` and `long double`); the `goto` statement; nonstructured forms of `switch` such as Duff's device; passing `struct` and `union` parameters and results by value; and functions with variable numbers of arguments. Other features of C are missing from Clight but are supported through code expansion (de-sugaring) during parsing: side effects within expressions (Clight expressions are side-effect free) and block-scoped variables (Clight has only global and function-local variables).

The semantics of Clight is formally defined in big-step operational style. The semantics is deterministic and makes precise a number of behaviors left unspecified or undefined in the ISO C standard, such as the sizes of data types, the results of signed arithmetic operations in case of overflow, and the evaluation order. Other undefined C behaviors are consistently turned into "going wrong" behaviors, such as dereferencing the null pointer or accessing arrays out of bounds. Memory is modeled as a collection of disjoint blocks, each block being accessed through byte offsets; pointer values are pairs of a block identifier and a byte offset. This way, pointer arithmetic is modeled accurately, even in the presence of casts between incompatible pointer types.

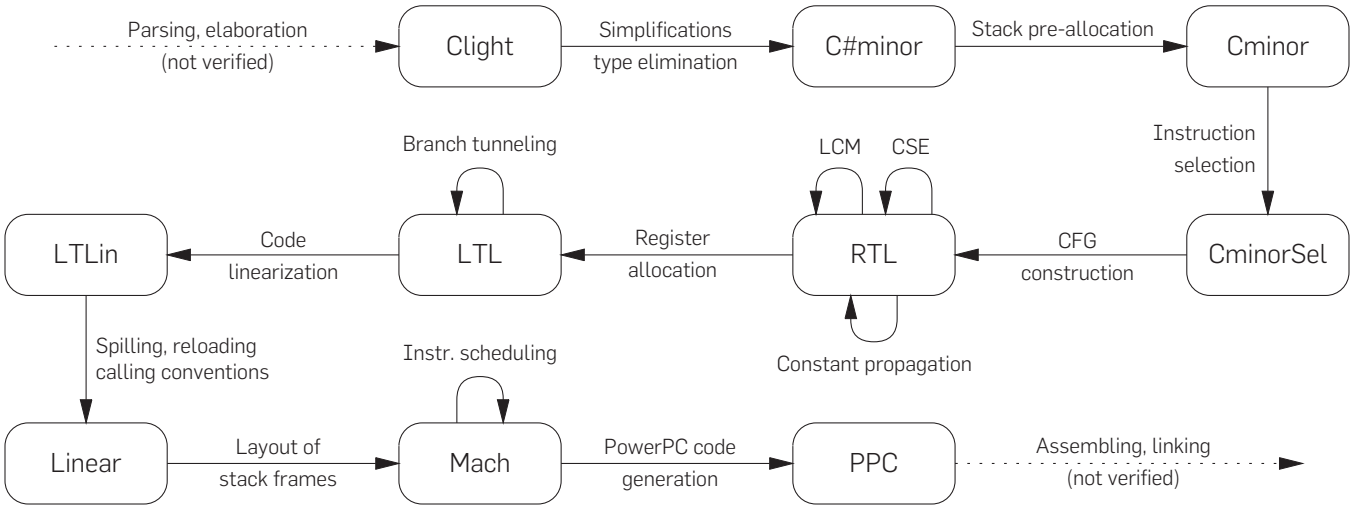
3.2. Compilation passes and intermediate languages

The formally verified part of the CompCert compiler translates from Clight abstract syntax to PPC abstract syntax, PPC being a subset of PowerPC assembly language. As depicted in Figure 1, the compiler is composed of 14 passes that go through eight intermediate languages. Not detailed in Figure 1 are the parts of the compiler that are not verified yet: upstream, a parser, type-checker and simplifier that generates Clight abstract syntax from C source files and is based on the CIL library[21]; downstream, a printer for PPC abstract syntax trees in concrete assembly syntax, followed by generation of executable binary using the system's assembler and linker.

The front-end of the compiler translates away C-specific features in two passes, going through the C#minor and Cminor intermediate languages. C#minor is a simplified, typeless variant of Clight where distinct arithmetic operators are provided for integers, pointers and floats, and C loops are replaced by infinite loops plus blocks and multilevel exits from enclosing blocks. The first pass translates C loops accordingly and eliminates all type-dependent behaviors: operator overloading is resolved; memory loads and stores, as well as address computations, are made explicit. The next intermediate language, Cminor, is similar to C#minor with the omission of the `&` (address-of) operator. Cminor function-local variables do not reside in memory, and their address cannot be taken. However, Cminor supports explicit stack allocation of data in the activation records of functions. The translation from C#minor to Cminor therefore recognizes scalar local variables whose addresses are never taken, assigning them to Cminor local variables and making them candidates for register allocation later; other local variables are stack-allocated in the activation record.

The compiler back-end starts with an instruction selection pass, which recognizes opportunities for using combined arithmetic instructions (add-immediate, not-and, rotate-and-mask, etc.) and addressing modes provided by the target processor. This pass proceeds by bottom-up rewriting of Cminor expressions. The target language is CminorSel, a processor-dependent variant of Cminor that offers additional operators, addressing modes, and a class

Figure 1: Compilation passes and intermediate languages.



of condition expressions (expressions evaluated for their truth value only).

The next pass translates CminorSel to RTL, a classic register transfer language where control is represented as a control-flow graph (CFG). Each node of the graph carries a machine-level instruction operating over temporaries (pseudo-registers). RTL is a convenient representation to conduct optimizations based on dataflow analyses. Two such optimizations are currently implemented: constant propagation and common subexpression elimination, the latter being performed via value numbering over extended basic blocks. A third optimization, lazy code motion, was developed separately and will be integrated soon. Unlike the other two optimizations, lazy code motion is implemented following the verified validator approach.[24]

After these optimizations, register allocation is performed via coloring of an interference graph.[6] The output of this pass is LTL, a language similar to RTL where temporaries are replaced by hardware registers or abstract stack locations. The CFG is then "linearized," producing a list of instructions with explicit labels, conditional and unconditional branches. Next, spills and reloads are inserted around instructions that reference temporaries that were allocated to stack locations, and moves are inserted around function calls, prologues and epilogues to enforce calling conventions. Finally, the "stacking" pass lays out the activation records of functions, assigning offsets within this record to abstract stack locations and to saved callee-save registers, and replacing references to abstract stack locations by explicit memory loads and stores relative to the stack pointer.

This brings us to the Mach intermediate language, which is semantically close to PowerPC assembly language. Instruction scheduling by list or trace scheduling can be performed at this point, following the verified validator approach again.[23] The final compilation pass expands Mach instructions into canned sequences of PowerPC instructions, dealing with special registers such as the condition registers and with irregularities in the PowerPC instruction set. The target language, PPC, accurately models a large subset of PowerPC assembly language, omitting instructions and special registers that CompCert does not generate.

From a compilation standpoint, CompCert is unremarkable: the various passes and intermediate representations are textbook compiler technology from the early 1990s. Perhaps the only surprise is the relatively high number of intermediate languages, but many are small variations on one another: for verification purposes, it was more convenient to identify each variation as a distinct language than as different subsets of a few, more general-purpose intermediate representations.

3.3. Proving the compiler

The added value of CompCert lies not in the compilation technology implemented, but in the fact that each of the source, intermediate and target languages has formally defined semantics, and that each of the transformation and optimization passes is proved to preserve semantics in the sense of Section 2.4.

These semantic preservation proofs are mechanized using the Coq proof assistant. Coq implements the Calculus of Inductive and Coinductive Constructions, a powerful constructive, higher-order logic which supports equally well three familiar styles of writing specifications: by functions and pattern-matching, by inductive or coinductive predicates representing inference rules, and by ordinary predicates in first-order logic. All three styles are used in the CompCert development, resulting in specifications and statements of theorems that remain quite close to what can be found in programming language research papers. In particular, compilation algorithms are naturally presented as functions, and operational semantics use mostly inductive predicates (inference rules). Coq also features more advanced logical features such as higher-order logic, dependent types and an ML-style module system, which we use occasionally in our development. For example, dependent types let us attach logical invariants to data structures, and parameterized modules enable us to reuse a generic dataflow equation solver for several static analyses.

Proving theorems in Coq is an interactive process: some decision procedures automate equational reasoning or Presburger arithmetic, for example, but most of the proofs consist in sequences of "tactics" (elementary proof steps) entered by the user to guide Coq in resolving proof obligations. Internally, Coq builds proof terms that are later rechecked by a small kernel verifier, thus generating very high confidence in the validity of proofs. While developed interactively, proof scripts can be rechecked a posteriori in batch mode.

The whole Coq formalization and proof represents 42,000 lines of Coq (excluding comments and blank lines) and approximately three person-years of work. Of these 42,000 lines, 14% define the compilation algorithms implemented in CompCert, and 10% specify the semantics of the languages involved. The remaining 76% correspond to the correctness proof itself. Each compilation pass takes between 1,500 and 3,000 lines of Coq for its specification and correctness proof. Likewise, each intermediate language is specified in 300 to 600 lines of Coq, while the source language Clight requires 1,100 lines. Additional 10,000 lines correspond to infrastructure shared between all languages and passes, such as the formalization of machine integer arithmetic and of the memory model.

3.4. Programming and running the compiler

We use Coq not only as a prover to conduct semantic preservation proofs, but also as a programming language to write all verified parts of the CompCert compiler. The specification language of Coq includes a small, pure functional language, featuring recursive functions operating by pattern-matching over inductive types (ML- or Haskell-style tree-shaped data types). With some ingenuity, this language suffices to write a compiler. The highly imperative algorithms found in compiler textbooks need to be rewritten in pure functional style. We use persistent data structures based on balanced trees, which support efficient updates without modifying data

in-place. Likewise, a monadic programming style enables us to encode exceptions and state in a legible, compositional manner.

The main advantage of this unconventional approach, compared with implementing the compiler in a conventional imperative language, is that we do not need a program logic (such as Hoare logic) to connect the compiler's code with its logical specifications. The Coq functions implementing the compiler are first-class citizens of Coq's logic and can be reasoned on directly by induction, simplifications, and equational reasoning.

To obtain an executable compiler, we rely on Coq's extraction facility,[15] which automatically generates Caml code from Coq functional specifications. Combining the extracted code with hand-written Caml implementations of the unverified parts of the compiler (such as the parser), and running all this through the Caml compiler, we obtain a compiler that has a standard, `cc`-style command-line interface, runs on any platform supported by Caml, and generates PowerPC code that runs under MacOS X. (Other target platforms are being worked on.)

3.5. Performance

To assess the quality of the code generated by CompCert, we benchmarked it against the GCC 4.0.1 compiler at optimization levels 0, 1, and 2. Since standard benchmark suites use features of C not supported by CompCert, we had to roll our own small suite, which contains some computational kernels, cryptographic primitives, text compressors, a virtual machine interpreter and a ray tracer. The tests were run on a 2 GHz PowerPC 970 "G5" processor.

As the timings in Figure 2 show, CompCert generates code that is more than twice as fast as that generated by GCC without optimizations, and competitive with GCC at optimization levels 1 and 2. On average, CompCert code is only 7% slower than `gcc -O1` and 12% slower than `gcc -O2`. The test suite is too small to draw definitive conclusions, but these results strongly suggest that while CompCert is not going to win a prize in high performance computing, its performance is adequate for critical embedded code.

Compilation times of CompCert are within a factor of 2 of those of `gcc -O1`, which is reasonable and shows that the overheads introduced to facilitate verification (many small passes, no imperative data structures, etc.) are acceptable.

4. REGISTER ALLOCATION

To provide a more detailed example of a verified compilation pass, we now present the register allocation pass of CompCert and outline its correctness proof.

4.1. The RTL intermediate language

Register allocation is performed over the RTL intermediate representation, which represents functions as a CFG of abstract instructions, corresponding roughly to machine instructions but operating over pseudo-registers (also called "temporaries"). Every function has an unlimited supply of pseudo-registers, and their values are preserved across function call. In the following, r ranges over pseudo-registers and l over labels of CFG nodes.

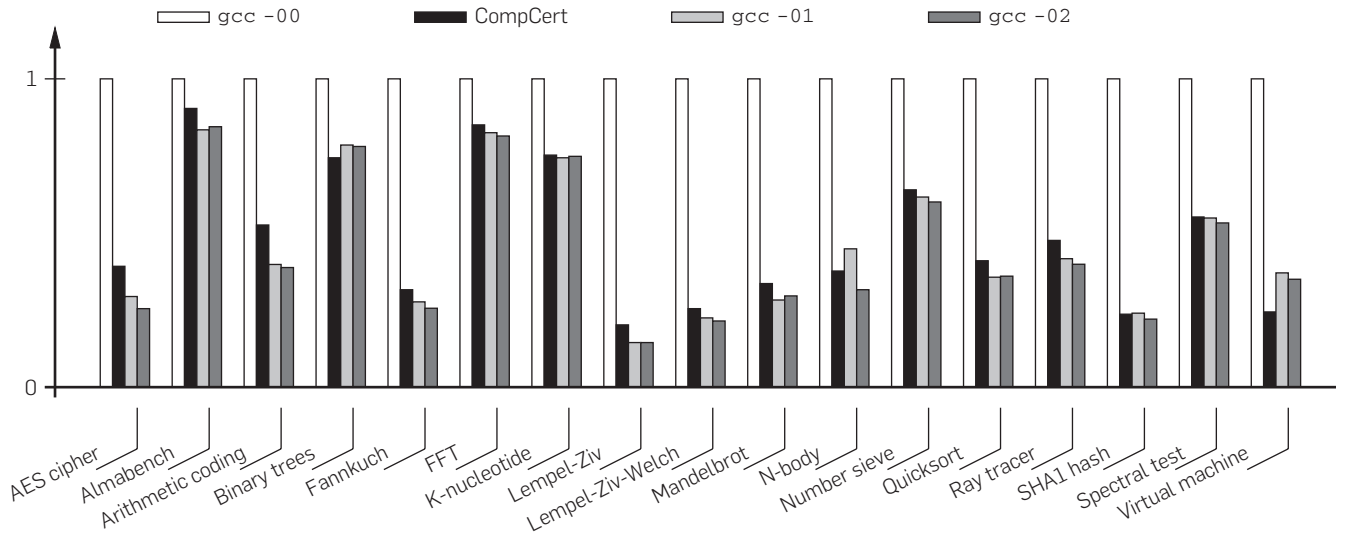
Instructions:

$i ::= \texttt{nop}\,(l)$	no operation (go to l)
$\mid \texttt{op}(op, \vec{r}, r, l)$	arithmetic operation
$\mid \texttt{load}(\kappa, mode, \vec{r}, r, l)$	memory load
$\mid \texttt{store}(\kappa, mode, \vec{r}, r, l)$	memory store
$\mid \texttt{call}(sig, (r \mid id), \vec{r}, r, l)$	function call
$\mid \texttt{tailcall}(sig, (r \mid id), \vec{r})$	function tail call
$\mid \texttt{cond}(cond, \vec{r}, l_{true}, l_{false})$	conditional branch
$\mid \texttt{return} \mid \texttt{return}(r)$	function return

Control-flow graphs:

$g ::= l \mapsto i$	finite map

Figure 2: Relative execution times of compiled code.



Internal functions:

$F ::= \{\texttt{name} = id; \texttt{sig} = sig;$
$\quad \texttt{params} = \vec{r};$ parameters
$\quad \texttt{stacksize} = n;$ size of stack data block
$\quad \texttt{entrypoint} = l;$ label of first instruction
$\quad \texttt{code} = g\}$ control-flow graph

External functions:

$Fe ::= \{\texttt{name} = id; \texttt{sig} = sig\}$

Each instruction takes its arguments in a list of pseudo-registers $\vec{r}$ and stores its result, if any, in a pseudo-register r. Additionally, it carries the labels l of its possible successors. Instructions include arithmetic operations op (with an important special case $\texttt{op}(\texttt{move}, r, r', l)$ representing a register-to-register copy), memory loads and stores (of a quantity κ at the address obtained by applying addressing mode $mode$ to registers $\vec{r}$), conditional branches (with two successors), and function calls, tail-calls, and returns.

An RTL program is composed of a set of named functions, either internal or external. Internal functions are defined within RTL by their CFG, entry point in the CFG, and parameter registers. External functions are not defined but merely declared: they model input/output operations and similar system calls. Functions and call instructions carry signatures sig specifying the number and register classes (`int` or `float`) of their arguments and results.

The dynamic semantics of RTL is specified in small-step operational style, as a labeled transition system. The predicate $G \vdash S \xrightarrow{t} S'$ denotes one step of execution from state S to state S'. The global environment G maps function pointers and names to function definitions. The trace t records the input–output events performed by this execution step: it is empty ($t = \varepsilon$) for all instructions except calls to external functions, in which case t records the function name, parameters, and results of the call.

Execution states S are of the form $S(\Sigma, g, \sigma, l, R, M)$ where g is the CFG of the function currently executing, l the current program point within this function, and σ a memory block containing its activation record. The register state R maps pseudo-registers to their current values (discriminated union of 32-bit integers, 64-bit floats, and pointers). Likewise, the memory state M maps (pointer, memory quantity) pairs to values, taking overlap between multi-byte quantities into account.[14] Finally, Σ models the call stack: it records pending function calls with their (g, σ, l, R) components. Two slightly different forms of execution states, call states and return states, appear when modeling function calls and returns, but will not be described here.

To give a flavor of RTL's semantics, here are two of the rules defining the one-step transition relation, for arithmetic operations and conditional branches, respectively:

$$\frac{g(l) = \texttt{op}(op, \vec{r}, r, l') \quad \texttt{eval\_op}(G, \sigma, op, R(\vec{r})) = v}{G \vdash S(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} S(\Sigma, g, \sigma, l', R\{r \leftarrow v\}, M)}$$

$$g(l) = \texttt{cond}(cond, \vec{r}, l_{true}, l_{false})$$

$$\frac{l' = \begin{cases} l_{true} & \text{if } \texttt{eval\_cond}(cond, R(\vec{r})) = \texttt{true} \\ l_{false} & \text{if } \texttt{eval\_cond}(cond, R(\vec{r})) = \texttt{false} \end{cases}}{G \vdash S(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} S(\Sigma, g, \sigma, l', R, M)}$$

4.2. The register allocation algorithm

The goal of the register allocation pass is to replace the pseudo-registers r that appear in unbounded quantity in the original RTL code by locations l, which are either hardware registers (available in small, fixed quantity) or abstract stack slots in the activation record (available in unbounded quantity). Since accessing a hardware register is much faster than accessing a stack slot, the use of hardware registers must be maximized. Other aspects of register allocation, such as insertion of reload and spill instructions to access stack slots, are left to subsequent passes.

Register allocation starts with a standard liveness analysis performed by backward dataflow analysis. The dataflow equations for liveness are of the form

$$LV(l) = \cup \{T(s, LV(s)) \mid s \text{ successor of } l\} \tag{8}$$

The transfer function $T(s, LV(s))$ computes the set of pseudo-registers live "before" a program point s as a function of the pseudo-registers $LV(s)$ live "after" that point. For instance, if the instruction at s is $\texttt{op}(op, \vec{r}, r, s')$, the result r becomes dead because it is redefined at this point, but the arguments $\vec{r}$ become live. because they are used at this point: $T(s, LV(s)) = (LV(s)\{r\}) \cup \vec{r}$. However, if r is dead "after" ($r \notin L(s)$), the instruction is dead code that will be eliminated later, so we can take $T(s, LV(s)) = LV(s)$ instead.

The dataflow equations are solved iteratively using Kildall's worklist algorithm. CompCert provides a generic implementation of Kildall's algorithm and of its correctness proof, which is also used for other optimization passes. The result of this algorithm is a mapping LV from program points to sets of live registers that is proved to satisfy the correctness condition $LV(l) \supseteq T(s, LV(s))$ for all s successor of l. We only prove an inequation rather than the standard dataflow equation (8) because we are interested only in the correctness of the solution, not in its optimality.

An interference graph having pseudo-registers as nodes is then built following Chaitin's rules,[6] and proved to contain all the necessary interference edges. Typically, if two pseudo-registers r and r' are simultaneously live at a program point, the graph must contain an edge between r and r'. Interferences are of the form "these two pseudo-registers interfere" or "this pseudo-register and this hardware register interfere," the latter being used to ensure that pseudo-registers live across a function call are not allocated to caller-save registers. Preference edges ("these two pseudo-registers should preferably be allocated the same location" or "this pseudo-register should preferably be allocated this location") are also recorded, although they do not affect correctness of the register allocation, just its quality.

The central step of register allocation consists in coloring the interference graph, assigning to each node r a "color" $\varphi(r)$ that is either a hardware register or a stack slot, under the constraint that two nodes connected by an

interference edge are assigned different colors. We use the coloring heuristic of George and Appel.[9] Since this heuristic is difficult to prove correct directly, we implement it as unverified Caml code, then validate its results a posteriori using a simple verifier written and proved correct in Coq. Like many NP-hard problems, graph coloring is a paradigmatic example of an algorithm that is easier to validate a posteriori than to directly prove correct. The correctness conditions for the result φ of the coloring are:

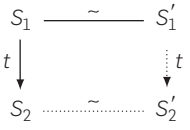
1. $\varphi(r) \neq \varphi(r')$ if r and r' interfere
2. $\varphi(r) \neq l$ if r and l interfere
3. $\varphi(r)$ and r have the same register class (`int` or `float`)

These conditions are checked by boolean-valued functions written in Coq and proved to be decision procedures for the three conditions. Compilation is aborted if the checks fail, which denotes a bug in the external graph coloring routine.

Finally, the original RTL code is rewritten. Each reference to pseudo-register r is replaced by a reference to its location $\varphi(r)$. Additionally, coalescing and dead code elimination are performed. A side-effect-free instruction l: `op`($op, \vec{r}, r, l'$) or l: `load`($\kappa, mode, \vec{r}, r, l'$) is replaced by a no-op l: `nop`(l') if the result r is not live after l (dead code elimination). Likewise, a move instruction l: `op`(`move`, r_s, r_d, l') is replaced by a no-op l: `nop`(l') if $\varphi(r_d) = \varphi(r_s)$ (coalescing).

4.3. Proving semantic preservation

To prove that a program transformation preserves semantics, a standard technique used throughout the CompCert project is to show a simulation diagram: each transition in the original program must correspond to a sequence of transitions in the transformed program that have the same observable effects (same traces of input–output operations, in our case) and preserve as an invariant a given binary relation ~ between execution states of the original and transformed programs. In the case of register allocation, each original transition corresponds to exactly one transformed transition, resulting in the following "lock-step" simulation diagram:



(Solid lines represent hypotheses; dotted lines represent conclusions.) If, in addition, the invariant ~ relates initial states as well as final states, such a simulation diagram implies that any execution of the original program corresponds to an execution of the transformed program that produces exactly the same trace of observable events. Semantic preservation therefore follows.

The gist of a proof by simulation is the definition of the ~ relation. What are the conditions for two states $S(\Sigma, g, \sigma, l, R, M)$ and $S(\Sigma', g', \sigma', l', R', M')$ to be related? Intuitively, since register allocation preserves program structure and control flows, the control points l and l' must be identical, and the CFG g' must be the result of transforming g according to some register allocation φ as described in Section 4.2. Likewise, since register allocation preserves memory stores and allocations, the memory states and stack pointers must be identical: $M' = M$ and $\sigma' = \sigma$.

The nonobvious relation is between the register state R of the original program and the location state R' of the transformed program. Given that each pseudo-register r is mapped to the location $\varphi(r)$, we could naively require that $R(r) = R'(\varphi(r))$ for all r. However, this requirement is much too strong, as it essentially precludes any sharing of a location between two pseudo-registers whose live ranges are disjoint. To obtain the correct requirement, we need to consider what it means, semantically, for a pseudo-register to be live or dead at a program point l. A dead pseudo-register r is such that its value at point l has no influence on the program execution, because either r is never read later, or it is always redefined before being read. Therefore, in setting up the correspondence between register and location values, we can safely ignore those registers that are dead at the current point l. It suffices to require the following condition:

$R(r) = R'(\varphi(r))$ for all pseudo-registers r live at point l.

Once the relation between states is set up, proving the simulation diagram above is a routine case inspection on the various transition rules of the RTL semantics. In doing so, one comes to the pleasant realization that the dataflow inequations defining liveness, as well as Chaitin's rules for constructing the interference graph, are the minimal sufficient conditions for the invariant between register states R, R' to be preserved in all cases.

5. CONCLUSIONS AND PERSPECTIVES

The CompCert experiment described in this paper is still ongoing, and much work remains to be done: handle a larger subset of C (e.g. including `goto`); deploy and prove correct more optimizations; target other processors beyond PowerPC; extend the semantic preservation proofs to shared-memory concurrency, etc. However, the preliminary results obtained so far provide strong evidence that the initial goal of formally verifying a realistic compiler can be achieved, within the limitations of today's proof assistants, and using only elementary semantic and algorithmic approaches. The techniques and tools we used are very far from perfect—more proof automation, higher-level semantics and more modern intermediate representations all have the potential to significantly reduce the proof effort—but good enough to achieve the goal.

Looking back at the results obtained, we did not completely rule out all uncertainty concerning the correctness of the compiler, but reduced the problem of trusting the whole compiler down to trusting the following parts:

1. The formal semantics for the source (Clight) and target (PPC) languages.
2. The parts of the compiler that are not verified yet: the

CIL-based parser, the assembler, and the linker.

3. The compilation chain used to produce the executable for the compiler: Coq's extraction facility and the Caml compiler and run-time system. (A bug in this compilation chain could invalidate the guarantees obtained by the correctness proof.)
4. The Coq proof assistant itself. (A bug in Coq's implementation or an inconsistency in Coq's logic could falsify the proof.)

Issue (4) is probably the least concern: as Hales argues,[10] proofs mechanically checked by a proof assistant that generates proof terms are orders of magnitude more trustworthy than even carefully hand-checked mathematical proofs.

To address concern (3), ongoing work within the CompCert project studies the feasibility of formally verifying Coq's extraction mechanism as well as a compiler from Mini-ML (the simple functional language targeted by this extraction) to Cminor. Composed with the CompCert back-end, these efforts could eventually result in a trusted execution path for programs written and verified in Coq, like CompCert itself, therefore increasing confidence further through a form of bootstrapping.

Issue (2) with the unverified components of CompCert can obviously be addressed by reimplementing and proving the corresponding passes. Semantic preservation for a parser is difficult to define, let alone prove: what is the semantics of the concrete syntax of a program, if not the semantics of the abstract syntax tree produced by parsing? However, several of the post-parsing elaboration steps performed by CIL are amenable to formal proof. Likewise, proving the correctness of an assembler and linker is feasible, if unexciting.

Perhaps the most delicate issue is (1): how can we make sure that a formal semantics agrees with language standards and common programming practice? Since the semantics in question are small relative to the whole compiler, manual reviews by experts, as well as testing conducted on executable forms of the semantics, could provide reasonable (but not formal) confidence. Another approach is to prove connections with alternate formal semantics independently developed, such as the axiomatic semantics that underline tools for deductive verification of programs (see Appel and Blazy[2] for an example). Additionally, this approach constitutes a first step towards a more ambitious, long-term goal: the certification, using formal methods, of the verification tools, code generators, compilers and runtime systems that participate in the development, validation and execution of critical software.

Acknowledgments

The author thanks S. Blazy, Z. Dargaye, D. Doligez, B. Grégoire, T. Moniot, L. Rideau, and B. Serpette for their contributions to the CompCert development, and A. Appel, Y. Bertot, E. Ledinot, P. Letouzey, and G. Necula for their suggestions, feedback, and help. This work was supported by Agence Nationale de la Recherche, grant number ANR-05-SSIA-0019.

References

1. Appel, A.W. Foundational proof-carrying code. In *Logic in Computer Science 2001* (2001), IEEE, 247–258.
2. Appel, A.W., Blazy, S. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *LNCS* (2007), Springer, 5–21.
3. Bertot, Y., Castéran, P. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions* (2004), Springer.
4. Blazy, S., Dargaye, Z., Leroy, X. Formal verification of a C compiler front-end. In *FM 2006: International Symposium on Formal Methods*, volume 4085 of *LNCS* (2006), Springer, 460–475.
5. Blazy, S., Leroy, X. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reasoning* (2009). Accepted for publication, to appear.
6. Chaitin, G.J. Register allocation and spilling via graph coloring. In *1982 SIGPLAN Symposium on Compiler Construction* (1982), ACM, 98–105.
7. Coq development team. The Coq proof assistant. Available at http://coq.inria.fr/, 1989–2009.
8. Dave, M.A. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes 28*, 6 (2003), 2.
9. George, L., Appel, A.W. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst. 18*, 3 (1996), 300–324.
10. Hales, T.C. Formal proof. *Notices AMS 55*, 11 (2008), 1370–1380.
11. Leroy, X. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd Symposium on the Principles of Programming Languages* (2006), ACM, 42–54.
12. Leroy, X. The CompCert verified compiler, software and commented proof. Available at http://compcert.inria.fr/, Aug. 2008.
13. Leroy, X. A formally verified compiler back-end. arXiv:0902.2137 [cs]. Submitted, July 2008.
14. Leroy, X., Blazy, S. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning 41*, 1 (2008), 1–31.
15. Letouzey, P. Extraction in Coq: An overview. In *Logic and Theory of Algorithms, Computability in Europe, CiE 2008*, volume 5028 of *LNCS* (2008), Springer, 359–369.
16. McCarthy, J., Painter, J. Correctness of a compiler for arithmetical expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics* (1967), AMS, 33–41.
17. Milner, R., Weyhrauch, R. Proving compiler correctness in a mechanized logic. In *Proceedings of 7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence* (1972), Edinburgh University Press, 51–72.
18. Morrisett, G., Walker, D., Crary, K., Glew, N. From System F to typed assembly language. *ACM Trans. Prog. Lang. Syst. 21*, 3 (1999), 528–569.
19. Necula, G.C. Proof-carrying code. In *24th Symposium on the Principles of Programming Languages* (1997), ACM, 106–119.
20. Necula, G.C. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000* (2000), ACM, 83–95.
21. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, volume 2304 of *LNCS* (2002), Springer, 213–228.
22. Pnueli, A., Siegel, M., Singerman, E. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *LNCS* (1998), Springer, 151–166.
23. Tristan, J.-B., Leroy, X. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th Symposium of the Principles of Programming Languages* (2008), ACM, 17–27.
24. Tristan, J.-B., Leroy, X. Verified validation of lazy code motion. In *Programming Language Design and Implementation 2009* (2009), ACM. To appear.

Xavier Leroy (xavier.leroy@inria.fr) INRIA Paris-Rocquencourt, France