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 incorrect 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.

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

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 establishing a given correctness property $Prop(S, C)$ 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 *Spec*, then C satisfies *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 *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 *Comp* is a total function from source programs to either compiled code (written $Comp(S) = \texttt{Some}(C)$) or an error (written $Comp(S) = \texttt{None}$). 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 *Comp* accompanied with a formal proof of the following theorem:

$$\forall S, C, \quad Comp(S) = \texttt{Some}(C) \Rightarrow Prop(S, C) \qquad (i)$$

In other terms, either the compiler reports an error or produces code that satisfies the desired correctness property. Notice that the trivial compiler $Comp(S) = \texttt{None}$ 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 *CComp* that either fails ($CComp(S) = \texttt{None}$) or returns both a compiled code C and a proof π of the property $Prop(S, C)$ ($CComp(S) = \texttt{Some}(C, \pi)$). The proof π 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" π that do not establish $Prop(S, C)$. 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 *Comp* is complemented by a *verifier*: a boolean-valued function $Verif(S, C)$ that verifies the property $Prop(S, C)$ by static analysis of S and C. To obtain formal guarantees, the verifier must be itself certified. That is, we must prove that

$$\forall S, C, \quad Verif(S, C) = \texttt{true} \Rightarrow Prop(S, C)$$

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 π of $Prop(S, C)$: 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 $CComp(S)$ returns either $\texttt{None}$ or $\texttt{Some}(C, A)$, 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

$$\forall S, A, C, \quad Verif(S, C, A) = \texttt{true} \Rightarrow Prop(S, C) \qquad (ii)$$

In the case of pure proof-carrying code, A is a proof term and *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 *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 *CComp* is a certifying compiler and *Verif* a correct verifier, the following function is a certified compiler:

$$\begin{array}{l} Comp(S) = \\ \quad \texttt{match } CComp(S) \texttt{ with} \\ \quad \mid \texttt{None} \rightarrow \texttt{None} \\ \quad \mid \texttt{Some}(C, A) \rightarrow \texttt{if } Verif(S, C, A) \texttt{ then Some}(C) \texttt{ else None} \end{array}$$

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

Symmetrically, let *Comp* be a certified compiler and Π be a Coq proof term for theorem (i). Via the Curry-Howard isomorphism, Π is a function that takes S, C and a proof of $Comp(S) = \texttt{Some}(C)$ and returns a proof of $Prop(S, C)$. A certifying compiler can be defined as follows:

$$
\begin{array}{l}
CComp(S) = \\
\quad \texttt{match}\ Comp(S)\ \texttt{with} \\
\quad \mid \texttt{None} \rightarrow \texttt{None} \\
\quad \mid \texttt{Some}(C) \rightarrow \texttt{Some}(C, \Pi\ S\ C\ \pi_{eq})
\end{array}
$$

(Here, π_{eq} is a proof term for the proposition $Comp(S) = \texttt{Some}(C)$, 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 the compilation of all source programs, not just for S), it can conceivably be specialized for S and C using partial evaluation techniques.

Compositionality Compilers are generally decomposed into several passes that communicate through intermediate languages. It is fortunate that both certified and certifying compilers can also be decomposed in this manner. If $Comp_1$ and $Comp_2$ are certified compilers from languages L_1 to L_2 and L_2 to L_3, respectively, their (monadic) composition

$$
\begin{array}{l}
Comp(S) = \\
\quad \texttt{match}\ Comp_1(S)\ \texttt{with} \\
\quad \mid \texttt{None} \rightarrow \texttt{None} \\
\quad \mid \texttt{Some}(I) \rightarrow Comp_2(I)
\end{array}
$$

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 $CComp_1$ and $CComp_2$ are certifying compilers from languages L_1 to L_2 and L_2 to L_3, and $Verif_1$, $Verif_2$ the accompanying verifiers, a certifying compiler *cum* verifier from L_1 to L_3 can be constructed as follows:

$$
\begin{array}{l}
CComp(S) = \\
\quad \texttt{match}\ CComp_1(S)\ \texttt{with} \\
\quad \mid \texttt{None} \rightarrow \texttt{None} \\
\quad \mid \texttt{Some}(I, A_1) \rightarrow \\
\qquad \texttt{match}\ CComp_2(I)\ \texttt{with} \\
\qquad \mid \texttt{None} \rightarrow \texttt{None} \\
\qquad \mid \texttt{Some}(C, A_2) \rightarrow \texttt{Some}(C, (A_1, I, A_2)) \\
\\
Verif(S, C, (A_1, I, A_2)) = \\
\quad Verif_1(S, I, A_1) \wedge Verif_2(I, C, A_2)
\end{array}
$$

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. Second, 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 theorem (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:

$a ::=$	id	local variable
	$\mid id = a$	variable assignment
	$\mid op(\vec{a})$	constants and arithmetic
	$\mid \texttt{load}(chunk, a)$	memory load
	$\mid \texttt{store}(chunk, a_1, a_2)$	memory store
	$\mid \texttt{call}(sig, a, \vec{a})$	function call
	$\mid a_1\ \texttt{\&\&}\ a_2$	sequential boolean "and"
	$\mid a_1\ \texttt{||}\ a_2$	sequential boolean "or"
	$\mid a_1\ \texttt{?}\ a_2 : a_3$	conditional expression
	$\mid \texttt{let}\ a_1\ \texttt{in}\ a_2$	local binding
	$\mid n$	reference to `let`-bound variable

Operators in expressions include all the arithmetic, logical and comparison operations of the C language. Unlike in C, there is no operator overloading nor implicit conversions: distinct arithmetic operations are provided over 32-bit integers and 64-bit floats, as well as operators for float/integer conversions and integer zero- and sign-extensions. Loads and stores are given a memory address, explicitly computed using address arithmetic (byte-addressed memory), and a "memory chunk" indicating the kind, size and signedness of the memory datum being accessed, e.g. "64-bit float" or "8-bit sign-extended integer". The `let` construct, written in de Bruijn notation, enables sharing the evaluation of sub-expressions.

Statements:

$s ::=$	$a;$	expression evaluation
	$\mid \texttt{if}\ a\ \{\vec{s}_1\}\ \texttt{else}\ \{\vec{s}_2\}$	conditional
	$\mid \texttt{loop}\ \{\vec{s}\}$	infinite loop
	$\mid \texttt{block}\ \{\vec{s}\}$	delimited block
	$\mid \texttt{exit}\ n;$	block exit
	$\mid \texttt{return};\ \mid \texttt{return}\ a;$	function return

The `exit` n statement terminates prematurely the n enclosing `block` constructs. Combined with infinite loops and `if-else` statements, blocks and exits suffice to express efficiently all reducible control-flow graphs, notably those arising from C loops. No general `goto` statement is provided.

Functions: $fn ::= \texttt{fun}(\vec{id}) : sig\ \{\ \texttt{stack}\ n; \texttt{vars}\ \vec{id}; \vec{s}\ \}$

Programs: $prog ::= \texttt{functions}\ \ldots id = fn \ldots;$
$\texttt{vars}\ \ldots id[n] \ldots;$
$\texttt{main}\ id$

In addition to parameters, local variable declarations and a function body (a list of statements), a function definition comprises a type signature sig (see "Static typing" below) and a declaration of how many bytes of stack-allocated space it needs. Variables in Cminor do not reside in memory and their address cannot be taken. However, the Cminor producer can explicitly stack-allocate some data (such as, in C, arrays and scalar variables whose addresses are taken). The $\texttt{stackaddr}_n$ nullary operator returns a pointer within the stack block at byte offset n.

Programs are composed of a set of named function definitions, a set of global variables along with their sizes, and a distinguished function name representing the program entry point (`main` function). Addresses of functions and global variables can be taken using the $\texttt{addrsymbol}(id)$ nullary operator.

Dynamic semantics The dynamic semantics of Cminor is given in big-step, structured operational semantics. The semantics is completely deterministic and imposes a left-to-right evaluation order. The following judgments are defined using Coq inductive predicates (inference rules):

$$
\begin{array}{rcll}
G, sp, L & \vdash & a, E, M \Rightarrow v, E', M' & \text{(expressions)} \\
G, sp, L & \vdash & \vec{a}, E, M \Rightarrow \vec{v}, E', M' & \text{(expression lists)} \\
G, sp & \vdash & s, E, M \Rightarrow out, E', M' & \text{(statements)} \\
G, sp & \vdash & \vec{s}, E, M \Rightarrow out, E', M' & \text{(statement lists)} \\
G & \vdash & fn(\vec{v}), M \Rightarrow v, M' & \text{(function calls)} \\
 & \vdash & prog \Rightarrow v & \text{(whole programs)}
\end{array}
$$

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 `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 `let`-bound variables inside expressions. For statement evaluations, out (the "outcome" of the evaluation) expresses how the statement terminated: either normally by falling through the next statement or prematurely through an `exit` or `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. The same memory model is used for all languages in our compiler. To give the flavor of this model, the memory operations provided are *alloc*, *free*, *load* and *store*. *load* and *store* return an error if the accessed block was freed or the memory access is outside the bounds of this block. They satisfy the following "good variable" properties: if $load(M, chunk, b, \delta) = v$ and $store(M, chunk', b', \delta') = M'$ and the addresses do not overlap ($b \neq b'$ or $\delta + |chunk| \leq \delta'$ or $\delta' + |chunk'| \leq \delta$), then $load(M', chunk, b, \delta) = v$; and if $store(M, chunk, b, \delta) = M'$ and $chunk$ agrees with $chunk'$ in kind and size (but may differ in signedness), then $load(M', chunk', b, \delta) = cast(v, chunk')$. Here, *cast* represents the zero- or sign-extension of integers or the rounding of 64-bit floats to 32-bit floats possibly prescribed by $chunk'$. In all other cases of *store* followed by *load* (partial overlap of the memory ranges; disagreement over the chunks), the result of the *load* is `undef`. Thus, the memory model abstracts over the byte-level representation of integers and floats.

Static typing Cminor is equipped with a trivial type system having only two types: `int` and `float`. (Memory addresses have static type `int`.) In addition, function definitions and function calls are annotated with signatures sig giving the number and types of arguments, and an optional type for the result. All operators are monomorphic; therefore, the types of local variables can be inferred from their uses and are not declared. The primary purpose of this trivial type system is to facilitate later transformations (see sections 4.4 and 4.6): by itself, the type system is too weak to give type soundness properties (absence of run-time type errors). In particular, applying an operator to values of the wrong type, using an `undef` value in any way, or calling a function whose signature differs from that given at the `call` site cause the program to get stuck: its semantics are not defined and the compiler can (and does) generate incorrect code for this program. It is the responsibility of the Cminor producer to avoid these situations, e.g. by using a richer type system.

External Cminor vs. internal Cminor The language we have described so far is the processor-independent interface offered to the front-end. Internally, the back-end works over a slightly different Cminor language that includes processor-specific constructs. The conversion from external to internal Cminor is performed on the fly via optimizing "smart constructors" described in section 4.1. The internal Cminor constructs that differ from those of external Cminor are the following:

Expressions:

$a ::= op_{ppc}(\vec{a})$	constants and arithmetic
$\mid$ `load`$(chunk, mode, \vec{a})$	memory load
$\mid$ `store`$(chunk, mode, \vec{a_1}, a_2)$	memory store
$\mid c_1 \; ? \; a_2 : a_3$	conditional expression

Conditional expressions:

$c ::=$ `true` $\mid$ `false`	always true, always false
$\mid$ `cond`$(cond, \vec{a})$	test $cond$ over $\vec{a}$
$\mid c_1 \; ? \; c_2 : c_3$	nested conditional

Statements:

$s ::=$ `if` $c \; \{\vec{s_1}\}$ `else` $\{\vec{s_2}\}$	conditional

The set of operators op_{ppc} for internal Cminor includes combined operators that reflect what the target processor can do in one instruction. In the case of the PowerPC, internal operators include integer operations with one immediate operand, as well as combined operations (rotate-and-mask, multiply-add). Similarly, `load` and `store` expressions in internal Cminor compute the memory address from the value of several sub-expressions $\vec{a}$ combined using one of the processor's addressing modes $mode$. Finally, a syntactic class of conditional expressions (those expressions that are evaluated for their truth value) is introduced, along with a specific evaluation judgment $G, sp, L \vdash c, E, M \Rightarrow b, E', M'$ where b is `true` or `false`.

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:

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

RTL control-flow graph:

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

RTL functions:

$fn ::=$ `fun`$(\vec{r}) : sig$
$\{$ `stack` n; `start` l; `graph` g $\}$

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 are statically typed using the same trivial type system as Cminor. Each register can be assigned a type `int` or `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:

$$\begin{array}{rcll} G, g, sp & \vdash & l, R, M \to l', R', M' & \text{(one instruction)} \\ G, g, sp & \vdash & l, R, M \xrightarrow{*} l', R', M' & \text{(several instructions)} \\ G & \vdash & fn(\vec{v}), M \Rightarrow v, M' & \text{(function call)} \end{array}$$

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.

$$\frac{g(l) = \texttt{op}(op, \vec{r}, r_d, l') \qquad v = \texttt{eval\_op}(op, R(\vec{r}))}{G, g, sp \vdash l, R, M \to l', R\{r_d \leftarrow v\}, M}$$

$$\frac{\begin{array}{c} g(l) = \texttt{call}(sig, r_f, \vec{r}, r_d, l') \qquad G(R(r_f)) = fn \\ fn.\texttt{sig} = sig \qquad G \vdash fn(R(\vec{r})), M \Rightarrow v, M' \end{array}}{G, g, sp \vdash l, R, M \to l', R\{r_d \leftarrow v\}, M'}$$

$$\frac{\begin{array}{c} \texttt{alloc}(M, 0, fn.\texttt{stack}) = (sp, M_1) \qquad R = \{fn.\texttt{params} \leftarrow \vec{v}\} \\ G, fn.\texttt{graph}, sp \vdash fn.\texttt{start}, R, M_1 \xrightarrow{*} l', R', M_2 \\ fn.\texttt{graph}(l') = \texttt{return}(r) \qquad R'(r) = v \qquad M' = \texttt{free}(M_2, sp) \end{array}}{G \vdash fn(\vec{v}), M \Rightarrow v, M'}$$

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:

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

LTL control-flow graph:

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

LTL functions:

$fn ::=$	$\texttt{fun}() : sig$	
	$\{\ \texttt{stack}\ n; \texttt{start}\ l; \texttt{graph}\ g\ \}$	

Stack slots:

$s ::=$	$\texttt{Local}(\tau, \delta)$	local variables
$\mid$	$\texttt{Incoming}(\tau, \delta)$	incoming parameters
$\mid$	$\texttt{Outgoing}(\tau, \delta)$	outgoing arguments

In stack slots, τ is the intended type of the slot (`int` or `float`) and δ 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:

$$\frac{\begin{array}{c} G(L(r_f)) = fn \qquad fn.\texttt{sig} = sig \\ G \vdash fn, \texttt{entryfun}(L), M \Rightarrow v, L', M' \end{array}}{G, g, sp \vdash \texttt{call}(sig, r_f) :: b, L, M \to b, \texttt{exitfun}(L, L'), M'}$$

$$\frac{\begin{array}{c} \texttt{alloc}(M, 0, fn.\texttt{stack}) = (sp, M_1) \\ G, fn.\texttt{graph}, sp \vdash g(fn.\texttt{start}), L, M_1 \xrightarrow{*} \texttt{return}, L', M_2 \\ M' = \texttt{free}(M_2, sp) \end{array}}{G \vdash fn, L, M \Rightarrow v, L', M'}$$

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

Location l	$\texttt{entryfun}(L)(l)$	$\texttt{exitfun}(L, L')(l)$
r	$L(r)$	$L(r)$ if r is callee-save
		$L'(r)$ if r is caller-save
$\texttt{Local}(\tau, \delta)$	`undef`	$L(\texttt{Local}(\tau, \delta))$
$\texttt{Incoming}(\tau, \delta)$	$L(\texttt{Outgoing}(\tau, \delta))$	$L(\texttt{Incoming}(\tau, \delta))$
$\texttt{Outgoing}(\tau, \delta)$	`undef`	$L'(\texttt{Incoming}(\tau, \delta))$

Another peculiarity of locations is that distinct stack slots may overlap, that is, they will be later mapped to overlapping memory areas. For instance, $\texttt{Outgoing}(\texttt{float}, 0)$ overlaps with $\texttt{Outgoing}(\texttt{int}, 0)$ and $\texttt{Outgoing}(\texttt{int}, 1)$: 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: $(L\{loc_1 \leftarrow v\})(loc_2) = L(loc_2)$ only if loc_1 and loc_2 do not overlap; it does not suffice that $loc_1 \neq loc_2$.

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:

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

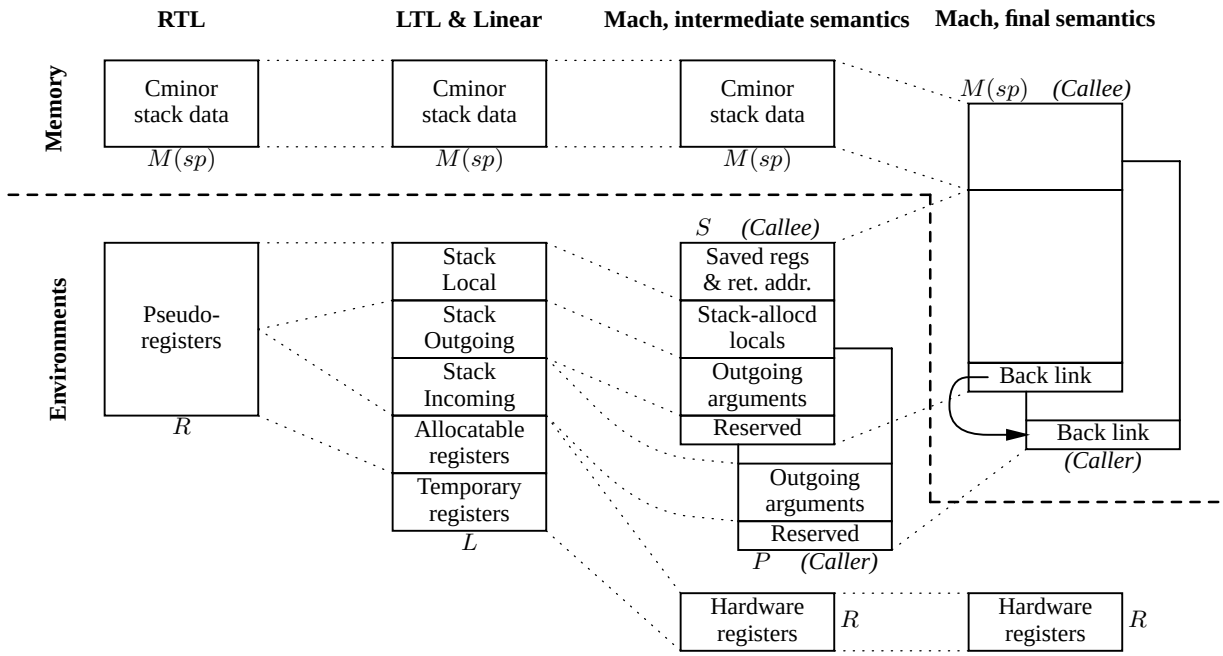


Figure 1. Overview of register allocation and introduction of activation records. For each intermediate language, the placement of function-local data is outlined, either in the memory-allocated activation record (top part) or in non memory-resident execution environments (bottom part).

$\mid$ `goto`(l)	unconditional branch
$\mid$ `cond`$(cond, \vec{r}, l_{true})$	conditional branch
$\mid$ `return`	function return

Linear functions:

$fn ::=$ `fun`$() : sig$
{ `stack` n; `start` l; `code` $\vec{i}$ }

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 `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:

$i ::=$ `setstack`(r, τ, δ)	register to stack move
$\mid$ `getstack`(τ, δ, r)	stack to register move
$\mid$ `getparent`(τ, δ, r)	caller's stack to register move
$\mid \ldots$	as in Linear

In the three new move instructions, τ is the type of the data moved and δ its word offset in the corresponding activation record. The semantics of `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 $G, fn, sp \vdash \vec{i}, R, M \rightarrow \vec{i}', R', M'$, 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 `fmadd` and `fmsub` 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 `fmadd` and `fmsub` 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 `fmadd` and `fmsub` 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 formalization of IEEE float arithmetic, which we have not done (we simply axiomatize float operations and their arithmetic properties). We estimate that, just like conversion from assembly language to machine code, the expansion of these macro-instructions can realistically 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 Cminor 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 processor instruction, e.g. binary not is expressed using not-or. Additionally, algebraic reassociation is performed for integer addition and multiplication, and for shifts and logical "and". About 50 rewriting rules are applied to Cminor expressions. Representative examples of rules are:

$$\begin{array}{rcl}
\texttt{add}(e, \texttt{intconst}(n)) & \rightarrow & \texttt{addi}_n(e) \\
\texttt{add}(\texttt{addi}_n(e_1), \texttt{addi}_m(e_2)) & \rightarrow & \texttt{addi}_{n+m}(\texttt{add}(e_1, e_2)) \\
\texttt{add}(e_1, \texttt{addi}_n(e_2)) & \rightarrow & \texttt{addi}_n(\texttt{add}(e_1, e_2)) \\
\texttt{muli}_m(\texttt{addi}_n(e)) & \rightarrow & \texttt{addi}_{m \times n}(\texttt{muli}_m(e)) \\
\texttt{shl}(e, \texttt{intconst}(n)) & \rightarrow & \texttt{rolm}_{n,(-1) \ll n}(e) \\
\texttt{shru}(e, \texttt{intconst}(n)) & \rightarrow & \texttt{rolm}_{32-n,(-1) \gg n}(e) \\
\texttt{and}(e, \texttt{intconst}(n)) & \rightarrow & \texttt{rolm}_{0,n}(e) \\
\texttt{rolm}_{n_1,m_1}(\texttt{rolm}_{n_2,m_2}(e)) & \rightarrow & \texttt{rolm}_{n_1+n_2,m}(e) \\
 & & \text{with } m = \texttt{rol}(m_1, n_2) \wedge m_2 \\
\texttt{or}(\texttt{rolm}_{n,m_1}(e), \texttt{rolm}_{n,m_2}(e)) & \rightarrow & \texttt{rolm}_{n,m_1 \vee m_2}(e) \\
 & & \text{if } e \text{ has no side-effects}
\end{array}$$

($\texttt{rolm}_{n,m}$ is a left rotation by n bits followed by a logical "and" with m.) While innocuous-looking, these rules are powerful enough to e.g. recognize a $\texttt{rolm}_{3,-1}$ instruction for the common C idiom `(x << 3) | (x >> 29)`.

These rewriting rules are applied in one bottom-up pass and encapsulated as "smart constructor" functions invoked by the Cminor producer. For instance, the `make_add` constructor takes two normalized internal Cminor expressions e_1, e_2 and returns a normalized expression semantically equivalent to $\texttt{add}(e_1, e_2)$. Normalized internal Cminor expressions therefore play the role of abstract syntax for external Cminor expressions.

Proving the correctness of these transformations amounts to showing that the external Cminor evaluation rule for, say, `add` expressions is admissible in internal Cminor if the `add` operator is interpreted by its smart constructor `make_add`. Routine case analysis and inversion on the evaluation derivations of sub-expressions reduces this proof to showing that the rewriting rules above are valid when interpreted as equalities over values. To this end, we had to develop a fairly large and difficult formalization of N-bit machine integers and of the algebraic properties of their arithmetic and logical operations.

4.2 RTL generation

The translation from internal Cminor to RTL is conceptually simple: the structured control is encoded as a flow graph; expressions are decomposed into sequences of RTL instructions; pseudo-registers are generated to hold the values of Cminor variables and intermediate 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.

A first difficulty is that we need to generate fresh pseudo-registers 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 mutation 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 translation that computes a result of type α becomes a function with return type $\texttt{mon}\ \alpha = \texttt{state} \rightarrow (\texttt{OK}(\texttt{state} \times \alpha) \mid \texttt{Error})$. The `state` type comprises the current state of the CFG, an infinite supply of fresh graph nodes, and an infinite supply of fresh registers. For instance, the translation function for expressions is of the form $\texttt{transl\_expr}\ map\ mut\ a\ r_d\ n_d : \texttt{mon node}$, where a is a Cminor expression, map a translation environment mapping local variables and `let`-variables to pseudo-registers, and 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_{true} and n_{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 following excerpt from `transl_expr` should give the flavor of the translation:

```
match a with
| Eop op al =>
    do rl <- alloc_regs map mut al;
    do no <- add_instr (Iop op rl rd nd);
       transl_exprlist map mut al rl no
```

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

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:

$$\begin{array}{ccc}
sp, L, a, E, M & \xrightarrow{\quad I \wedge P \quad} & sp, n_s, R, M \\
\Big\Downarrow & & \vdots\, * \\
sp, L, v, E', M' & \xrightarrow{\quad I \wedge Q \quad} & sp, n_d, R', M'
\end{array}$$

where $\texttt{transl\_expr}\ map\ mut\ a\ r_d\ n_d\ s = \texttt{OK}(s', n_s)$, 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 *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

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

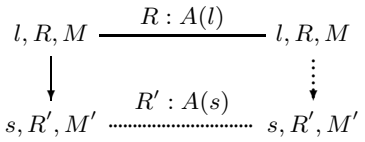
$$\begin{array}{rcl} A(s) & \geq & T(l, A(l)) \text{ if } s \text{ is a successor of } l \\ A(l) & \geq & A_0(l) \text{ for all } l \end{array}$$

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 (functors) parameterized over the structure $\mathcal{A}$ of abstract values. The first is Kidall's worklist algorithm, applicable if $\mathcal{A}$ 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 $\top \mid \bot \mid \texttt{int}(i) \mid \texttt{float}(f) \mid \texttt{addrglobal}(id + \delta)$, 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: `Iop` 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; `Icond` instructions where the condition can be statically evaluated are turned into `Inop` 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:



where the left column is one step of execution in the input code and the right column is one step of execution in the optimized code. The $R : A(l)$ condition expresses agreement between the actual register values and the results of the static analysis: if the analysis predicts that a register has a known value, it must have this value at run-time (in R).

Common subexpression elimination is performed by value numbering over extended basic blocks. The abstract values are pairs of a value numbering (a mapping from registers to value numbers) and a set of equations between value numbers. The dataflow inequations are solved with the extended basic block solver, sparing us from computing l.u.b. of these abstractions (which would be too expensive). In the simulation arguments, the agreement relation $R : A(l)$ is defined as equation satisfiability: there must exist a mapping from abstract value numbers to concrete values that satisfies the equations between value numbers stated in $A(l)$.

4.4 Register allocation

The next and most involved pass is a translation from RTL to LTL that performs register allocation, insertion of spills and reloads, and explicitation of calling conventions. The register allocator is based on coloring of an interference graph, in the style of Chaitin [7], using the George-Appel [12] heuristic for graph coloring and coalescing. The notable feature of our implementation is that the George-Appel coloring procedure is not certified, but the colorings it returns are verified correct a posteriori by a certified verifier. Graph coloring is a paradigmatic example of a computation where verifying the results is much simpler than certifying the algorithm itself.

Register allocation starts with a standard liveness analysis performed by backward dataflow analysis. We reuse our generic implementation of Kildall's algorithm after inverting the edges of the control-flow graph. An interference graph is then built following Chaitin's rules and proved to contain all the necessary interference edges. 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. Finally, type reconstruction is performed on the RTL input code, associating an `int` or `float` type to every pseudo-register. This enables the graph coloring to choose hardware registers and stack slots of the appropriate class.

Coloring of the interference graph is then performed by an implementation of the George-Appel algorithm written directly in Caml and not certified. It returns a mapping σ from pseudo-registers to locations. The correctness conditions for this mapping are:

1. $\sigma(r) \neq \sigma(r')$ if r and r' interfere;
2. $\sigma(r) \neq l$ if r and l interfere;
3. $\sigma(r)$ is a hardware non-temporary register or a `Local` stack slot of the same type as r, for all r.

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.

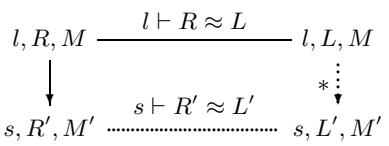
The translation from RTL to LTL replaces references to pseudo-register r by references to $\sigma(r)$ if the latter is a hardware register. If $\sigma(r)$ is a stack slot, `reload` or `spill` instructions between $\sigma(r)$ and temporary hardware registers are inserted around the instruction, and the temporary register is used instead of r.[1] Dead instructions (side-effect-free instructions whose result is not live) are eliminated, as well as `move` r, r' instructions where $\sigma(r) = \sigma(r')$.

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:



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, `goto`s that branch to an immediately following label are eliminated.

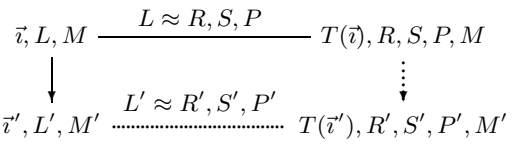
The correctness of the tunneling and final `goto` elimination transformations is trivial to prove. More interestingly, the actual linearization according to a pre-computed enumeration can be proved correct under surprisingly weak hypotheses: any enumeration where every CFG reachable node appears exactly 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 $G, fn, sp \vdash \vec{i}, R, S, P, M \rightarrow \vec{i}', R', S', P', M'$. 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 this semantics, we show simulation diagrams of the form



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, $L(s) = \texttt{load}(B, chunk, \delta)$ where $chunk$ and δ 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 `undef`, 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 $\vec{i}$ 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 `undef` values (which are preserved by an `int` store followed by an `int` load), and locations of type `float` always contain float or `undef` 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 alloca-

	Code (in Coq)	Code (in Caml)	Specifications	Theorems	Proofs	Other	Total
Data structures, proof auxiliaries	268	70	-	514	933	489	2274
Integers, floats, memory model, values	221	18	851	1159	2872	449	5570
Cminor semantics	-	-	257	-	-	21	278
Instruction recognition, reassociation	693	-	-	186	462	99	1440
RTL semantics and type reconstruction	194	-	196	253	673	94	1410
RTL generation	279	-	-	858	1626	224	2987
Optimizations over RTL	1235	-	-	633	1447	325	3640
LTL semantics	-	-	354	61	151	56	622
Register allocation	800	521	-	1907	4897	375	8500
Linear semantics	-	-	238	20	34	48	340
Linearization	133	-	-	412	749	129	1423
Mach semantics	-	-	494	366	710	96	1666
Layout of activation records	116	-	-	469	987	156	1728
PPC semantics	-	-	479	6	9	33	527
PPC generation	407	-	-	700	1705	127	2939
Compiler driver, Cminor parser, PPC printer	32	704	-	43	98	61	938
Total	4378	1313	2869	7587	17353	2782	36282

Figure 2. Size of the development (in non-blank lines of code)

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 $G, fn, sp \vdash \vec{\imath}, R, M \rightarrow \vec{\imath}', R', M'$ 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:

$$\begin{array}{ccc} \vec{\imath}, R, S, P, M & \xrightarrow{\quad M \le M_1 \wedge S, P \approx sp/M_1 \quad} & \vec{\imath}, R, M_1 \\ \Big\downarrow & & \vdots \\ \vec{\imath}', R', S', P', M' & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & \vec{\imath}', R', M_1' \end{array}$$

$$M' \le M_1' \wedge S', P' \approx sp/M_1'$$

Here, $M \le M_1$ 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 1500 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-

Test program	Certified compiler	`gcc -O0`	`gcc -O1`	`gcc -O3`
AES	1.26s (94%)	4.02s (29%)	1.18s (100%)	0.88s (134%)
Almabench	5.35s (99%)	6.09s (86%)	5.28s (100%)	5.20s (101%)
FFT	1.32s (97%)	1.58s (81%)	1.28s (100%)	1.27s (101%)
Fibonacci	0.71s (96%)	1.64s (41%)	0.68s (100%)	0.55s (124%)
Integral	0.47s (53%)	1.10s (22%)	0.25s (100%)	0.18s (139%)
Quicksort	1.04s (103%)	2.15s (50%)	1.08s (100%)	0.96s (112%)
SHA1	4.42s (93%)	13.77s (29%)	4.10s (100%)	3.43s (119%)

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)

ing is the fact that our proofs, read between the lines, actually show much stronger results: function calls and returns, as well as reads and writes to global variables, are in one-to-one correspondence between the source and compiled code. However, actually proving such correspondences necessitates a shift away from big-step semantics and towards small-step (transition) semantics, which account both for terminating and diverging executions as finite or infinite transition sequences, respectively.

For the intermediate languages, it should not be too difficult to replace the "mixed-step" semantics by pure small-step (transition) semantics, at some cost in the clarity of the proofs (e.g. the call stack needs to be manipulated explicitly). For Cminor, a small-step semantics is more difficult. Reduction semantics (where expressions are rewritten at each execution step) do not commute with compiler transformations such as RTL generation, which complicates tremendously the correctness proof of the latter. Transition semantics using explicit program points [9] appear more amenable to proving the correctness of RTL generation, but require restricting the Cminor language, prohibiting function calls within expressions.

On memory consumption Our memory model assumes an infinite memory: allocation requests always succeed. This conflicts with the intended applications to embedded software, where precise memory bounds is a must-have. It would be trivial to bound the total memory size in the memory model and modify the semantics so that they report failure if this bound is exceeded. The problem is that compilation can increase the memory needs of a program: stack allocation of spilled pseudo-registers increases arbitrarily the size of activation records. Therefore, the semantic preservation theorem would no longer hold. In other terms, it is hopeless to prove a stack memory bound on the source program and expect this resource certification to carry out to compiled code: stack consumption, like execution time, is a program property that is not preserved by compilation. The correct alternative is to establish the memory bound directly on the compiled code. If recursion and function pointers are not used, which is generally the case for critical embedded software, a simple, certified static analysis over Mach code that approximates the call graph can provide the required memory bounds.

On the choice of intermediate languages The RTL language and its variants that we used as intermediate languages have several advantages: they are simple, very well understood in compiler folk lore, and easy to give formal semantics for. However, they do not lend themselves to the efficient implementation of static analyses and optimizations such as global CSE. We considered using SSA in intermediate languages to enable efficient static analysis algorithms such as global value numbering, but encountered two problems: formal dynamic semantics for SSA are not obvious (but see [5] for a recent exploration), and it is difficult for proofs to (locally) exploit the (global) SSA property. Functional representations such as A-normal forms could offer some of the benefits of SSA with clearer semantics. The alternative that looks most promising is to keep the RTL language unchanged, but perform static analyses in untrusted Caml code that could internally convert to SSA and back for efficiency reasons, then verify the correctness of the results of the analysis. This way, only the verifier needs to be certified, but not the analyses.

On programming in Coq At the beginning of this project, it was not clear that the functional subset of the Coq specification language was a powerful enough programming language for writing a compiler. It turns out that it is, and being forced to program in a purely functional style was actually a pleasant experience for a long-time ML programmer such as the author. Three difficulties were encountered, however. The first is the paucity of efficient purely functional data structures in the Coq standard library. We had to develop and prove correct some of our own data structures, but in the end were able to implement everything on top of only three data structures: finite sets, finite maps and a functional "union-find", all implemented as binary trees with reasonable $O(\log n)$ efficiency.

The second difficulty is the way complex pattern-matchings are represented internally in Coq. For instance, the matching against `C1, C1 => a | _, _ => b` of two terms of a 10-constructor data type `C1 | ... | C10` is represented internally as the complete matching having 10×10 cases, 99 of which being `b`. This causes much duplication in proofs and explosion of the size of the extracted code. Various workarounds were needed, such as the recourse to auxiliary classification functions.

The last difficulty is the treatment of general recursion in Coq. The logic underlying Coq only supports structural recursion over tree-shaped structures, generalizing primitive recursion. General terminating recursion is provided by a library as Noetherian induction over well-founded ordered types [2, chap.15]. Noetherian induction is turned into structural induction over a proof that no infinite decreasing sequences exist, which is an amazing feat of logic, but is very hard to utilize when writing and proving programs. We therefore avoided general recursion as much as possible. For instance, the formalizations of Kildall's algorithm presented in [1, 15, 3, 8] require that the semi-lattice of approximations is well founded in order to guarantee termination. It is however painful to ensure well-foundedness for approximations that are mappings from pseudo-registers to abstract values: the number of registers must be bounded, either a priori or by the set of registers actually mentioned in the current function, which significantly complicates proofs. We cut corners here in a major way by simply bounding a priori the number of iterations performed by Kildall's algorithm, and turning off the corresponding optimization if a fixpoint is not reached within that number of iterations (which can be chosen suitably huge).

Certified transformation vs. certified verification In the current state of our compiler, there is only one algorithm (the George-

Appel graph coloring) that is not proved correct in Coq, but whose results are checked a posteriori by a verifier certified in Coq. In retrospect, we should have followed this verification approach for other algorithms, notably Kildall's dataflow inequation solver (whose results can be checked easily, avoiding the problem with general recursion described above), RTL type reconstruction (checking a register type assignment is easier than inferring it), and the "parallel move" algorithm (whose results can also be checked easily by symbolic interpretation of the sequence of elementary moves produced). In every case, proving the correctness of the verifier is significantly easier than proving that of the algorithm.

Going further, whole transformation passes could be performed by untrusted code, then checked for semantic equivalence using Hoare logic and symbolic evaluation as in [24]. However, it is not clear yet how difficult it is to prove mechanically the correctness of those semantic equivalence checkers. Still, this is a direction worth investigating before embarking in the certification of complex transformation algorithms.

On program extraction Automatic extraction of Caml code from Coq specifications is another feature of the Coq environment on which we depended crucially and which turned out to work satisfactorily. The major difficulty for extraction is dealing with specifications that use dependent types intensively: the resulting Caml code is not necessarily typeable in Caml, and it can contain large, costly proof computations that do not contribute to the final result and are eliminated by techniques akin to slicing [19]. However, our functional specifications of the compiler were written without dependent types, in pedestrian ML style. The delicate aspects of extraction were therefore not exercised. The only difficulties encountered were with the compile-time reductions that the extraction tool performs to produce cleaner, more efficient Caml code. It sometimes goes too far with β-reductions and η-expansions, resulting in expensive computations such as graph coloring being performed too many times. Our first extraction of the compiler was therefore correct, but ran very slowly. Manual patching of the extracted Caml code was necessary to undo this "optimization".

On proving in Coq Coq proofs are developed interactively using a number of tactics as elementary proof steps. The sequence of tactics used constitutes the proof script. Building such scripts is surprisingly addictive, in a videogame kind of way, but reading and reusing them when specifications change is difficult. Our proofs make good use of the limited proof automation facilities provided by Coq, mostly `eauto` (Prolog-style resolution), `omega` (Presburger arithmetic) and `congruence` (equational reasoning). However, these tactics do not combine automatically and significant manual massaging of the goals is necessary before they apply.

Coq also provides a dedicated language for users to define their own tactics. We used this facility occasionally, for instance to define a "monadic inversion" tactic that recursively simplifies hypotheses of the form $(\texttt{do}\ x\ \texttt{<-}\ a; b)\ s = \texttt{OK}(s', r)$ into $s_1;\ x;\ a\ s = \texttt{OK}(s_1, x);\ b\ s_1 = \texttt{OK}(s', r)$. There is no doubt that a Coq expert could have found more opportunities for domain-specific tactics and could have improved our proof scripts.

On the usability of Cminor as an intermediate language Cminor was designed to allow relatively direct translation of a large subset of C: everything except `goto`, unstructured `switch`, un-prototyped functions, variable-argument functions, `long long` arithmetic, `setjmp`/`longjmp`, and `malloc`/`free`. None of these features are crucial for embedded critical software, except possibly `goto`. Dynamic allocation (`malloc` and `free`) are easy to add to Cminor since the memory model does not require that uses of *alloc* and *free* follow a stack discipline. Another planned easy extension of Cminor is a multi-way branch enabling more efficient compilation of structured `switch` statements. The big unknown is whether `goto` is needed, in which case a complete rework of Cminor semantics is in order. The remainder of the compiler (RTL and down) would be unaffected, however, since nowhere we assume that the CFG is reducible.

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 Verifix 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. Our 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

[1] G. Barthe, P. Courtieu, G. Dufay, and S. M. de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In *Proceedings of AMAST'02*, volume 2422 of *LNCS*, pages 41–59. Springer-Verlag, 2002.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[3] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs, Workshop TYPES 2004*, LNCS. Springer-Verlag, 2005.

[4] S. Blazy and X. Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 280–299. Springer-Verlag, 2005.

[5] J. O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing code generation from SSA form: A comparison between two formal correctness proofs in Isabelle/HOL. In *Proc. COCV Workshop (Compiler Optimization meets Compiler Verification)*, 2005.

[6] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *European Symposium on Programming 2004*, volume 2986 of *LNCS*, pages 385–400. Springer-Verlag, 2004. Extended version to appear in *Theor. Comp. Sci.*

[7] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Symp. Compiler Construction*, volume 17(6) of *SIGPLAN Notices*, pages 98–105. ACM Press, 1982.

[8] S. Coupet-Grimal and W. Delobel. A Uniform and Certified Approach for Two Static Analyses. Research report 24-2005, Laboratoire d'Informatique Fondamentale, Marseille, France, April 2005.

[9] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[10] M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[11] A. Dold and V. Vialard. A mechanically verified compiling specification for a Lisp compiler. In *Proc. FST TCS 2001*, volume 2245 of *LNCS*, pages 144–155. Springer-Verlag, 2001.

[12] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, 1996.

[13] G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*, pages 201–230. Springer-Verlag, 1999.

[14] B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml*. PhD thesis, University Paris 7, 2003.

[15] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Mar. 2004. To appear in ACM TOPLAS.

[16] J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? In *Proc. Compiler Construction '98*, volume 1383 of *LNCS*, pages 65–79. Springer-Verlag, 1998.

[17] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Int. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–11. IEEE Computer Society Press, 2005.

[18] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *32nd symp. Principles of Progr. Lang*, pages 364–377. ACM Press, 2005.

[19] P. Letouzey. A new extraction for Coq. In *Types for Proofs and Programs, Workshop TYPES 2002*, volume 2646 of *LNCS*, pages 200–219. Springer-Verlag, 2003.

[20] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[21] J. S. Moore. *Piton: a mechanically verified assembly-language*. Kluwer, 1996.

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Prog. Lang. Syst.*, 21(3):528–569, 1999.

[23] G. C. Necula. Proof-carrying code. In *24th symp. Principles of Progr. Lang*, pages 106–119. ACM Press, 1997.

[24] G. C. Necula. Translation validation for an optimizing compiler. In *Prog. Lang. Design and Impl. 2000*, pages 83–95. ACM Press, 2000.

[25] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *28th symp. Principles of Progr. Lang*, pages 142–154. ACM Press, 2001.

[26] S. L. Peyton Jones, N. Ramsey, and F. Reig. C−−: a portable assembly language that supports garbage collection. In *PPDP'99: International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 1–28. Springer-Verlag, 1999.

[27] D. Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés*. PhD thesis, University Rennes 1, Dec. 2005.

[28] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *LNCS*, pages 151–166. Springer-Verlag, 1998.

[29] L. Rideau and B. P. Serpette. Coq à la conquête des moulins. In *Journées françaises des langages applicatifs (JFLA 2005)*, pages 169–180. INRIA, 2005.

[30] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proc. FLoC Workshop on Run-Time Result Verification*, 1999.

[31] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symp. Principles of Progr. Lang*, pages 1–13. ACM Press, 2004.

[32] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.

[33] M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 63–77. Springer-Verlag, 2002.

[34] M. Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, April 2005.

[35] L. D. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. *Electr. Notes Theor. Comput. Sci.*, 65(2), 2002.