Formal certification of a compiler back-end, or: programming a compiler with a proof assistant

Xavier Leroy

To cite this version:

HAL Id: inria-00000963
https://inria.hal.science/inria-00000963
Submitted on 6 Jan 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Formal Certification of a Compiler Back-end

or: Programming a Compiler with a Proof Assistant

Xavier Leroy

INRIA Rocquencourt
Xavier.Leroy@inria.fr

Abstract
This paper reports on the development and formal certification (proof of semantic preservation) of a compiler from Cminor (a C-like imperative language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. Such a certified compiler is useful in the context of formal methods applied to the certification of critical software: the certification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Categories and Subject Descriptors F.3.1 [Logics and meanings of programs]: Specifying and verifying and reasoning about programs—Mechanical verification.; D.2.4 [Software engineering]: Software/program verification—Correctness proofs, formal methods, reliability; D.3.4 [Programming languages]: Processors—Compilers, optimization

General Terms Languages, Reliability, Security, Verification.

Keywords Certified compilation, semantic preservation, program proof, compiler transformations and optimizations, the Coq theorem prover.

1. Introduction
Can you trust your compiler? Compilers are 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. We all know horror stories of bugs in compilers silently turning a correct program into an incorrectly executable.

For low-assurance software, validated only by testing, the impact of compiler bugs is negligible: what is tested is the executable code produced by the compiler; rigorous testing will expose errors in the compiler along with errors in the source program. The picture changes dramatically for critical, high-assurance software whose certification at the highest levels requires the use of formal methods (model checking, program proof, etc). What is formally verified using formal methods is almost universally the source code; bugs in the compiler used to turn this verified source into an executable can potentially invalidate all the guarantees so painfully obtained using formal methods. In other terms, from a formal methods perspective, the compiler is a weak link between a source program that has been formally verified and a hardware processor that, more and more often, has also been formally verified. The safety-critical software industry is aware of this issue and uses a variety of techniques to alleviate it, 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 issue, 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. Many different approaches have been proposed and investigated, including on-paper and on-machine proofs of semantic preservation, proof-carrying code, credible compilation, translation validation, and type-preserving compilers. (These approaches are compared in section 2.) For the last two years, we have been working on the development of a realistic, certified compiler. By certified, we mean a compiler that is accompanied by a machine-checked proof of semantic preservation. By realistic, we mean a compiler that compiles a language commonly used for critical embedded software (a subset of C) down to assembly code for a processor commonly used in embedded systems (the PowerPC), and that generates reasonably efficient code.

This paper reports on the completion of one half of this program: the certification, using the Coq proof assistant [2], of a lightly-optimizing back-end that generates PowerPC assembly code from a simple imperative intermediate language called Cminor. A front-end translating a subset of C to Cminor is being developed and certified, and will be described in a forthcoming paper.

While there exists a considerable body of earlier work on machine-checked correctness proofs of parts of compilers (see section 7 for a review), our work is novel in two ways. First, recent work tends to focus on a few parts of a compiler, mostly optimizations and the underlying static analyses [18, 6]. In contrast, our work is modest on the optimization side, but emphasizes the certification of a complete compilation chain from a structured imperative language down to assembly code through 4 intermediate languages. We found that many of the non-optimizing translations performed, while often considered obvious in compiler literature, are surprisingly tricky to formally prove correct. The other novelty of our work is that most of the compiler is written directly in the Coq specification language, in a purely functional style. The executable compiler is obtained by automatic extraction of Caml code from this specification. This approach has never been applied before to a program of the size and complexity of an optimizing compiler.
Besides proclaiming “we have done it!”, the purpose of this paper is to put our proof development in more general perspectives (within the page limits of a conference paper). One such perspective is to revisit classic compiler technology from the semanticist’s eye, in particular by distinguishing clearly between the correctness-relevant and the performance-relevant aspects of compilation algorithms, which are inextricably mixed in compiler literature. The other perspective is on the suitability of a proof assistant such as Coq not only for proving compiler-like symbolic computations, but even for programming them.

2. Certified, certifying, verified compilation

**General setting** The formal verification of a compiler consists of even for programming them. Relevant and the performance-relevant aspects of compilation algorithms, which are inextricably mixed in compiler literature. The other perspective is on the suitability of a proof assistant such as Coq not only for proving compiler-like symbolic computations, but even for programming them.

2. Certified, certifying, verified compilation

**General setting** The formal verification of a compiler consists of establishing a given correctness property \(\text{Prop}(S,C)\) between a source program \(S\) and its compiled code \(C\). Examples of correctness properties include:

1. “\(S\) and \(C\) are observationally equivalent”;
2. “if \(S\) has well-defined semantics (does not go wrong), then \(S\) and \(C\) are observationally equivalent”;
3. “if \(S\) has well-defined semantics and satisfies the functional specification \(\text{Spec}\), then \(C\) satisfies \(\text{Spec}\)”;
4. “if \(S\) is type- and memory-safe, then so is \(C\)”;
5. “\(C\) is type- and memory-safe”.

The correctness property we will use in the remainder of this paper is of the form (2). Full observational equivalence (property 1) is generally too strong: it requires the compiled code to go wrong whenever the source program does according to its semantics. In practice, compilers are free to generate arbitrary code for a source program that has undefined behavior. Preservation of a specification (property 3) is implied by property 2 if the specification \(\text{Spec}\) depends only on the observable behavior of the program, which is often the case. Therefore, establishing property 2 once and for all spares us from establishing property 3 for many specifications of interest. Property 4 is typical of a type-preserving compiler, and is an instance of property 3. Finally, property 5 is an example where the source code \(S\) plays no role in the property; it is typical of a compiler that either establishes the property of interest (type safety) directly on the compiled code, or establishes it on the source program and preserves it during compilation.

In our general setting, a compiler \(\text{Comp}\) is a total function from source programs \(S\) to either compiled code (written \(\text{Comp}(S) = \text{Some}(C)\)) or an error (written \(\text{Comp}(S) = \text{None}\)). The error case must be taken into account, since compilers can fail to produce compiled code, for instance if the source program is incorrect (syntax error, type error, etc.), but also if it exceeds the capacities of the compiler (see section 4.6 for an example).

**Certified compilers** Using the definitions above, a certified compiler is any compiler \(\text{Comp}\) accompanied with a formal proof of the following theorem:

\[
\forall S, C, \quad \text{Comp}(S) = \text{Some}(C) \Rightarrow \text{Prop}(S, C) \quad (i)
\]

In other terms, either the compiler reports an error or produces code that satisfies the desired correctness property. Notice that the trivial compiler \(\text{Comp}(S) = \text{None}\) for all \(S\) is indeed certified, though 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 non-formal methods such as testing. The important feature, from a formal methods standpoint, is that the compiler never silently produces incorrect code.

**Proof-carrying code** Proof-carrying code [23] and credible compilation [30] make use of a certifying compiler, which is a function \(\text{CComp}\) that either fails (\(\text{CComp}(S) = \text{None}\)) or returns both a compiled code \(C\) and a proof \(\pi\) of the property \(\text{Prop}(S,C)\) \((\text{CComp}(S) = \text{Some}(C,\pi))\). The proof \(\pi\) can be checked independently by the code user; there is no need to trust the code producer, nor to formally verify the compiler itself. Of course, the certifying compiler can produce incorrect “proofs” \(\pi\) that do not establish \(\text{Prop}(S,C)\). This is again a quality of implementation issue: wrong code \(C\) will not be silently accepted as long as the code user checks the proof.

**Translation validation** In the translation validation approach [28, 24, 35, 31], the standard compiler \(\text{Comp}\) is complemented by a verifier: a boolean-valued function \(\text{Verif}(S,C)\) that verifies the property \(\text{Prop}(S,C)\) by static analysis of \(S\) and \(C\). To obtain formal guarantees, the verifier must be itself certified. That is, we must prove that

\[
\forall S, C, \quad \text{Verif}(S,C) = \text{true} \Rightarrow \text{Prop}(S,C)
\]

However, no formal verification of the compiler is needed. There are no guarantees that the code generated by the compiler will always pass the verifier: this is, again, a quality of implementation issue, not a correctness issue.

**Unifying PCC and translation validation** In practical uses of proof-carrying code, the certifying compiler is not required to generate a full proof term \(\pi\) of \(\text{Prop}(S,C)\): it is sufficient to generate enough hints so that such a full proof can be reconstructed cheaply on the client side by a specialized prover [25]. Symmetrically, practical uses of translation validation can take advantage of annotations produced by the compiler (such as debugging information) to help building a correspondence between \(S\) and \(C\). A typical middle ground between proof-carrying code and translation validation is type-checking of compiled code, as in Java bytecode verification [32] or typed assembly language [22]: the compiler annotates the code with enough type declarations that a full typing derivation can be reconstructed easily at the client side.

To account for these mixed approaches, we consider that a certifying compiler \(\text{CComp}(S)\) returns either \(\text{None}\) or \(\text{Some}(C, A)\), where \(A\) is an annotation (also called a certificate) that is passed to the verifier, in addition to \(S\) and \(C\), and helps it establish the desired property. The correctness theorem for the verifier becomes

\[
\forall S, A, C, \quad \text{Verif}(S, C, A) = \text{true} \Rightarrow \text{Prop}(S,C) \quad (ii)
\]

In the case of pure proof-carrying code, \(A\) is a proof term and \(\text{Verif}\) a general proof-checker; in the case of pure translation validation, \(A\) is empty; finally, in the case of bytecode verification and typed assembly language, \(A\) is a set of type annotations and \(\text{Verif}\) performs type-checking with partial type inference.

**Bridging certified and certifying compilers** With this reformulation, we can bridge formally the certified compiler approach and the certifying/verified compiler approach. If \(\text{CComp}\) is a certifying compiler and \(\text{Verif}\) a correct verifier, the following function is a certified compiler:

\[
\text{Comp}(S) = \begin{cases} \text{match } \text{CComp}(S) \text{ with} & \\
\text{None } \rightarrow \text{None} & \\
\text{Some}(C, A) & \text{if } \text{Verif}(S, C, A) \text{ then } \text{Some}(C) \text{ else None} \\
\end{cases}
\]

Theorem (i) follows immediately from theorem (ii).

Symmetrically, let \(\text{Comp}\) be a certified compiler and \(\Pi\) be a Coq proof term for theorem (i). Via the Curry-Howard isomorphism, \(\Pi\) is a function that takes \(S, C\) and a proof of \(\text{Comp}(S) = \text{Some}(C)\) and returns a proof of \(\text{Prop}(S,C)\). A certifying compiler can be defined as follows:
from languages L is a certified compiler from C to CComp S, a certified verifier. The latter can reduce the amount of code that can be constructed as follows:

\[ C\text{Comp}(S) = \]
\[ \text{match } C\text{Comp}(S) \text{ with} \]
\[ \text{None } \rightarrow \text{None} \]
\[ \text{Some}(C) \rightarrow \text{Some}(C, \Pi S C \pi_{eq}) \]

(Here, \( \pi_{eq} \) is a proof term for the proposition \( C\text{Comp}(S) = \text{Some}(C) \), which trivially holds in the context of the match above. Actually building this proof term in Coq requires additional baggage in the definition above that we omitted for simplicity.) The accompanying verifier is the Coq proof checker, as in the pure proof-carrying code approach. While the annotation produced by CComp looks huge (it contains the proof of correctness for a function), the former can be specialized for \( C, S \). However, decomposition in this manner. If \( C\text{Comp}_1 \) and \( C\text{Comp}_2 \) are certified compilers from languages \( L_1 \) to \( L_2 \) and \( L_2 \) to \( L_3 \), respectively, their (monadic) composition

\[ C\text{Comp}(S) = \]
\[ \text{match } C\text{Comp}_1(S) \text{ with} \]
\[ \text{None } \rightarrow \text{None} \]
\[ \text{Some}(I) \rightarrow \text{Comp}_2(I) \]

is a certified compiler from \( L_1 \) to \( L_3 \), provided that the property \( Prop \) is transitive, that is \( Prop(S, I) \) and \( Prop(I, C) \) imply \( Prop(S, C) \). This is the case for the five examples of \( Prop \) we gave earlier.

Similarly, if \( C\text{Comp}_1 \) and \( C\text{Comp}_2 \) are certifying compilers from languages \( L_1 \) to \( L_2 \) and \( L_2 \) to \( L_3 \), and \( Verif_1 \), \( Verif_2 \) are the accompanying verifiers, a certifying compiler \( C\text{Comp} \) verifier from \( L_1 \) to \( L_3 \) can be constructed as follows:

\[ C\text{Comp}(S) = \]
\[ \text{match } C\text{Comp}_1(S) \text{ with} \]
\[ \text{None } \rightarrow \text{None} \]
\[ \text{Some}(I) \rightarrow \text{Comp}_2(I) \]

\[ \text{Verif}(S, C, (A_1, I, A_2)) = \]
\[ \text{Verif}(S, I, A_1) \land \text{Verif}_2(I, C, A_2) \]

Summary The conclusions of this discussion are simple and define the methodology we have followed to certify our compiler back-end. First, a certified compiler can be structured as a composition of compilation passes, as usual; each pass can be proved correct independently. Once, 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 certified verifier. The latter can reduce the amount of code that needs to be proved. Finally, provided the proof of \( i \) is carried out in a prover such as Coq that supports proof terms and follows the Curry-Howard isomorphism, a certified compiler can at least theoretically be used in a context of proof-carrying code.

3. The languages
3.1 The source language: Cminor

The input language of our back-end is called Cminor. It is a simple, low-level imperative language inspired from C and C++ [26].

Syntax The language is classically structured in expressions, statements, functions and programs.
Expressions evaluate to values \( v \), which range over the discriminated union of 32-bit integers, 64-bit floats, pointers (pairs of a memory block reference and a byte offset), and \texttt{undef} (representing e.g. the value of an uninitialized variable). \( sp \) is the reference to the stack block for the current function. The global environment \( G \) maps symbols (function or global variable name) to values, and function pointers to function definitions. We therefore have function pointers as first-class values, but in a “Harvard” model where functions and data reside in different memory spaces. \( E \) is the local environment, mapping local variables to values. \( L \) gives values to \texttt{let}-bound variables inside expressions. For statement evaluations, \texttt{out} (the “outcome” of the evaluation) expresses how the statement terminated: either normally by falling through the next statement or prematurely by an \texttt{exit} or \texttt{return} statement.

The only non-obvious components of the evaluation rules are the initial and final memory states, \( M \) and \( M' \). Memory states map block references to memory blocks, consisting of lower and upper bounds (fixed at allocation-time) plus a mapping from byte offsets to their current contents. A full description of the memory model is given in [4] and goes beyond the scope of this paper.

Cminor is equipped with a trivial type system having only two types: \texttt{int} and \texttt{float}. (Memory addresses have static type \texttt{int}.) In addition, function definitions and function calls are annotated with signatures \( \texttt{sig} \) giving the number and types of arguments, and an optional type for the result. All operators that reflect what the target processor can do in one instruction.

### 3.2 Intermediate languages: RTL and variants

**RTL** Following a long-established tradition, the intermediate languages of our compiler are of the “register transfer language” kind, also known as “3-address code”. The first intermediate language, called RTL, represents functions as a control-flow graph (CFG) of abstract instructions operating over pseudo-registers (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.

**RTL instructions**

```plaintext
i := \text{nop}(l) \quad \text{no operation (go to)}
| \text{op}(optype, r, r, l) \quad \text{arithmetic operation}
| \text{load}(chunk, mode, r, r, l) \quad \text{memory load}
| \text{store}(chunk, mode, r, r, l) \quad \text{memory store}
| \text{call}(sig, \{ r | id \}, r, r, l) \quad \text{function call}
| \text{cond}(cond, \{ r | block \}, \{ false \}) \quad \text{conditional branch}
| \text{return}(r) \quad \text{function return}
```

**RTL control-flow graph**

```plaintext
g := \text{if} n \text{ then} i \text{ else} \text{undefined}
```

Each instruction takes its arguments in a list of pseudo-registers \( \vec{r} \) and stores its result, if any, in a pseudo-register. Additionally, it carries the set of its possible successors. We use instructions rather than basic blocks as nodes of the control-flow graph because this simplifies the semantics and reasoning over static analyses, without significantly slowing compilation [16].

**RTL and its variants**

RTL and its variants are statically typed using the same trivial type system as Cminor. Each register can be assigned a type \texttt{int} or \texttt{float} based on the signature of the function and its uses within the function.
The dynamic semantics of RTL is an original (to the best of our knowledge) combination of small-step and big-step semantics, expressed by three inductive predicates:

\[ G, g, \text{sp} \vdash l, R, M \rightarrow l', R', M' \quad \text{(one instruction)} \]
\[ G, g, \text{sp} \vdash l, R, M \rightarrow \delta, R', R', M' \quad \text{(several instructions)} \]
\[ G \vdash \text{fn}(\vec{r}), M \Rightarrow v, M' \quad \text{(function call)} \]

Here, \( R \) ranges over mappings from pseudo-registers to values. Each instruction is executed as one transition over the triple (current label, current values of registers, current memory state). However, function calls (the call instruction) are also executed as one transition, thus hiding the sequence of transitions performed by the called function. The following selected evaluation rules should give the flavor of the semantics.

\[
g(l) = \text{op}(\text{op}, \vec{r}, r_d, l') \quad v = \text{eval}_{\text{op}}(\text{op}, R(\vec{r}))
\]
\[
g(l) = \text{call}(\text{sig}, r_f, \vec{r}, r_d, l') \quad G(R(\vec{r})) = \text{fn}
\]
\[
G, g, \text{sp} \vdash l, R, M \rightarrow l', R\{r_d \leftarrow v\}, M
\]
\[
G \vdash \text{fn}(\vec{r}), M \Rightarrow v, M'
\]

This “mostly small-steps” semantics is perfectly suited to reasoning over intra-procedural analyses and transformations. An induction over a derivation of an evaluation produces exactly the expected proof obligations: one for each single-instruction transition, one for the sequential composition of two transitions, and one for the execution of a function body. Unlike in pure small-step semantics, the call stack does not need to be exposed in the semantics nor in the proofs.

**LTL**

Several variants of RTL are used as intermediate steps towards PowerPC assembly code. These variants progressively refine the notion of pseudo-register (eventually mapped to hardware registers and stack slots in the activation record), as depicted in Figure 1, as well as the representation of control (eventually mapped to a linear list of instructions with explicit labels and branches). The first such variant is LTL (Location Transfer Language). Control is still represented as a flow graph, but the nodes are now basic blocks instead of individual instructions. (The transformation from RTL to LTL inserts reload and spill instructions; such insertions are easier to perform on basic blocks.) More importantly, the pseudo-registers are replaced by locations, which are either hardware processor registers \( r \) (taken from a fixed, finite set of integer and float registers) or stack slots \( s \).

**LTL basic blocks**:

\[
b ::= \text{setstack}(r, s) \quad \text{register to slot move}
\]
\[
\quad \text{getstack}(s, r) \quad \text{slot to register move}
\]
\[
\quad \text{op}(\text{op}, \vec{r}, r) \quad \text{arithmetic operation}
\]
\[
\quad \text{store}(\text{chunk}, \text{mode}, \vec{r}, r) \quad \text{memory load}
\]
\[
\quad \text{call}(\text{sig}, (r \mid id)) \quad \text{function call}
\]
\[
\quad \text{goto}(l) \quad \text{unconditional branch}
\]
\[
\quad \text{cond}(\text{cond}, \vec{F}, l_{\text{true}}, l_{\text{false}}) \quad \text{conditional branch}
\]
\[
\quad \text{return} \quad \text{function return}
\]

**LTL control-flow graph**:

\[
g ::= l \rightarrow b \quad \text{finite map}
\]

**LTL functions**:

\[
\text{fn} ::= \text{fun}(\vec{r}) \quad \text{sig}
\]

Stack slots:

\[
s ::= \text{Local}(\tau, \delta) \quad \text{local variables}
\]
\[
\text{Incoming}(\tau, \delta) \quad \text{incoming parameters}
\]
\[
\text{Outgoing}(\tau, \delta) \quad \text{outgoing arguments}
\]

In stack slots, \( \tau \) is the intended type of the slot (int or float) and \( \delta \) an integer intended as a word offset in the corresponding area of the activation record.

Note that functions as well as call and return instructions no longer list the registers where arguments, parameters and return values reside. Instead, arguments and return values are passed in fixed registers and stack slots determined by the calling conventions of the processor.

Stack slots are not yet mapped to memory locations in the activation record; their values, like those of processor registers, is found in a mapping \( L \) of locations to values that plays the same role as the mapping \( R \) of pseudo-registers to values in the RTL semantics. However, the behavior of location maps \( L \) reflects what will happen when stack slots are later mapped to memory. This is especially apparent in the transition rule for function calls in LTL:

\[
G(L(\vec{r})) = \text{fn} \quad \text{fn}.\text{sig} = \text{sig}
\]
\[
G, g, \text{sp} \vdash \text{call}(\text{sig}, r_f, \vec{r}, r_d, l') : b, L, M \rightarrow b, \text{exitfun}(L, L'), M'
\]
\[
G, \text{fn}.\text{graph}, sp \vdash g(\text{fn}.\text{start}), L, M_1 \downarrow \text{return}, L', M_2
\]
\[
M' = \text{free}(M_2, sp)
\]

The \text{entryfun} and \text{exitfun} functions capture the behavior of locations across function calls: processor registers are global but some are preserved by the callee; \text{Local} and \text{Incoming} slots of the caller are preserved, and the \text{Incoming} slots of the callee are the \text{Outgoing} slots of the caller.

**Location i**

<table>
<thead>
<tr>
<th>entryfun(L)(l)</th>
<th>exitfun(L, L')(l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(r) ) if ( r ) is callee-save</td>
<td>( L(r) ) if ( r ) is caller-save</td>
</tr>
<tr>
<td>\text{Local}(\tau, \delta)</td>
<td>\text{Local}(\tau, \delta)</td>
</tr>
<tr>
<td>\text{Incoming}(\tau, \delta)</td>
<td>\text{Incoming}(\tau, \delta)</td>
</tr>
<tr>
<td>\text{Outgoing}(\tau, \delta)</td>
<td>\text{Incoming}(\tau, \delta)</td>
</tr>
</tbody>
</table>

Another peculiarity of locations is that distinct stack slots may overlap, that is, they will be later mapped to overlapping memory areas. For instance, \text{Outgoing(float, 0)} overlaps with \text{Outgoing(int, 0)} and \text{Outgoing(int, 1)}: a write to the former invalidates the values of the latter, and conversely. This is reflected in the weak “good variable” property for location maps: \( L\{\text{loc}_1 \leftarrow v\}(\text{loc}_2) = L(\text{loc}_2) \) only if \( \text{loc}_1 \) and \( \text{loc}_2 \) do not overlap; it does not suffice that \( \text{loc}_1 \neq \text{loc}_2 \).

**Linear**

The next intermediate language in our series is Linear, a variant of LTL where the control-flow graph and the basic blocks are replaced by a list of instructions with explicit labels and branches. (Non-branch instructions continue at the next instruction in the list.)

**Linear instructions**:

\[
i ::= \text{setstack}(r, s) \quad \text{register to slot move}
\]
\[
\quad \text{getstack}(s, r) \quad \text{slot to register move}
\]
\[
\quad \text{op}(\text{op}, \vec{r}, r) \quad \text{arithmetic operation}
\]
\[
\quad \text{load}(\text{chunk}, \text{mode}, \vec{r}, r) \quad \text{memory load}
\]
\[
\quad \text{store}(\text{chunk}, \text{mode}, \vec{r}, r) \quad \text{memory store}
\]
\[
\quad \text{call}(\text{sig}, (r \mid id)) \quad \text{function call}
\]
\[
\quad \text{label}(l) \quad \text{branch target label}
\]
Linear functions:

\[
\begin{align*}
goto(l) & \quad \text{unconditional branch} \\
n \oplus \text{cond}(l, \bar{r}, \bar{l}_{\text{true}}) & \quad \text{conditional branch} \\
\text{return} & \quad \text{function return}
\end{align*}
\]

The dynamic semantics of Linear, like those of RTL and LTL, is "mostly small-steps": each instruction is a transition in the semantics, but a \text{call} instruction transitions directly to the state at function return.

**Mach** The last intermediate language in our gentle descent towards PowerPC assembly is called Mach. It is a variant of Linear where the three infinite supplies of stack slots (local, incoming and outgoing) are mapped to actual memory locations in the stack frames of the callee (for local and outgoing slots) or the caller (for incoming slots).

Mach instructions:

\[
\begin{align*}
i & \triangleq \text{setstack}(r, \tau, \delta) \quad \text{register to stack move} \\
\text{getstack}(\tau, \delta, r) & \quad \text{stack to register move} \\
\text{getparent}(\tau, \delta, r) & \quad \text{caller’s stack to register move} \\
\ldots & \quad \text{as in Linear}
\end{align*}
\]

In the three new move instructions, \(\tau\) is the type of the data moved and \(\delta\) its word offset in the corresponding activation record. The semantics of \text{call} is also modified so that all hardware registers are global and shared between caller and callee: there is no automatic restoration of callee-save registers at function return; instead, the Mach code producer must produce appropriate setstack and getstack instructions to save and restore used callee-save registers at function prologues and epilogues.

The semantics for Mach is of the form \(G, fn, sp \vdash \bar{r}, R, M \rightarrow \bar{r}', R', M'\), where \(R\) is a mapping from hardware registers to values, and setstack, getstack and getparent are interpreted as \(sp\)-relative memory accesses. An alternate semantics, described in section 4.6, is also used as an intermediate step in one proof.

### 3.3 The target language: PowerPC macro-assembler

The target language for our compiler is abstract syntax for a subset of the PowerPC assembly language (90 instructions of the 200+ offered by the processor). The semantics is purely small-step and defines a transition over the state of registers and memory for every instruction. The registers modeled are: all general-purpose integer and float registers, the PC, LR and CTR special registers, and bits 0 to 3 of the condition register.

The semantics is (to the best of our knowledge) faithful to the actual behavior of PowerPC instructions, with one exception: the \text{fma} and \text{fms} instructions (combined multiply-add and multiply-sub over floats) are treated as producing the same results as a normal multiply followed by a normal addition or subtraction, ignoring the fact that \text{fma} and \text{fms} skip a rounding step on the result of the multiply. Depending on how the certification of the source program treats floating-point numbers (e.g. as IEEE floats or as intervals of real numbers), this infidelity can be semantically correct or not. It is however trivial to turn off the generation of \text{fma} and \text{fms} instructions in case exact results at the bit level are required.

Our PowerPC assembly language features a handful of macro-instructions that expand to canned sequences of actual instruction during pretty-printing of the abstract syntax to concrete assembly syntax. These macro-instructions include allocation and deallocation of the stack frame (mapped to arithmetic on the stack pointer register), integer to float conversions (mapped to complicated bit-level manipulations of IEEE floats), and loading of a floating-point literal (mapped to a load from a memory-allocated constant). The reason for treating these operations as basic instructions is that it was deemed too difficult and not worthwhile to certify the correctness of the corresponding canned sequences. For instance, proving
the integer to float conversions necessitates a full-bit-level formal-  
zation of IEEE float arithmetic, which we have not done (we sim-  
ply axiomatize float operations and their arithmetic properties). We  
estimate that, just like conversion from assembly language to ma-  
chine code, the expansion of these macro-instructions can realisti-  
cally be certified by rigorous testing: the need for a formal proof is  
lower than for the rest of the compiler.

4. Compiler passes and their correctness proofs

4.1 Instruction selection and reassociation

The first pass of the compiler is a translation from external Cmi-

nor to internal Cminor that recognizes the combined operations and  
addressing modes provided by the target processor. It also encodes  
the external Cminor operators that do not correspond to a proces-  
sor instruction, e.g. binary not is expressed using not-or. Addition-  
and logical operations.

4.2 RTL generation

The translation from internal Cminor to RTL is conceptually sim- 
ple: the structured control is encoded as a flow graph; expressions  
are decomposed into sequences of RTL instructions; pseudo-regis-  
ters are generated to hold the values of Cminor variables and inter-  
nmediate results of expression evaluations. The decomposition of  
expressions is made trivial by the prior conversion to internal  
Cminor: every operation becomes exactly one Iop instruction.

4.3 Simulation and correctness proofs

The simulation property is summarized by the following diagram:

A first difficulty is that we need to generate fresh pseudo-regis- 
ters and fresh CFG nodes, and incrementally enrich the CFG  
with new instructions. Additionally, the translation can fail, e.g.  
in case of a reference to an undeclared local variable. This would  
cause no programming difficulties in a language featuring muta- 
tion and exceptions, but these luxuries are not available in Coq,  
which is a pure functional language. We therefore use a monadic  
programming style using the state-and-error monad: every tran- 
slation that computes a result of type \( \alpha \) becomes a function with  
return type \( \text{mon } \alpha = \text{state} \rightarrow (\text{OK}(\text{state} \times \alpha) \mid \text{Error}) \). The  
state type comprises the current state of the CFG, an infinite sup- 
ply of fresh graph nodes, and an infinite supply of fresh registers.  
For instance, the translation function for expressions is of the form  
\( \text{transl} \_\text{expr} \text{map} \text{mut} a \ r_d \ n_d : \text{mon node} \), where \( a \) is a Cminor  
expression, \( \text{map} \) a translation environment mapping local variables  
and let-variables to pseudo-registers, and \( \text{mut} \) the set of Cminor  
variables assigned to in \( a \). If successful, the translation adds to  
the flow graphs the instructions that compute the value of \( a \), leave its  
value in register \( r_d \), and branch to graph node \( n_d \). The return value  
is the first node of this instruction sequence. Similar functions exist  
for expression lists, conditional expressions (with two continuation  
nodes \( n_{\text{true}} \) and \( n_{\text{false}} \), and statements (with \( n + 2 \) continuation  
nodes for normal continuation, return continuation, and \( n \) exit  
continuations corresponding to the \( n \) blocks in scope). The follow- 
ing excerpt from \( \text{transl} \_\text{expr} \) should give the flavor of the trans- 
lation:

\[
\begin{align*}
\text{match } a \text{ with } & \mid \text{Eop op al } \Rightarrow \\
& \text{do } r_l \leftarrow \text{alloc_regs map} \text{mut} a l; \\
& \text{do } n_o \leftarrow \text{add_instr (Iop op rl rd);} \\
& \text{transl_exprlist map mut al rl no }
\end{align*}
\]

Inspired by Haskell, do \( x \leftarrow b \) is a user-defined Coq notation  
standing for bind \( \lambda x. b \).

A crucial property of these monadic translation functions is that  
the state evolves in a monotone fashion: if \( s \) is the input state and \( s' \)  
the output state, all nodes and registers fresh in \( s' \) are also fresh  
in \( s \), and all node to instruction mappings present in \( s \) are present  
identically in \( s' \). This ensures in particular that all RTL executions  
valid in the CFG of \( s \) also hold in the CFG of \( s' \). A large part of  
the correctness proof is devoted to establishing and exploiting this  
monotonicity property.

The correctness of the translation follows from a simulation  
argument between the executions of the Cminor source and the  
RTL translation, proved by induction on the Cminor evaluation  
derivation. In the case of expressions, the simulation property is  
summarized by the following diagram:

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

where \( \text{transl} \_\text{expr} \text{map} \text{mut} a \ r_d \ n_d : \text{mon node} \), the  
left column is the Cminor evaluation, and the right column is the  
execution of several RTL instructions in the CFG of state \( s' \).  
Full lines stand for hypotheses, dotted lines for conclusions. The  
invariant \( I \) expresses that the values of Cminor local variables and  
let variables given by \( E \) and \( L \) are equal to the values of the  
corresponding registers (according to \( \text{map} \)) in \( R \). The precondition  
\( P \) says that \( r_d \) is either the register associated to the variable that  
\( a \) refers to, or a non-fresh register not mapped with any variable  
otherwise. The postcondition \( Q \) says that \( R'(r_d) = v \) and \( R'(r) =  
R(r) \) for all registers \( r \) distinct from \( r_d \), not mapped with any  
variable, and not fresh in the initial state \( s \). In other terms, the
The generated instruction sequence stores value \( v \) in register \( r_d \) and preserves the values of temporary registers generated earlier to hold the results of other sub-expressions.

35 such diagrams are proved, one for each Cminor evaluation rule. An easy induction on the evaluation derivation in Cminor then shows the main correctness theorem: if a Cminor program \( p \) evaluates to value \( v \) and successfully translates to the RTL program \( p' \), then \( p' \) evaluates to value \( v \).

### 4.3 Optimizations at the RTL level

The next passes in our compiler are optimizations based on dataflow analysis performed on the RTL intermediate language. Two such optimizations are currently implemented: constant propagation and common subexpression elimination. (A detailed description of an earlier development of these optimizations can be found in [3].) Both optimizations make use of generic solvers for forward dataflow inequations of the form

\[
A(s) \geq T(l, A(l)) \quad \text{if } s \text{ is a successor of } l
\]

\[
A(l) \geq A_0(l) \quad \text{for all } l
\]

where \( T \) is a transfer function and the unknowns \( A(l) \) range over an ordered type of abstract values (compile-time approximations). Two such solvers are provided as modules (functions) parameterized over the structure \( A \) of abstract values. The first is Kildall’s worklist algorithm, applicable if \( A \) has a least upper bound operation. The second performs simple propagation over extended basic blocks, setting \( A(l) = \top \) in the solution for all program points \( l \) that have several predecessors.

In the case of constant propagation, the abstract values are functions from pseudo-registers to value approximations \( \top \) \( \mid \bot \) \( \mid \text{int}(i) \mid \text{float}(f) \mid \text{addrGlobal}(id + \delta) \), and the transfer function \( T \) is the obvious abstract interpretation of the semantics of RTL instructions over these abstract values. Kildall’s algorithm is used to solve the dataflow inequations.

The code transformation exploiting the results of this analysis is straightforward: top instructions become “load constant” instructions if the values of all argument registers are statically known, or are turned into cheaper immediate forms of the instructions if some argument values are known; \( \text{cond} \) instructions where the condition can be statically evaluated are turned into \( \text{nop} \) to the appropriate successor. The structure of the control-flow graph is preserved (no nodes are inserted), making this transformation easy to express.

The correctness proof is, as usual, a simulation argument performed by induction on the RTL evaluation of the input code. The simulation diagrams are of the following form:

\[
\begin{array}{c}
\text{R : A(l)} \\
\text{if} \ r, r' \text{ interfere;} \\
\text{if} \ r \neq r' \text{且} r' \text{ interferes;} \\
\text{if} \ r \neq l \text{ and} l \text{ interferes;} \\
\text{if} \ r \text{ is a hardware non-temporary register or a Loca1 stack slot of the same type as} r, \text{for all} r.
\end{array}
\]

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.

The translation from RTL to LTL replaces references to pseudo-register \( r \) by references to \( \sigma(r) \) if the latter is a hardware register. If \( \sigma(r) \) is a stack slot, reload or spill instructions between \( \sigma(r) \) and temporary hardware registers are inserted around the instruction, and the temporary register is used instead of \( r \). Dead instructions (side-effect-free instructions whose result is not live) are eliminated, as well as \( \text{mov} \) and \( \text{spill} \) instructions where \( \sigma(r) \neq \sigma(r') \).

In parallel, calling conventions are enforced: moves are introduced between the function parameters and arguments and results of function calls, on the one hand, and fixed locations prescribed

---

\( ^1 \) We designate 2 integer registers and 3 float registers as temporaries, not used by register allocation. This does not follow compiler textbooks, which prescribe re-running register allocation to assign registers to reload and spill temporaries. However, this practice is hard to prove correct, if only w.r.t. termination of register allocation.
by the PowerPC calling conventions (as functions of the type signatures of the functions and calls) on the other hand. These moves are of the “parallel move” kind, since some locations can be both sources and destinations. It is folklore that such parallel moves can be translated to sequences of individual moves using at most one temporary register of each type. Showing the correctness of the parallel move compilation algorithm is one of the most difficult proofs in this project; the proof was kindly contributed by Rideau and Serpette [29].

The correctness of this translation is proved using simulation diagrams of the following form:

\[ l, R, M \quad l \vdash R \approx L \quad l, L, M \]

\[ s, R', M' \quad s \vdash R' \approx L' \quad s, L', M' \]

putting in correspondence the execution of one RTL source instruction (left) with that of zero, one or several LTL transformed instructions (right). The invariant \( l \vdash R \approx L \) is defined as

\[ R(r) = L(\sigma(r)) \text{ for all pseudo-registers } r \text{ live at point } l \]

This property – that we have never seen spelled explicitly in compiler literature – captures concisely and precisely the essence of register allocation: allocation preserves the values of all registers, with the exception of dead registers whose values are irrelevant. Simple properties of the invariant above, combined with those of the interference graph, provide most of the proofs for the diagrams. The proofs are however marred by lots of administrative work to enforce non-overlapping hypotheses between the various kinds of locations, especially between allocatable and temporary hardware registers.

4.5 Linearization

The next compilation step is a translation from LTL to Linear that linearizes the control-flow graph. Discussions of linearization in textbooks focus on trace picking heuristics that reduce the number of jumps introduced, but consider the actual production of linearized code trivial. Our first attempts at proving directly the correctness of a trace picking algorithm that builds linearized code and shortens branches to branches on the fly showed that this is not so trivial. A better approach is to decompose linearization in a way that clearly separates the heuristic parts from the correctness-critical parts. First, branches to branches are eliminated by rewriting the CFG (tunneling). Second, an enumeration of the reachable nodes of the CFG is produced as an ordered list. Third, the CFG instructions are put in a list according to this order. Every instruction is unconditionally followed by a goto to the label of its successor in the CFG. Fourth, gotos that branch to an immediately following label are eliminated.

The correctness of the tunneling and final goto elimination transformations is trivial to prove. More interestingly, the actual linearization according to a pre-computed enumeration can be proved correct under surprisingly weak hypotheses: any enumeration where every CFG reachable node appears exactly once produces Linear code semantically equivalent to the input LTL code. For the naive enumeration function we used, this property can be proved easily on the function itself. However, more advanced trace picking heuristics can also be used as “black boxes” and their results validated a posteriori, like we did for graph coloring.

4.6 Laying out the stack frame

The next translation step, from Linear to Mach, makes explicit the layout of the stack frame for each function. This is another example of a compiler pass that is considered obvious in compiler literature, yet surprisingly hard to prove correct – in particular because this is the first and only pass that changes the memory layout.

Our stack frames are standard, comprising Cminor stack data, Local and Outgoing stack slots, an area to save callee-save registers and the return address register, and a back pointer to the frame of the caller. Since our memory model does not ensure that our frame is allocated just below that of our caller, the back pointer is necessary to access our Incoming stack slots (stack-allocated parameters), which reside in the Outgoing part of the caller’s frame. The compiler determines the size and layout of the stack frame after scanning the Linear code to compute how many slots of each kind and how many callee-save registers are used. It then translates references to stack slots into actual sp-relative loads and stores. It also adds function prologues and epilogues that save and restore the values of used callee-save registers.

This simple code transformation is surprisingly difficult to prove correct – so much so that we broke the proof in two sub-proofs, using two different semantics for Mach code. (See figure 1 again for intuitions.) The first semantics is of the form \( G, fn, sp \triangleright v, v, R, S, P, M \rightarrow v', R', S', P', M' \). Here, the stack block \( M(sp) \) is the same as in Linear and contains only Cminor stack data; the other components of the stack frame are stored in the \( S \) environment, which has the same structure as a memory block contents, but resides outside the heap (and therefore cannot be aliased with other memory blocks). \( P \) is similar to \( S \), but represents the caller’s frame. The Mach instructions getstack, setstack and getparent are interpreted as accesses inside \( S \) and \( P \). Equipped with these semantics, we show simulation diagrams of the form

\[ \delta, L, M \quad L \approx R, S, P \quad T(\delta), R, S, P, M \]

\[ \delta', L', M' \quad L' \approx R', S', P' \quad T(\delta'), R', S', P', M' \]

The \( L \approx R, S, P \) invariant expresses agreement between the location-to-value mapping \( L \) and the register-to-mapping \( R \) plus the frame contents \( S \) and \( P \): for all registers, \( L(r) = R(r) \), and for all valid stack slots \( s \), \( L(s) = load(B, \text{chunk}, \delta) \) where \( \text{chunk} \) and \( \delta \) are the memory chunk and offset appropriate for \( s \), and \( B \) is \( S \), \( S \) or \( P \) respectively for local, outgoing, and incoming slots \( s \).

The main difficulty of the proof, apart from exploiting the absence of overlap between the different areas of the stack frame, is type-related. Linear environments \( L \) are untyped, meaning that \( L(l \leftarrow v)(l) = v \) even if the type of \( v \) does not match that of location \( l \). However, this does not hold for memory block contents such as \( S \); storing a float value in an integer memory chunk and reading it back returns underf, not the float value. To show the commutation of the diagram in the case of the setstack operation, we therefore add the hypothesis that the Linear code \( \delta \) is well typed in the same trivial, int-or-float type system that we used for Cminor. This type system is weakly sound, in the following sense: if the program does not get stuck at run-time, locations of type int always contain integer, pointer or underf values (which are preserved by an int store followed by an int load), and locations of type float always contain float or underf values (preserved by a float store followed by a float load).

The well-typedness of the input Mach code could be established by prior type reconstruction. However, we already performed type reconstruction on the RTL code for the purposes of register allo-
tion, so we chose instead to prove that the subsequent transformations (to LTL, then to Linear, then to Mach) are type-preserving.

We are not done yet: it remains to show semantic preservation for the "real" semantics of Mach code $G, fn, sp ⊢ \bar{r}, R, M \rightarrow \bar{i}, R', M'$ where the whole of the activation record is stored in memory (in block $M(sp)$), the $S$ and $P$ state components disappears, and getstack and setstack instructions are interpreted as $sp$-relative memory loads and stores. To this end, we show that the two Mach semantics are equivalent for the Mach code produced in this pass:

$$
\begin{align*}
\bar{i}, R, S, P, M & \vdash M \leq M_1 \land S, P \approx sp/M_1 \\
\bar{i}', R', S', P', M' & \vdash M'_1 \leq S', P' \approx sp/M'_1 \\
\bar{i}', R', M'_1 & \vdash M'_1 \leq S', P' \approx sp/M'_1
\end{align*}
$$

Here, $M \leq M_1$ means that all memory blocks in $M_1$ are at least as big as in $M$ and have the same contents on offsets that are valid in $M$. Also, $S, P \approx sp/M_1$ means that the contents of $S$ and $P$ agree with those of the first two memory blocks in the chained frame list starting at $sp$ in $M_1$. The proof of this semantic equivalence is difficult, involving non-trivial reasoning on memory operations and on the fact that stack blocks for all simultaneously active function activations are pairwise distinct. A required hypothesis, which came as a complete surprise to us, is that the size of every activation record must be less than $2^{32}$ bytes, otherwise some stack slots cannot be accessed using 32-bit offsets from $sp$. The compiler therefore checks the size of activation records and fails if this bound is exceeded.

4.7 Generation of PowerPC assembly code

The final pass of the compiler translates from Mach code to abstract syntax for PowerPC assembly language. The translation is a straightforward expansion of Mach instructions, operators and addressing modes to canned PowerPC instruction sequences. The correctness proof is large because many cases have to be considered, but presents no difficulties.

5. Experimental evaluation

Size of the development The size of the development can be estimated from the line counts given in figure 2. The whole development, which took one man-year, represents approximately 35000 lines of Coq, plus 15000 lines of code directly written in Caml. The Coq function definitions that represent the compiler itself accounts for 13% of the Coq source. In other terms, the certification is about 8 times bigger than the code it proves. The remaining 87% comprise 8% of specifications (mostly, operational semantics for the various languages), 22% of statements of theorems and lemmas and supporting definitions, 50% of proof scripts and 7% of directives and custom tactics. Concerning the sizes of individual passes, the largest by far is register allocation, which is not surprising given that it is the most sophisticated part of the compiler. Some passes involving large definitions and proofs by case (optimizations over RTL, PPC generation) also have high line counts but are much simpler to prove correct.

Performance of generated code The C to Cminor front-end is not operational at the time of this writing; therefore, the compiler was benchmarked only on small programs hand-translated to Cminor. Figure 3 compares the execution times of these programs compiled by our certified back-end with those of equivalent C programs compiled by gcc at various optimization levels.

The test suite is too small to draw definitive conclusions. Yet, it seems that the certified compiler delivers performance only slightly inferior to gcc at optimization level 1, and much better than gcc at optimization level 0. Therefore, the original performance goal – being competitive with the non-optimizing compilers used for critical embedded software – appears met. The figures for gcc -O3 suggest that 20–30% performance gains are possible if we certify a few more optimizations, notably loop optimizations.

Compilation times are entirely acceptable: between 50% and 200% of the compilation times for gcc -O1.

6. Lessons learned, limitations, and perspectives

On the choice of semantics We used big-step semantics for the source language, "mixed-step" semantics for the intermediate languages, and small-step semantics for the target language. A consequence of this choice is that our semantic preservation theorems hold only for terminating source programs: they all have premises of the form "if the source program evaluates to result $r$", which do not hold for non-terminating programs. This is unfortunate for the application area targeted: critical embedded software are generally reactive programs that never terminate... Equally disappoint-

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structures, proof auxiliaries</td>
<td>268</td>
<td>70</td>
<td>-</td>
<td>514</td>
<td>933</td>
<td>489</td>
<td>2274</td>
</tr>
<tr>
<td>Integers, floats, memory model, values</td>
<td>221</td>
<td>18</td>
<td>851</td>
<td>1159</td>
<td>2872</td>
<td>449</td>
<td>5570</td>
</tr>
<tr>
<td>Cminor semantics</td>
<td>-</td>
<td>-</td>
<td>257</td>
<td>-</td>
<td>-</td>
<td>21</td>
<td>278</td>
</tr>
<tr>
<td>Instruction recognition, reassociation</td>
<td>693</td>
<td>-</td>
<td>-</td>
<td>186</td>
<td>462</td>
<td>99</td>
<td>1440</td>
</tr>
<tr>
<td>RTL semantics and type reconstruction</td>
<td>194</td>
<td>-</td>
<td>196</td>
<td>253</td>
<td>673</td>
<td>94</td>
<td>1410</td>
</tr>
<tr>
<td>RTL generation</td>
<td>279</td>
<td>-</td>
<td>-</td>
<td>858</td>
<td>1626</td>
<td>224</td>
<td>2987</td>
</tr>
<tr>
<td>Optimizations over RTL</td>
<td>1235</td>
<td>-</td>
<td>-</td>
<td>633</td>
<td>1447</td>
<td>325</td>
<td>3640</td>
</tr>
<tr>
<td>LTL semantics</td>
<td>-</td>
<td>-</td>
<td>354</td>
<td>61</td>
<td>151</td>
<td>56</td>
<td>622</td>
</tr>
<tr>
<td>Register allocation</td>
<td>800</td>
<td>521</td>
<td>-</td>
<td>1907</td>
<td>4897</td>
<td>375</td>
<td>8500</td>
</tr>
<tr>
<td>Linear semantics</td>
<td>-</td>
<td>-</td>
<td>238</td>
<td>20</td>
<td>34</td>
<td>48</td>
<td>340</td>
</tr>
<tr>
<td>Linearization</td>
<td>133</td>
<td>-</td>
<td>-</td>
<td>412</td>
<td>749</td>
<td>129</td>
<td>1423</td>
</tr>
<tr>
<td>Mach semantics</td>
<td>-</td>
<td>-</td>
<td>494</td>
<td>366</td>
<td>710</td>
<td>96</td>
<td>1666</td>
</tr>
<tr>
<td>Layout of activation records</td>
<td>116</td>
<td>-</td>
<td>-</td>
<td>469</td>
<td>987</td>
<td>156</td>
<td>1728</td>
</tr>
<tr>
<td>PPC semantics</td>
<td>-</td>
<td>-</td>
<td>479</td>
<td>6</td>
<td>9</td>
<td>33</td>
<td>527</td>
</tr>
<tr>
<td>PPC generation</td>
<td>407</td>
<td>-</td>
<td>-</td>
<td>700</td>
<td>1705</td>
<td>127</td>
<td>2939</td>
</tr>
<tr>
<td>Compiler driver, Cminor parser, PPC printer</td>
<td>32</td>
<td>704</td>
<td>-</td>
<td>43</td>
<td>98</td>
<td>61</td>
<td>938</td>
</tr>
<tr>
<td>Total</td>
<td>4378</td>
<td>1313</td>
<td>2869</td>
<td>7587</td>
<td>17353</td>
<td>2782</td>
<td>36282</td>
</tr>
</tbody>
</table>
Figure 3. Performance of generated PowerPC code. Times are in seconds on a 500 MHz G3 processor (Apple Cube). In parentheses, performance relative to that of gcc -O1 (higher percentages are better).

<table>
<thead>
<tr>
<th>Test program</th>
<th>gcc -O0</th>
<th>gcc -O1</th>
<th>gcc -O3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>1.26s (94%)</td>
<td>4.02s (29%)</td>
<td>1.18s (100%)</td>
</tr>
<tr>
<td>Almabench</td>
<td>5.35s (99%)</td>
<td>6.09s (86%)</td>
<td>5.28s (100%)</td>
</tr>
<tr>
<td>FFT</td>
<td>1.32s (97%)</td>
<td>1.58s (81%)</td>
<td>1.28s (100%)</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>0.71s (96%)</td>
<td>1.64s (41%)</td>
<td>0.66s (100%)</td>
</tr>
<tr>
<td>Integral</td>
<td>0.47s (53%)</td>
<td>1.10s (22%)</td>
<td>0.25s (100%)</td>
</tr>
<tr>
<td>Quicksort</td>
<td>1.04s (103%)</td>
<td>2.15s (50%)</td>
<td>1.08s (100%)</td>
</tr>
<tr>
<td>SHA1</td>
<td>4.42s (93%)</td>
<td>13.77s (29%)</td>
<td>4.10s (100%)</td>
</tr>
</tbody>
</table>

Certified transformation vs. certified verification

In the current state of our compiler, there is only one algorithm (the George-
The usability of Cminor for compiling higher-level source languages is unclear. Function pointers are supported, enabling the compilation of object-oriented and functional languages. However, Java, C++ and ML would also require primitive support for exceptions, which demands a major rework of the compiler. Tail-call optimization also requires significant work, as it is delicate to perform when some function arguments are stack-allocated.

7. Related work

We have already discussed the relations between certified compilers and other approaches to trusted compilation in section 2. In this section, we focus the discussion on proofs of correctness for compilers. A great many on-paper proofs for program analyses and compiler transformations have been published – too many to survey here, but see Dave’s bibliography [10]. In the following, we restrict ourselves to discussing correctness proofs that involve on-machine verification.

As is often the case in the area of machine-assisted proofs, Moore was one of the first to mechanically verify semantic preservation for a compiler [20, 21], although for a custom language and a custom processor that are not commonly used.

The Veriflix project [13] had goals broadly similar to ours: the construction of mathematically correct compilers. The only part that led to a machine-checked proof was the formal verification in PVS of a compiler for a subset of Common Lisp to Transputer code [11], neither of which are used for critical embedded systems.

Strecker [33] and Klein and Nipkow [15] certified non-optimizing byte-code compilers from a subset of Java to a subset of the Java Virtual Machine using Isabelle/HOL. They did not address compiler optimizations nor generation of actual machine code. Another certification of a byte-code compiler is that of Grégoire [14], for a functional language.

In the context of the German Verisoft initiative, Leinenbach et al [17] and Strecker [34] formally verified a compiler for a C-like language called C0 down to DLX assembly code using the Isabelle/HOL proof assistant. This compiler appears to work in a single pass and to generate unoptimized code.

Rhodium [18] is a domain-specific language to describe program analyses and transformations. From a Rhodium specification, both executable code and an automatically-verified proof of semantic preservation are generated. Rhodium is impressive by the degree of automation it achieves, but applies only to the optimization phases of a compiler and not to the non-optimizing translations from one language to another, lower-level language.

Another project that concentrates on optimizations is the certified framework for abstract interpretation and dataflow analysis of Cachera et al [6] and Pichardie [27]. Like us, they use the Coq proof assistant for their certification.

8. Conclusions

The certified back-end presented in this paper is a first step, and much work remains to be done to meet the initial goal of certifying a compiler that is practically usable in the context of industrial formal methods. However, the present work provides strong evidence that this objective can eventually be met. We hope that this work also contributes to renew scientific interest in the semantic understanding of compiler technology, in operational semantics “on machine”, and in integrated environments for programming and proving.
Acknowledgments

E. Ledinot made us aware of the need for a certified C compiler in the aerospace industry. Several of the techniques presented here were discussed and prototyped with members of the Concert INRIA coordinated research action, especially Y. Bertot, S. Blazy, B. Grégoire and L. Rideau. Their formalization of dataflow analyses and constant propagation builds on that of B. Grégoire. L. Rideau and B. Serpette contributed the hairy correctness proof for parallel moves. D. Doligez proved properties of finite maps and type reconstruction for RTL. G. Necula argued convincingly in favor of a posteriori verification of static analyses and optimizations.

References


